A SYSTEMATIC APPROACH TO DELIVERING INSTRUCTION-LEVEL PARALLELISM IN EPIC SYSTEMS

BY

JOHN WOLLENBURG SIAS

B.S.,University of Illinois at Urbana-Champaign, 1997
M.S., University of Illinois at Urbana-Champaign, 1999

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Electrical Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2005

Urbana, Illinois

ABSTRACT

Computer systems designed under the explicitly parallel instruction computing (EPIC) paradigm rely extensively on compiler technology to deliver the instruction-level parallelism (ILP) required for them to achieve high levels of performance. While manifold techniques have been proposed in the literature for delivering such parallelism, this dissertation is unique in integrating and applying a comprehensive suite of techniques, embodied in the IMPACT Research Compiler, to a concrete system, comprised of the SPEC CINT2000 benchmarks and the Intel Itanium 2 platform. These techniques include advanced pointer analysis, aggressive cross-file procedure inlining, targeted region formation, profile-guided optimizations, and, most importantly, aggressive and pervasive use of predication and control speculation.

The collective effect of these techniques is evaluated with real-system measurements, showing them to achieve a 1.20 average (up to 1.59) speedup relative to classically optimized code and a 1.70 average (up to 2.51) speedup relative to code compiled with the Gnu GCC compiler. Achieving these results in the real-machine environment required advances in region formation heuristics, optimization, and speculation methods. Modern application tendencies toward decreased instruction locality and increased control-intensiveness made finding successful (sufficient and stable) ILP transformations more challenging, requiring adaptation and intensification of previous techniques.

As we look both to newer applications and to achieving the next level of ILP, more sweeping program transformations are called for, but the compiler is today hard pressed to deliver stable transformations with current techniques. High-performance compilation of nonnumeric codes for EPIC chafes against programming style and control flow structure. This dissertation provides a thoroughgoing evaluation of the classical approach to

ILP formation in an extant EPIC system, illustrating the circumstances that dictate the success of EPIC features in achieving high performance in contemporary benchmarks.

Soli Deo Gloria!

ACKNOWLEDGMENTS

The author is first indebted to his dissertation advisor, Prof. Wen-mei W. Hwu, for patiently fostering this work and for providing an environment in which the author could find his professional identity. For his patience with my admitted idiosyncrasies and wrestlings with vocation, he deserves especial thanks.

As is typical of any dissertation involving the IMPACT compiler, the author is indebted to a long string of developers, both those whose work formed the seeds out of which this work emerged, and those whose ongoing work nurtured it as it developed. A partial listing includes especially David August, Ronald Barnes, Daniel Connors, Kevin Crozier, Robert Kidd, Matthew Merten, Erik Nystrom, James Player, Shane Ryoo, Ian Steiner, and Sain-zee Ueng. To his dear friends among this motley crew, who offered timely advice, encouragement and criticism as the need arose, the author offers his profound thanks.

The work presented here relied largely on the generosity and general supportiveness of several partners in the corporate sphere. The author's Itanium-specific efforts (and much of the motivation for this work) began with summer projects on the (now-defunct) IBM IA64 compiler at the Centre for Advanced Studies at the IBM Toronto Research Lab. In addition to monetary support and technical challenges, the author appreciates being exposed to the wide array of expertise in the TOBEY, TPO, and Java teams.

The work leading directly to this dissertation began with the donation of a prototype Itanium compiler (based on a then-antiquated version of IMPACT), for which the efforts of Jim Pierce must be acknowledged, and prototype Itanium systems from Intel Corporation. Throughout this work, the author has been able to rely on Carole Dulong and

Daniel Lavery for technical advice and support. Also at Intel, John Crawford provided early machines and support.

Support from Hewlett-Packard has also enabled this work. Real-machine results would have been impossible without the outstanding Perfmon system developed for the Linux kernel by Stephane Eranian and David Mosberger of HP Labs. On the production side, Richard Hank and James McCormick have always been ready to lend advice on thorny technical problems. HP has also supported this work with the generous donation of much equipment, including some rare prerelease machines. Bob Rau, Michael Schlansker, and Vinod Kathail also contributed much to this project over the years.

The author must also acknowledge the monetary support of the National Defense Science and Engineering Graduate Research Fellowship Program, which underwrote the costs of my early graduate career. The author also thanks the members of his committee, Professors Wen-mei Hwu, Michael Loui, Steven Lumetta, Matthew Frank and Craig Zilles, for their patience, kind attention and helpful guidance.

Finally, for the patient support of his family during the several years of this work, which often separated him from them more than either would have liked, the author offers his deepest gratitude.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

1 PROLEGOMENA

Parallel execution, the concurrent processing of multiple operations in a programmable computing device, continues to increase in importance as a component of microprocessor performance. With physical limits (for example, the speed of electrons; minimum designable feature size; nonidealities of conductive, semi-conductive, and resistive materials; planar circuit layout constraints; and practical limits on design complexity [1, 2]) dictating increasingly rigid limits on the maximum practicable processing speed for any given logical operation, the only alternative for designers who need to achieve higher computing performance is to perform more operations in parallel.

Parallel execution exists in many forms, but is most commonly described as the execution of multiple threads of control on independent processors or at least in independent thread contexts [3]. Exploitation of such a model (multiprocessing or thread-level parallelism) requires an explicitly parallel program or set of programs. Since most programs are written in a serial fashion (popular, imperative languages such as C and C++ have no inherent model for parallel algorithmic expression), this model often seems "unnatural" and is generally difficult and expensive to use, except for certain classes of problems in which parallel programs may be written conveniently or in which special compilers can extract parallel threads of control from an essentially serial program.

1.1 Instruction-Level Parallelism

This dissertation is concerned with a different level, or granularity, of parallelism that applies within a single thread of execution. *Instruction-level parallelism* (ILP), meaning the contemporaneous execution of multiple machine instructions within a single program context, using a single instruction pointer, is a more natural form of parallel execution for traditional serial programs [4]. Opportunities for such parallelism appear when the program is decomposed into machine instructions as groups of instructions that are not constrained (by dependences) to be executed serially. The conventional wisdom suggests that the approach of decomposition and instruction-parallel execution is more easily implemented than a broader or more regular form of parallelism (e.g., task-level parallelism or vector parallelism; while these, and other, more regular and less centralized forms of parallelism are ideally more efficient than ILP, there has been little success putting most ordinary programs into a form that takes advantage of them). ILP has thus been recognized for over 20 years to be an effective compromise between the desire for parallel execution and the need for efficient parallelization of common programs, not written with parallel intent [4]. When decomposed for execution into machine instructions, programs typically exhibit at least some suitability for instruction-level parallel execution. The degree to which this is exploitable depends on two factors: first, the amenability of programs to instruction-level parallel execution; and, second, the means employed in compiler, architecture, and microarchitecture to extract and exploit available parallelism.

Typical integer programs[1] exhibit some degree of inherent, latent ILP, but various constructs interfere with exposing, enhancing, and exploiting it. Historically speaking, there has been considerable disagreement among researchers as to what degree of ILP is exploitable in increasing the number of instructions executed per cycle (IPC), (which equates to an increase in performance) [6–9]. This is an important concern, as the

[1] In the common parlance, "integer" or "nonnumeric" programs are differentiated from "floating-point" or "numeric" programs, the execution time of the latter being dominated by computational loops, generally using floating-point arithmetic, on regular, matrix or vector data structures. Many forms of parallelism, including ILP, are easy to identify in floating-point applications, so this dissertation concerns itself primarily with the integer class, as represented by SPEC CINT2000 [5].

amount of ILP available bounds the success of ILP exploitation techniques, at least in a the view of hardware (or an ILP "purist"). As will be explored shortly, ILP compiler techniques rely on path specialization, which can enhance optimization, possibly resulting in better performance without an increase in IPC. This does not happen in hardware. Whether this "compiler optimization bonus" can be called a success for ILP or merely an optimization enhancement is a possible, if abstruse, argument. Generally, though, it is accepted that the ILP limit studies do in some way bound the potential for success of ILP-based compiler techniques.

Practical constraints limit the accessibility of ILP in real programs, and different interpretations of these constraints led to different estimates of available ILP. Lam and Wilson explained the importance of multipath execution and control speculation in exposing ILP in programs, criticizing the single-thread, VLIW approach for being unable to effectively expose ILP in the presence of complex, data-dependent control flow [10]. Lee et al. emphasized the importance of region boundaries and register renaming in determining the degree of ILP exploitable in SPEC CINT95 [11] applications on EPIC architectures. Their work emulates an idealized, but limited, compile-time scheduler by preventing scheduling across function and/or loop boundaries (idealized, that is, when compared to a typical compiler, but limited with respect to the dynamic trace). Using the unit-latency, infinite-issue machine abstraction typical of ILP limit studies, they find an average ILP of 6.8 (without renaming) and 12.2 (with renaming). With all window restrictions lifted, they report an IPC of 32.9 [12]. While it is simple to lift restrictions in abstract simulation, however, it is difficult to translate this result into, for example, useful insight for compiler work. To be useful, an estimate of available ILP must consider the context (architecture, applications, compiler) within which the ILP is to be exploited. ILP limit studies are encouraging, as they suggest that at least some ILP is available in typical programs, but their predictions must be normed with real-world empirical data.

The role and nature of ILP in microprocessor performance have fluctuated over time. This dissertation relates to one particular model of ILP exploitation, but it is instructive to begin with a consideration of the general ILP landscape. Instruction-level parallel

execution itself exists in two forms, herein referred to as *pipeline parallelism* and *instruction parallelism*. Pipeline parallelism refers to the partial overlapping of the execution times of two or more instructions, as occurs in a pipelined machine. Higher degrees of pipelining (division of work into a larger number of finer stages) allow more instructions to be executing at the same time, ideally achieving performance inversely proportional to the number of stages [13, 14]. Increasing levels of pipelining (up to 31 stages in recent Pentium 4 microprocessor implementations) have historically supported a super-linear scaling of clock frequency with respect to improvements in process technology [15], sustaining the long-expected doubling in computer performance each 18 months[2] [16]. This improvement comes at a substantial cost, however, as increases in clock frequency dramatically increase power consumption, both due to increased switching and to the increased leakage currents inherent to faster transistors, and as the ratio of latch time to compute time (pipeline overhead) steadily increases. Furthermore, longer pipelines mean increased time spent in recovery from branch mispredictions, reducing achievable ILP. One of the most important computing technology developments of the past few years is that superpipelining is no longer a viable path to increasing performance. Work that takes a more modern, power-constrained, view sets a much lower limit on the optimal number of pipeline stages [17], at the very least casting serious doubt on the remaining longevity of the superpipelining approach. Intel's recent cancellation of further Pentium 4 development, short of the previously touted 4-GHz clock frequency mark, in favor of more power-efficient design lines serves as a particularly strong case in point.

The exhaustion of pipeline parallelism leaves instruction parallelism as the subject of examination for performance increase. Instruction parallelism focuses on the simultaneous issue of instructions and the reordering of instructions for more efficient issue, rather than purely the overlapping of their execution times, as in pure pipeline parallelism. While pipelining (aside from disagreements about appropriate degree) is achieved with relatively uniform techniques, here there are two divergent approaches. The first, *dynamic scheduling*, relies on sophisticated hardware features to expose parallelism.

[2] At its apex a few years ago, proponents of this approach projected that these principles could sustain profitable development to 70 pipeline stages or beyond [14].

In a dynamically scheduled design, hardware actively seeks out parallelism among a program's operations, executing them out of order as necessary to avoid simply waiting for results [18]. Operations to be executed are sought along a putative trajectory of future operations, as divined using generally sophisticated branch prediction hardware, within a maximum search distance called the *window*. Accommodating this search for parallelism adds stages to the front of the processor pipeline, increasing the effective penalty of bad branch predictions. Supporting this parallel execution while maintaining correctness requires complex and expensive hardware. One important industry figure, Fred Pollack of Intel Corporation, has proposed the principle that performance increases in rough proportion to the square root of complexity increase in these designs (that is, a doubling in chip features (transistors) yields only a 40% increase in performance) [19]. Already in 2000, studies suggested that the era of bi- to triennial doubling of computing performance could not be sustained for long with with traditional pipeline and capacity scaling techniques [15].

In recent years, then, mainstream microarchitecture development has exploited what may be a final burst of pipeline-lengthening-based performance. Recent systems increased performance by sharply decreasing the clock period, sacrificing (nonpipeline) instruction-level parallelism, and adding complex and potentially inefficient replay mechanisms to meet this goal (e.g., Pentium 4) [20]. These systems rely on essentially classical compiler technology; in this "hectic" model of parallelism, it is the microarchitecture that attempts to ensure in a highly dynamic manner that pipeline parallelism is effectively exploited.

Well before this approach began to show severe signs of stress, and in fact well before it reached its apex, engineers at Hewlett-Packard Laboratories [21] and a few like-minded academic researchers [22] worked on an alternative approach. In this second design approach, *static scheduling*, a much simpler hardware pipeline, having no capacity for instruction reordering, executes operations in-order, according to a static plan devised by the compiler. In order to enable the compiler to express a highly parallel *plan of execution* [23] that will execute efficiently on the available hardware, given the control and data flow constraints of typical applications, the architecture provides certain accommodating

features. These typically include the ability to demarcate groups of instruction for parallel issue, the ability to predicate instructions to avoid frequent control flow redirections, and the ability to speculate operations across potential control and data dependence arcs. This model for static planning of instruction-level parallelism was initially referred to as the very long instruction word (VLIW) model, in reference to the wide groups of operations it grouped for issue each cycle. Exploitation of instruction-level parallelism in this fashion is a long-practiced art. The historical background undergirding this area of work is immense and important, extending well before the important HP Labs work [24, 25]. These VLIW ideas culminated in what is today called the *Explicitly Parallel Instruction Computing* (EPIC) paradigm. Rau and Fisher [26] provide an insightful history of this early era in their *Journal of Supercomputing* article; Schlansker and Rau [27] summarize the entire 1989–1999 development of EPIC ideas at HP Labs, as well as at the University of Illinois at Urbana-Champaign, in another interesting technical report. Those interested in the historical development of these ideas are referred to these excellent works. It is instructive to see how long and in how many various ways engineers and compiler writers have wrestled with the topics to be discussed in these pages.

This dissertation, written over a decade after the proposal of the EPIC approach, discusses the realization of these features and techniques in a second-generation hardware and compiler implementation. Most of the ideas to be discussed in this work were considered or even used in the earlier machines; the vast changes in microarchitecture and in application development over the past 10 years merit their re-evaluation at this juncture. As this work will go on to demonstrate, the evaluation of these techniques in the modern context is much more complex than it once was.

Before leaving the general topic of instruction-level parallelism, it should be noted that the ideal balance of pipeline and instruction parallelism, and the balance between dynamic and static modes of the latter, are still matters of considerable dispute. Of particular interest in this context is work that relies on the compiler to plan the bulk of an effective plan of execution, and then attempts to use the minimum possible degree of dynamism to overcome disruptions in the plan due to cache misses or other unanticipated events [28]. Other work has suggested that EPIC architectures need full dynamic

execution to overcome modern memory hierarchy obstacles [29]. Some have even proposed that, due to the poor scaling of memory latency and inability of software and microarchitectural techniques to hide it, ILP is no longer a broadly exploitable form of parallelism [1, 30]. While this work does not even claim to address these broader issues, the author intends that the lessons of this dissertation about the promise and limitations of the statically planned approach will contribute to a desperately needed, erudite, and informed discussion of these topics as they continue to evolve.

1.2 The Explicitly Parallel Instruction Computing Paradigm

In the explicitly parallel instruction computing (EPIC) paradigm [27, 31], the compiler takes sole responsibility for extracting the potential ILP inherent to programs and expressing it to the hardware for execution. This section introduces the historical and conceptual groundwork for the work presented in this dissertation, work informed by the availability of actual Itanium hardware.

Schlansker and Rau, two prominent proponents and long-time architects of the EPIC approach, delivered an extended presentation of the philosophy and historical events that led to the 11 year development of EPIC that culminated in 1999 with the announcement of the Intel Itanium processor [27]. Their primary motivation, already in the late 1980s, was concern about the scalability of the out-of-order superscalar approach beyond the end of the 1990s. EPIC was to be a means of continuing the superlinear growth of computing performance *within a traditional, sequential code development paradigm* after the then-still-emerging out-of-order scheme lost its way.

The motivating idea of EPIC is that using a superscalar microarchitecture to extract instruction-level parallelism from an essentially sequential or serial architecture[3] (i.e., having no means for the explicit expression of parallelism) is inefficient and unscalable. As an alternative, an EPIC architecture provides a means for the compiler to express a parallel *plan of execution* (POE) directly to the hardware. The hardware, then, has

[3]The term *architecture* as used here is a contract between hardware and software, as in "instruction set architecture," while microarchitecture refers to a particular implementation thereof.

simply to execute the preplanned, already-micro-parallel program. Preceding VLIW designs had provided some means of instruction-level parallel expression, but with limited degrees of success in integer and irregular code. The ideas synthesized into EPIC, including predication, control speculation, and data speculation, were intended to extend the applicability of these historically well-demonstrated techniques to more general classes of programs.[4] This type of architecture has implications for both hardware and compiler design. Most of the aspects have to do with the migration of ILP decisions from execution-time into compile-time. In EPIC, the classical problems from dynamic ILP exploitation may be moved out of the dynamic context, but they do not go away:

- For the compiler to produce an effective POE, run-time behavior must be predictable at compile-time. Most EPIC transformations rely on predictions about the likely flow of program control, often derived from profiling runs [32].

- A subtler aspect of predictability is that the compiler can deal effectively only with latencies it can predict statically. Load latency is a serious problem for EPIC in the modern context [29].

- The POE must be communicated from the compiler to the hardware. The EPIC architecture must contain features that allow the compiler to schedule operations explicitly, such as instruction bundles, and effectively, including large numbers of registers (to prevent live range constraints from becoming an obstacle to parallelism).

- Transfers in control flow break up otherwise parallel sequences of execution. The compiler must be able to schedule across branches to achieve parallel execution, hence *control speculation*.

[4] In fact, Schlansker and Rau [27] view EPIC as "an evolution of VLIW which has absorbed many of the best ideas of superscalar processors, albeit in a form adapted to the EPIC philosophy." EPIC thus aims to extend the VLIW approach to less regular applications, on which the superscalar approach excels.

- Memory operations tend to serialize operations as well. The compiler needs a means of moving loads past previous stores to increase parallelism. This is done either by analysis (proof of transformation safety) or by *data speculation*.

- Control and data speculation are based on predictions of execution bias, but will often be based on faulty guesses. A means must be provided of avoiding or recovering from misspeculation.

- Individual optimization of many interwoven execution paths can be inefficient to the point of infeasibility, and does not address the branch prediction problem. The ability to combine multiple paths with *predication* can enable a more efficient POE in programs that have suitable tendencies.

- The plan of execution may be much larger than the original program, if many versions of code segments are required to achieve high degrees of parallelism. Means like *kernel only modulo scheduling* must be introduced to reduce this code bloat [33].

To summarize, in the EPIC model, which in contrast to the "hectic," out-of-order model might be termed "deliberate," the compiler must produce explicit, static directions for utilization of each processor issue cycle. Placing this onus on the compiler allows the processor to provide wide issue with a minimum of execution core overhead. A simpler, wider pipeline, executing at a comfortably lower clock frequency, has merely to march according to the compiler's plan-of-execution. Complexity is displaced from the chip to the compiler, increasing efficiency *so long as* the compiler can "plan" sufficiently parallel execution and the microarchitecture can execute the plan without too many expensive dynamic anomalies. Which model, "hectic" or "deliberate," is ultimately more efficient for a particular application set is beyond the scope of this dissertation; the importance of strong control-intensive,[5] general-purpose application performance to the general success of EPIC systems, however, is beyond dispute (and, in fact, as discussed in [4], the goal of adding several features to EPIC was extending its applicability to these applications).

[5] A control-intensive program is one that is "branchy," one in which decision-making frequently breaks up the regular issue of instructions.

The compiler can approach the compilation of such programs for EPIC performance in a variety of ways. One may choose an "incremental"[6] approach that uses EPIC features to enhance traditional, global-scheduling-based scheduling schemes, incrementally enhancing the application of traditional compilation models, within the existing program control structure. Contemporary production compilers operate mostly within this model [34, 35]. Alternatively, one may take a "structural" approach, using the new features to perform more radical program control transformations, replicating code, predicating, and speculating freely to generate a vastly different and hopefully more efficient program representation [22, 36–38]. The latter approach is more consistent with EPIC's research lineage. While the literature includes some real-machine evaluations of EPIC's features [39, 40], they are based on compilers taking primarily an incremental approach.

Other research-based evaluations [38] examined the structural interaction of predication and speculation techniques in a hypothetical EPIC architecture, but did not have the benefit of real, implemented machines (or very large, complex benchmarks) to investigate many important considerations—instruction cache effects, microarchitectural implementation constraints, and exception processing, to name a few. This work did elucidate, among other important principles, that, because of the complex interaction of different types of program dependences, the performance impact of a collaborative suite of ILP transformations is greater than the sum of the parts applied individually. This means that techniques must be evaluated in context, considering the effects they will have when applied in concert with other techniques; focused studies of single transformations are likely to provide results with very limited applicability. That EPIC performance was attainable, in varying degrees, in the control-intensive benchmarks of the day was demonstrated with measurements of the scheduled IPC of compiled code, ignoring dynamic effects. This research evaluation is heretofore unreproduced on real hardware, taking into account instruction cache and other secondary costs, and on the larger, more complex, and more control-bound benchmarks of today (as even a passing comparison of

[6]To label this approach "incremental" is not to disparage it, as it is a stable and predictable means of extending conventional compilation techniques to EPIC. It does, however, make less aggressive use of EPIC features than the "structural" approach and therefore offers less opportunity for dramatic results.

SPEC92 and SPEC95 to SPEC2000 will show [5]). These developments necessitate more transformation to achieve expected levels of instruction-level parallelism, complicating the compilation process.

As we today look to synthesize a consistent lesson from these various artifacts, separated by the passage of time and their differing assumptions, we have the benefit of a real, second generation, EPIC implementation, the Intel Itanium 2 microprocessor [41], and a version of the IMPACT compiler that targets this machine [42, 43]. This dissertation provides a holistic and contemporary understanding of EPIC performance from the structural optimization perspective, explaining the benefits and costs of the more radical, structural techniques using experiments on real EPIC hardware and with modern, control-intensive benchmarks. We demonstrate the general effectiveness of these techniques in producing high performance, showing a speedup of up to 2.51 (average 1.70) over GCC and up to 1.59 (average 1.20) relative to IMPACT's classical optimization level. Excluding "nondeterminisms" such as data and instruction cache misses, as most simulation-based experiments [38] have done, IMPACT achieves an average speedup of 1.47 relative to the classically optimized baseline, a result comparable with past investigations. How this was achieved, and what we can learn from the details of these results, are the recurring topics of the following chapters.

1.3 The State of EPIC Compilation Today

The performance of EPIC systems on control-intensive, general-purpose applications depends heavily upon the efficacy of compiler transformations for exposing ILP. Such transformations must be both effective, producing regions of code suitable for highly instruction-parallel execution, and efficient, working in a way that does not unduly disrupt the work of other transformations or the execution of the program. While various individual ILP enhancement techniques are well-known [22, 36, 44–47], applying them together to contemporary programs in a collectively profitable manner is far from a solved problem.

While reasonable heuristics exist to control application of the individual techniques, one must distinguish between *tactical* decision-making and *strategic* understanding in

ILP formation. Individual techniques have been proposed and demonstrated on the basis of tactical application patterns, without necessarily taking the larger context into account. These approaches are tactical both "spatially," not anticipating the effects of the transformation on the rest of the program, and "temporally," not applying the transformation in the light of other potential transformations on the same code segment.

In the research environment, many transformations with potentially expensive side-effects, such as inline expansion of subroutines and Superblock and Hyperblock formation,[7] are today performed very aggressively with the hope of enabling future optimizations that will turn an initial loss into a benefit. Currently this risk is difficult to measure or manage. Given today's larger and more control-intensive programs and more constrained processor front-ends, deleterious effects like instruction cache footprint bloat could reduce the benefit or turn an expected benefit into a loss. In the commercial context, therefore, these optimizations are sharply curtailed (or simply not performed) in practice in an attempt to, at the very least, not make things worse. Compiler developers are left with a difficult choice, between a conservative compiler, incapable of reaping much benefit, or a compiler producing highly varying results. The former is more acceptable to customers, as it generally produces more consistent results, so production compilers accept it; researchers who need cutting-edge performance have to find ways of understanding highly volatile experimental situations to evaluate new techniques.

Because of this chaotic situation, it is today relatively easy to demonstrate suboptimalities in EPIC compiled code, even given a competent compiler and the assumption of accurate profile information in the compilation. Experiments with IMPACT and with production compilers have shown substantial opportunities to improve performance *even by applying only already-implemented techniques* in situations overlooked because of deficiencies in tactical heuristics and compiler phase ordering. August, for example, found opportunities in IMPACT for reversing aggressive if-conversion decisions after subsequent optimization and instruction scheduling [48]. More recently, Triantafyllis et al. found in

[7] These techniques produce specialized code paths by replicating segments of code shared by different traces (see Chapters 4 and 5)[22, 36].

a study of Intel's own highly tuned Itanium compiler that feedback-driven tuning of existing compiler parameters could result in single-benchmark performance improvements of up to 20% [40]. Comparisons of results between the Intel and IMPACT compiler, and among differently configured compilations using IMPACT, reveal similarly pronounced variations.

Lack of a strategic view of ILP formation leads production compiler writers to make only relatively conservative application of the techniques with the potential to reap the highest benefits or, just as easily, the deepest losses. Increasingly aggressive application of such techniques would adversely impact the compiler's *stability*—the property of a compiler that small changes in an input program can be expected to produce small changes in the compiled program's performance. This lukewarm stance is thus not an unjustifiable position, but it is one that obstructs further compiler research and development and limits the potential for gain from EPIC features. This is an especially pressing problem as EPIC processor design is still developing—experiments performed using conservatively compiled code may impact machine design parameters, limiting future performance. Only a few years after Itanium's introduction, the literature contains studies with "negative results" suggesting certain EPIC features have less value than previously believed [39]—these results are based on a production compiler making only a conservative application of EPIC transformations.

Finally, as we look to future EPIC generations, compiler research needs to correct the fact that even current machines are underutilized in control-intensive code. In recent experiments using SPEC CINT2000 and a *6-issue* Itanium 2 microprocessor, the IMPACT compiler, when tuned for reasonably stable performance in an aggressive configuration (later to be identified as **I-CS**), schedules 3.03 non-`nop`[8] instructions per executed cycle, on average, and sustains execution of only around 1.29 instructions per cycle, on average. Shockingly, this is a very competitive result. Attempting to improve on this utilization by simply increasing the aggressiveness of transformations decreases performance stability—possibly increasing performance noticeably on a handful of code segments but not producing a generally useful general-purpose compiler. The aggressiveness of even currently

[8] A `nop` is a useless instruction that merely fills an empty issue slot.

available transformations must be restricted because of the costs of unconstrained code specialization and speculation. Stability is thus a fundamental obstacle, not only for production compiler writers, but also for necessary EPIC compiler research as it probes the limits of EPIC effectiveness in in control-intensive applications.

1.4 What Does "Optimization" Really Mean?

"Optimization" for ILP in EPIC machines is of a fundamentally different character than traditional compiler optimization. It is, however, as important to relate this work to the larger body of classical compiler optimization techniques as it is to relate it to other architectural styles of ILP exploitation. Program optimization[9] attempts to produce a set of instructions that will perform a given algorithm in a minimum amount of time using given hardware resources. Within this daunting and ill-defined task, there exist two general possibilities: either (1) reduce the amount of work (number of instructions) required or (2) increase the rate at which work is done (increase the rate at which work can be marshaled through the processor, even if it means more instructions). In today's designs, the compiler is universally responsible for the first optimization "vector" and either the compiler or microarchitecture is responsible for the second.

One might characterize traditional compilation for now-familiar out-of-order, superscalar machines (such as the DEC Alpha or Intel Pentium series) as a process concerned primarily with the first objective. Though this is in reality decreasingly the case, the substrate microarchitecture can be assumed to provide a reasonably parallel, dynamically "optimized" execution of the instruction stream as minimized by the compiler. Ideally, it maps a simple sequential model targeted by the compiler onto a pipelined and potentially superscalar machine. Two problems, both exacerbated by increasing processor clock frequency, can militate against this activity, foisting more responsibility on the compiler; to this point, they have apparently been of relatively little concern to the publishing compiler community. First, microarchitectural schedulers are forced to either do less accurate

[9] As Aho points out [49, p. 585], when compiler writers presume to say "optimization" we should hear "a code-improving transformation, we hope," as any sense of transformation optimality is almost surely artificial, and fleeting at that. Optimizations for power, code size, etc., are also possibilities; performance is the focus in this work.

or aggressive scheduling of instructions or to occupy more precious pipeline stages. Second, and even more fundamentally, as pipeline lengths increase, a greater proportion of cycles is wasted due to recovery from misprediction. (These are joined by a host of other microarchitectural penalties that are becoming more difficult and expensive to tolerate.) As a result, compiler writers for even powerful out-of-order machines are beginning to need to think about more parallel, as well as more concise, program expressions.

In the EPIC world, on the other hand, in which the processor has no ability to search on its own for parallelism among executing instructions, the onus for developing an instruction-parallel expression of input program, a plan of execution, already rests squarely on the compiler—if EPIC machines are to be competitive with out-of-order ones for control-intensive programs, the compiler needs to carry the day. While an out-of-order machine can exploit ILP across multiple, dynamically predicted branches with a flexible front end and scheduler, an EPIC machine executes groups of instructions only as laid out by the compiler, requiring it to move and sometimes replicate instructions as necessary to utilize the width of the processor. While an out-of-order machine can dynamically reorder loads, their consumers, and store instructions to keep the machine busy, the EPIC machine executes each operation in its scheduled instruction group. If all goes well, the compiler can handle these new burdens, and the unencumbered EPIC microarchitecture can concentrate on the smooth and efficient processing of many instructions per cycle.

As has been noted, there are two ways of approaching the EPIC compilation problem. One is to retain more-or-less traditional compilation strategies and to apply the EPIC features of speculation and predication to improve parallelism within essentially traditional trace scheduling models. Work like wavefront scheduling [50], relied upon primarily in today's production compilers, does essentially this. Explicit control speculation is used to move instructions opportunistically above their controlling branches, and predication is applied, also opportunistically, where decisions can be made before the branch point to reduce the speculativeness of hoisted instructions. This is a safe approach, as it perturbs the norm only a little, but it applies EPIC features only to palliate some of the

more obvious symptoms of control dependence's stranglehold on ILP. It limits both the potential for loss and potential for gain.

The more adventurous approach, the one taken in IMPACT,[10] is to examine program structure, with the aid of profiling information, and restructure the program in a somewhat systematic way. This approach focuses on creating efficient program organizations out of generally obstructive control flow rather than scheduling instructions efficiently within the existing graph. This will become clear in the examples of Chapter 3, in which large regions of code, even those containing nested loops, are selected for if-conversion (the program transformation that removes internal branches from a region of code, replacing all internal control flow with predicate assignments and evaluations). Today this allows a wide variety of complex transformations to take place—most to good effect, but some not. An empirically determined set of heuristics and a fixed phase ordering control the process. This introduces a number of deficiencies, as will be described in the next section. Still, this framework's bold transformations achieve good results in a number of SPEC CINT2000 benchmarks.

This brings us back to the topic of the meaning of optimization. The compiler is attempting to provide an efficient plan of execution for a machine, of which it has some knowledge, with only limited predictions of run-time program characteristics (control flow tendencies, variable-latency instruction durations,[11] etc.). In no clear sense is the program it is producing "optimal," although the performance its program achieves is higher than that of one produced using only the traditional target of optimization. EPIC optimization is only optimization in Aho's sense, but the glimmer of optimality within it is even weaker than in the systems of which he spoke. Design of a successful EPIC compiler is better described as a problem of meta-engineering (of producing a system to engineer workable solutions in complicated situations) rather than one of optimization. This situation, while regrettable in the sense of producing well-characterized and satisfying

[10] also to a much lesser, opportunistic extent in production compilers, which have drawn techniques from more aggressive research but "governed" them

[11] These variations can have a profound effect on performance and are difficult to manage in the compiler.

algorithms, is nonetheless the reality of EPIC compilation. Meaningfully describing, evaluating, and improving such a system is the challenge this dissertation faces.

As both the demand for and the difficulty of extracting ILP increase, we believe a more strategic and far-reaching approach—further in the direction of IMPACT's historic work—rather than an evolution of the standard techniques will be required to make EPIC machines competitive in control-intensive programs. This dissertation aims to evaluate the feasibility of this approach, both to point the way to more systematic (and therefore more stable), more comprehensive (and therefore more effective) transformations for a new generation of EPIC compilation work and to guide profitable EPIC (micro)architecture development.

2 INTRODUCTION

This dissertation describes the culmination of a five-year experiment that applied the suite of ILP-enhancing transformations embodied in the IMPACT compiler to the modern, nonnumeric applications of SPEC (Standard Performance Evaluation Corporation) CINT2000 [5] applications in the context of the Intel Itanium 2 microprocessor [51], the first full, contemporary embodiment of the EPIC processing paradigm [27]. This work is the first comprehensive application of code specialization, predication, and speculation techniques developed in a research context [38] to a production microprocessor expressly designed to target them. As such, this work closes an important feedback loop—it provides data for conclusive evaluation of compiler and architecture techniques developed over the past two decades or more, in the light of a modern implementation and modern benchmarks. The chief conclusions of this dissertation, which will be substantiated in the chapters to follow, are:

- The control flow structural (CFS) ILP-enhancing transformations of the IMPACT compiler, when taken as a whole and as adapted in the course of this dissertation work, are effective at increasing the performance of SPEC CINT2000 applications on the Intel Itanium 2 architecture. They achieve an average speedup of 1.20 (geometric mean), and up to 1.59, on the benchmark suite, relative to traditionally optimized code using the same interprocedural analysis, using the same degree of

procedure inlining, and running on the same machine.[1] Formation of Superblock regions[2] and speculation of safe operations contribute a speedup of 8% relative to classically optimized code. Control speculation of potentially-excepting instructions adds an additional 7% to the speedup. Finally, the aggressive use of predication in a Hyperblock model adds an additional 5%. These benefits are shown to be unique and compatible with each other. The effects of underlying transformations and analyses such as procedure inlining and interprocedural pointer analysis are also examined. Quantitative evaluation of these techniques in a real-machine context is in itself a major contribution.

- The achieved degrees of speedup vary widely among the benchmarks. In some cases, the speedup is much greater than the average (up to 1.59 in the case of *vortex*). In other cases, the speedup is insignificant due to unaddressed performance factors (e.g., in *mcf*). No benchmark suffered an overall decrease in performance due to ILP-related transformations. This indicates a usable degree of stability in these transformations and allows characterization of the problems the techniques are presently inadequate to address. Function-level results further support this assertion of practical stability, although in individual cases performance is sometimes degraded.

  Even within the relatively narrow scope of applications selected for inclusion in SPEC CINT2000, programs exhibit a wide variety of reactions to CFS techniques. Some benchmarks, e.g., *twolf*, responded well to the use of Hyperblocks (predication); others, e.g., *vortex*, achieved approximately the same speedup with Superblock formation alone. These variations are due primarily to differences in algorithm features, coding styles, run-time patterns of control flow, and the degree or pattern of profile variation. These distinctions have not always been made clear in the literature, leading to difficulty in interpreting and comparing results.[3] The

[1] It is necessary to speak of "same machine" since in the baseline case, the machine suffers the cost of predication (6 bits per instruction) without reaping any of its benefits.

[2] with ancillary transformations, such as branch target expansion

[3] *cf.* [38], [39] and [52], which convey vastly different impressions of the value of ILP region formation techniques.

demonstration of a consistent application of ILP techniques that provides reasonable performance across this suite, and documentation of the principal variations therein, are important contributions.

In some cases, however, e.g., *crafty*, complex interplay among several techniques contributed to lackluster performance. This dashed hopes of finding effective, unified techniques for ILP optimization. Literature today is lacking definitive expositions of the important principles at work in compilation of these programs for instruction-level parallelism, impeding the application of important, aggressive techniques in production environments. The dissertation discusses these issues in detail, providing guidance for compiler developers investigating the future use of these techniques.

- Achieving the breadth and stability of results delivered required substantial improvement of various aspects of the compilation process. One of the most prominent examples of this was the extensive modification to region formation necessary to avoid an unacceptable degree of code expansion. Without these improvements, techniques previously described in the literature and designed for simpler, more regular programs either failed or were unable to best address the larger, more complex, and less regular programs of SPEC CINT2000. After reasonable controls were applied, only *crafty* and *twolf* exhibited substantial degrees of controllable instruction cache impact, and these resisted improvement within the constraints of the experimental framework. Experimental results thus suggest the insufficiency of traditional control mechanisms to adequately control the interaction of the ILP-enhancing techniques in a limited number of cases. In particular, the independent treatment of (even inlined) procedures after a brief, high-level, interprocedural optimization stage has the potential to provide inadequate management of instruction cache resources in the presence of code-replicating transformations.

- Code expansion is the primary cost of the specialization applied in the CFS approach to ILP enhancement. Experiments indicate that, for the benchmarks of SPEC CINT2000, this expansion does not result in unmanageable pressure on

instruction-delivery mechanisms, and thus does not unduly detract from performance, even with aggressive levels of transformation. In fact, instruction cache performance is improved in the typical case (always fewer accesses, and sometimes an increased hit rate). This validates the aggressive CFS optimization approach of the IMPACT compiler, at least for applications structured like the components of SPEC CINT2000, and suggests it may in fact be a useful alternative to the more conservative approaches employed in contemporary production compilers.

- The speedup achieved by IMPACT's techniques is heavily diluted (in the Amdahl's Law sense) by the prominence of stalls due to the latency of loads from secondary (and, in some cases, tertiary) levels of cache, a factor that was generally dismissed as insignificant or as a readily solvable factor in earlier studies [4, 53]. Despite limited successes using prefetching [54] (as performed in Intel's compiler), attempts to manage this latency in the compiler have, in general, proved fruitless. Alternatives are addressed in Chapter 11. The interaction of this problem with CFS transformation is described—by increasing both the number and the density of accesses to memory, CFS can aggravate these already-significant problems.

- Phase ordering, particularly as relates to inlining and subsequent region formation, posed continuing problems. The IMPACT compiler cultivates ILP across virtually the entire compilation process. Inlining and region formation, for example, both replicate code in the search of ILP; these compete for instruction cache resources, but this competition is not effectively managed by the compiler—an example of the "strategic versus tactical" problem described in the previous chapter. This interaction is difficult to control to "optimality," but current management strategies are shown to work well in practice. Deleterious effects of code expansion were not significant enough to allow for serious study of this problem; a different application set would be required.

- Sophisticated analysis is necessary to support the aggressive use of EPIC features, particularly the combination of predication and control speculation. A new method of predicated data flow analysis was developed that builds on the strengths of the

Predicate Analysis System [55] to deliver faster and more accurate results, enabling performance improvement and compile time reduction in procedures making heavy use of predication.

- The general speculation (no recovery code) model for control speculation, as has been employed throughout the development of EPIC [25, 56], but was supplanted in Itanium by the more rigorous but also more costly recovery code Sentinel model [52, 57], is practical and efficient, when the mechanisms described in this dissertation for its control are employed, and frequently dangerous to performance when they are not.

- IMPACT assumes control flow profile data. Even within SPEC CINT2000, substantial profile variation (affecting final code performance) was observed, in particular in *crafty* and *perlbmk*. This dissertation further characterizes this problem, but proffers no new solution.

Taken together, these conclusions present a solid evaluation of the state of the research art in EPIC compilation. They demonstrate that the various techniques tested in the literature, using sketchy simulation environments, indeed work together to deliver performance in an actual machine context. In many ways, though, this dissertation describes the end of an era. As examples throughout the document, and especially Chapter 11, will show, it appears that radically different techniques will be required to further increase the applicability of EPIC technologies to irregular, nonnumeric programs.

2.1 Introducing the Methodology

The goal of this dissertation's experimental system is to evaluate the ability of a compiler under development (the IMPACT Research Compiler for Itanium) to extract high levels of performance from a particular set of control-intensive, nonnumeric benchmarks (the SPEC CINT2000 suite) on a particular instantiation of the EPIC design paradigm (the Intel Itanium 2 microprocessor). The remainder of this introductory chapter provides the necessary overview of these elements and introduces results that will motivate

the discussion of the remainder of the dissertation. For this chapter, a brief overview of the salient features of each needs to suffice; details relevant to particular aspects of the experimental results are left to other chapters and to the appendices.

2.1.1 The Itanium 2 microprocessor and experimental configuration

This dissertation concerns itself with achieving performance on one particular EPIC model, that of the Itanium architecture [58], as embodied in the Itanium 2 processor microarchitecture [41]. The Itanium architecture provides the core EPIC features: control speculation (using either the general speculation or Sentinel/recovery code schema); full predication (64 predicate registers); large integer and floating-point register files; and support for kernel-only modulo scheduling [57]. This design incorporates all the critical features of the earlier HPL-PD [4] and IMPACT EPIC [38] research architectures, and thus provides a perfect opportunity to evaluate the outcomes of these research lines in a completed design.

One of the interesting features of the Itanium architecture is the means used by the compiler to express issue groups. Itanium provides a compressed VLIW encoding with unit-assumed-latency (UAL) semantics [4].[4] UAL semantics eliminate the need for vertical `nop` instructions, used in some VLIW machines to encode the latency of instructions by filling empty issue cycles; in Itanium, score-boarding ensures the machine is stalled on nonunit latencies. To reduce the number of horizontal `nop` instructions required in the encoding (those required to position other operations in the correct slots), Itanium introduces *bundles* and *stop bits*. A bundle is a group of three instructions (two, if one is long format). The bundle specifies the instruction type (memory, instruction, floating-point, branch) for each enclosed instruction. Only certain combinations are available, simplifying the machine's issue logic. The bundle also specifies the location of stop bits relative to the three enclosed instructions. A stop bit indicates the end of an issue group. A bundle may always specify a stop bit after the bundle's instructions; some bundles may specify an internal stop bit, as well [57]. Even with all these "compression" features, however,

[4] UAL refers to the presence of interlocking hardware.

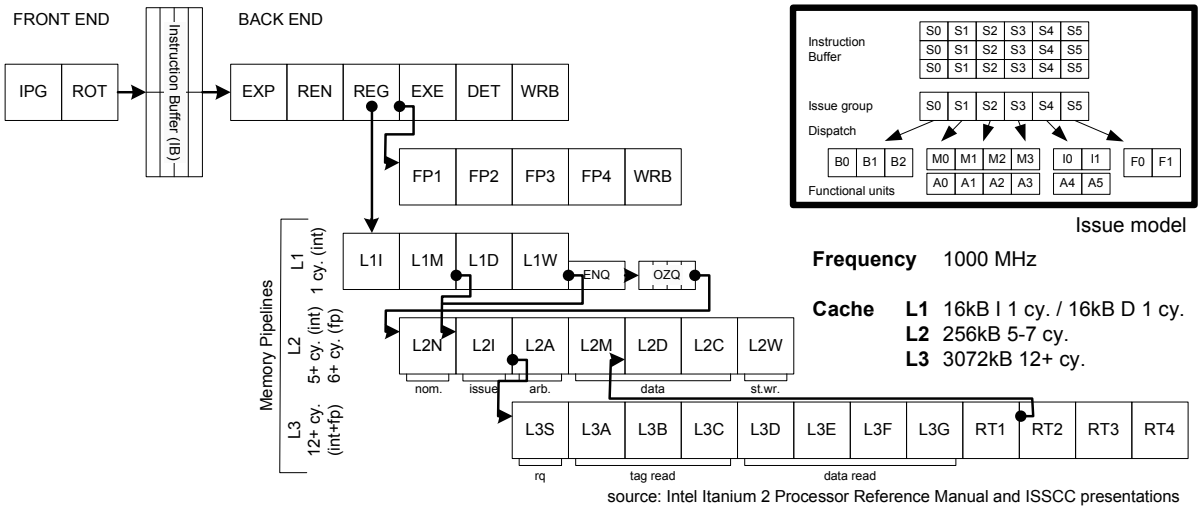


Figure 2.1 Intel Itanium 2 pipeline.

explicit `nop` operations are sometimes required. In classically optimized code, one out of four instructions is typically a `nop`. In aggressively optimized code, `nop`s may be as infrequent as every 10th instruction. The increase in instruction packing efficiency, that is, the reduction in cache storage and fetch bandwidth used for useless `nop` instructions, that comes with higher levels of ILP scheduling freedom is an important component of performance improvement for the techniques described here.

The Itanium 2 microarchitecture implements the Itanium model in an 8-stage, in-order pipeline, as shown in Figure 2.1. The experiments described in this dissertation were all performed on an Itanium 2 processor, clocked at 1.0 GHz. This processor supports the issue of up to six instructions per cycle[5] on eleven issue ports (which, in turn, feed six arithmetic/logic units, two special-purpose integer units, two load ports, two store ports, two floating-point units, and three branch units). The machine performs no register renaming (the REN stage of the pipeline has to do with rotating registers for modulo scheduled code, not conventional register renaming). As indicated by the buffer shown in Figure 2.1, instruction fetch and alignment are decoupled from the processor back end by a small buffer (48 operations) to allow limited fetching ahead during back end stalls [51].

[5]In this dissertation, the term *instruction* refers to a single operation, and *issue group* refers to the group of instructions arranged by the compiler to issue in a single cycle. This nomenclature differs from that used in earlier VLIW work.

The Itanium 2 used in this work has a three-level cache hierarchy, with a 16 KB L1I cache (1 cycle latency), 16 KB L1D cache (1 cycle), a unified 256 KB L2 cache (5 cycles) and a 3 MB L3 cache (12 cycles). A miss in the L3 cache has a minimum latency of 110 cycles in a 1 GHz processor. The actual latency of memory operations into the L2 and lower levels varies substantially, depending on other ongoing activities in the memory queue [51].

The experimental system consisted of a Hewlett-Packard zx6000 workstation with two such Itanium 2 processors (all experiments are, however, single-threaded) and a memory subsystem populated with 8 GB of DDR PC2100 memory (two 1 GB DIMMs per each of four out of the six available banks). The system runs Red Hat Linux Advanced Workstation release 2.1AW (Derry) with a specially modified Linux kernel of version 2.4.21 (modified to support general speculation). This implies an LP64 storage model (long integers and pointers require 64 bits, or 8 bytes, of storage).[6]

Further details regarding the Itanium architecture and Itanium 2 microarchitecture are provided in Appendix A.

2.1.2 The IMPACT Research Compiler

This dissertation's experiments characterize the effects of aggressive ILP transformations on SPEC CINT2000 programs running on the just-described Itanium 2 microprocessor. The compiler is the critical part of the system under test—the experiments presented here are intended to probe the effectiveness of the ILP exploitation techniques, some old and some created especially for this work, in the IMPACT compiler. The IMPACT compiler is an interesting tool to examine because it affords a greater degree of flexibility and more aggressive utilization of EPIC features than are available in commercial production compilers or the popular Gnu GCC.

Retargeting the IMPACT compiler, which for several years had focused on an abstract, parameterizable EPIC simulation architecture, to produce competitive code for

[6] It is important to note this distinction when comparing this dissertation's results to other published SPEC marks. HP-UX Itanium systems and code compiled, even for Linux, using Intel's compiler (with the `-auto_ilp32` flag), may use the 32-bit model, effectively doubling the pointer and long integer capacity of the cache hierarchy.

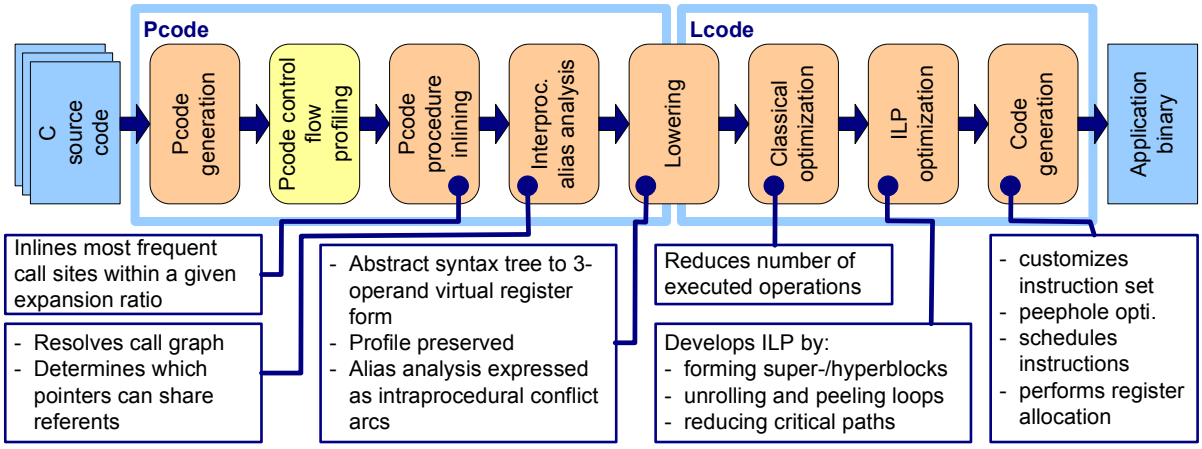


Figure 2.2 The IMPACT compiler: high-level phase ordering.

Itanium 2 proved to be a daunting challenge. This work involved adding 64-bit support to the previously 32-bit compiler, modernizing and generalizing an experimental Itanium back end written by Intel engineers, and modifying and extending previously developed, experimental ILP compilation techniques for success on a real hardware platform. The use of the IMPACT compiler and the time-frame of this conversion forced certain decisions (the compiler does not, for example, support Sentinel speculation with recovery code, so general speculation had to be enabled on the experimental system).[7] Today's IMPACT compiler, after this thoroughgoing revision, is generally competitive in compiled code performance (but certainly not in compile time—it is a *research compiler*, after all) with leading commercial compilers for Itanium 2. This demonstrates the effectiveness of the compiler's transformations and establishes a "validated" baseline for these studies. (When a compiler is not operating at a performance level competitive with other compilers, self-comparisons with varied parameters could reveal performance differentials that are irrelevant to other frameworks.) Throughout all this work, it is important to add, care was taken to ensure that IMPACT remained a flexible research compiler that could support the breadth of the experiments presented here (as well as the work of others).

An introductory description of the IMPACT compiler's structure is in order before proceeding with the detailed presentation. Figure 2.2 shows the principal phases of the

[7]Sentinel and general speculation are models of control speculation for potentially-excepting instructions. See Chapter 6.

compiler. The compiler cultivates instruction-level parallelism through successive application of various techniques, most of which operate at less than full-program, or even less than full-function, scope. The controlling philosophy is that each phase must be sufficiently aggressive to obliterate program barriers, such as control dependence, so far as possible, within interesting code regions, creating opportunities for later phases, but not so over-aggressive that it causes the accumulation of insurmountable secondary penalties. For example, function inlining takes place well before the compiler has the ability to gauge its instruction cache effects, so its aggressiveness must be constrained; it is, however, relied upon to expose regions to subsequent region formation and speculation techniques, and must therefore take some risks. Hyperblock and Superblock formation are similarly aggressive, operating before it is understood what benefits their specializations will purchase. Most of the work to be discussed here occurs in two program restructuring phases, which are most directly responsible for exposing instruction-level parallelism. The first is the "Pcode procedure inlining" phase, which selects frequently traversed call sites for inlining, allowing commingling of caller and callee code and elimination of optimization-obstructing subroutine calls. The second, "ILP optimization," performs code replication and forms Superblocks and Hyperblocks, the potentially predicated code regions devoid of internal control flow, as the basis for cultivation of parallel issue through optimization and scheduling. The interesting features of the rest of the compiler are described as they become relevant, or in Appendix B. The compiler used in this dissertation research is available at the Gelato / OpenIMPACT world wide web site [43].

The IMPACT compiler offers a highly configurable compilation path with dozens of parameter switches and knobs. For the purposes of this dissertation, five basic configurations will be used consistently as points of comparison. As the need arises, deviations from these five models will be noted in the text. Table 2.1 explains configurations. Generally, the first letter of the configuration's moniker indicates the structural approach (**O** means classical control flow optimization only; **S** indicates Superblock or single-path optimization; **I** indicates full ILP, including predication) and the latter part indicates whether

Table 2.1 Basic configurations of the IMPACT Research Compiler

included in all configurations	Profile-guided (using SPEC training inputs), cross-file procedure inlining is performed, up to a 2.0× touched-code expansion limit. (See Chapter 7.) Control flow profile information is annotated for subsequent use. FULCRA interprocedural pointer analysis (a field- and context-sensitive, Andersen's formulation with heap cloning) [59] and Omega test [60] are applied. Profile-guided optimization is used (with the SPEC CINT2000 training inputs).
O-NS	Classical optimizations [49, 61] are performed (copy propagation, common-subexpression elimination, partial redundancy and partial dead code elimination, register promotion, etc.). Local, acyclic instruction scheduling is performed, but without control speculation. Basic blocks are laid out according to available profile information, but no trace formation, tail duplication, loop peeling, or loop unrolling are performed. Predication is used only incidentally in performing optimizations, but no if-conversion is performed.
I-NS	In addition to the optimizations of the **O-NS** configuration, ILP-cultivating transformations are applied. Trace formation (Chapter 4), tail duplication, if-conversion (Chapter 5), loop peeling, and loop unrolling, as well as ancillary transformations such as expression height reduction, are performed. Modulo scheduling is applied where appropriate. Still, no control speculation is performed.
I-CS	All the optimizations performed in the **I-NS** configuration are performed, but control speculation (Chapter 6) is added (both in terms of scheduling across branches and of predicate promotion). This is the "peak" or "best" configuration of the IMPACT compiler presented here.
S-NS	This configuration is identical to **I-NS**, except that the use of if-conversion (and with it, loop peeling) is eliminated. Because if-conversion is not applied, this is a Superblock-based framework. This configuration is presented to help differentiate between the benefits of region formation in general and if-conversion in particular.
S-CS	All the optimizations performed in the **S-NS** configuration are performed, but control speculation (Chapter 6) is added (in terms of scheduling across branches). A comparison of **S-CS** to **I-CS** indicates the benefit of predication in the presence of control speculation.

(**CS**) or not (**NS**) control speculation of potentially-excepting instructions is enabled.[8] Benchmark and function-level performance and cycle accounting differences among these models are used to point out the benefits and costs of the various techniques applied.

2.1.3 Benchmarks and benchmarking

This dissertation uses the C and C++ programs of the SPEC CINT2000 benchmark suite [5, 62] in its experiments. These benchmarks can be classified as *compute-intensive*, tending to emphasize CPU performance (including, to a minor extent, the performance of nearby cache structures) rather than other system components; *control-intensive*, characterized by frequent, relatively difficult-to-predict branches; *integer* (or *nonnumeric*), not dominated by either long-latency floating-point operations or by easy-to-parallelize, large, regular numerical computations. Within these general parameters, each of the 12 component benchmarks offers a unique sample point for measurement of optimization effectiveness. The benchmarks respond in individual ways to EPIC optimization efforts—each can be seen as representative of a class of typical programs. The intention here is not to provide simply a tuning guide for SPEC CINT2000 on Itanium 2; rather the SPEC CINT2000 suite is used to show concrete examples of certain general problems in EPIC compilation, as an accepted means of evaluating the fitness of the described techniques in a nonnumerical code environment.

Given the complex nature of ILP transformation, it is impossible to characterize a compiler's effectiveness for all programs in general, and difficult to understand even concrete compiler-derived speedups, without detailed knowledge of both the techniques applied and the programs manipulated. This detailed evaluation of the IMPACT compiler's effectiveness in optimizing well-known, readily available benchmarks on an easily characterizable, readily available system is intended to allow the reader to assess concretely the results of this dissertation.

[8] Even in the **NS** configurations, those instructions that are not potentially excepting or are provably safe (such as loads to known-valid addresses) may be scheduled (and promoted, in the case of predicated instructions) without regard to their control dependence. Without allowing this, the benefit of trace formation is minimal. See Section 6.1.

This dissertation relied primarily on *in situ* performance assessment—the use of complete runs of reference inputs for full-sized benchmarks, on real systems. Except for estimated SPEC CINT2000 ratios, obtained using the provided runtime measurement scripts, all performance results presented here were obtained using the Itanium 2's hardware performance monitoring features as supported by *Perfmon* kernel support and the *Pfmon* interface developed by Hewlett-Packard Laboratories [63]. This combination allows over 400 events to be either counted or sampled (correlating events to instruction or data addresses), up to four at a time, in a low-overhead, per-process measurement environment. These tools were extended by the author to provide the high-resolution, low-overhead sampling features required to provide measurements of event frequencies localized to particular regions of code. Benchmark-level numerical results are based on event counting, not sampling. Useful event counts are available and reliable on Itanium 2, much more so than on out-of-order processors, which have to deal with much more complex execution state. Function-level results are based on a sampling approach that, while not guaranteed to be totally accurate, has served well in practice. This information has proven invaluable, not only in demonstrating performance benefits, but also in localizing performance problems.

In addition to providing results for discussion, these tools have proven invaluable in the compiler tuning process. When combined with some analysis skill they provide a reasonable level of understanding across an entire program execution without the substantial overhead of simulation. The tools can both identify the magnitude of perturbation by various microarchitectural sources and localize the causes of these problems. Chapter 9 describes further details of the measurement tools and procedures used in this effort.

One of the underlying themes of this dissertation will be the analysis of the suitability of the benchmarks themselves for EPIC exploitation. In this dissertation's prolegomena (Chapter 1) it was observed that instruction-level parallelism was a convenient means of extracting parallelism from typical, sequentially programmed code. Most programs are written sequentially, and it is arguably much more natural for programmers to express most programs in this manner [4]. Programming for parallel systems often assumes a

particular target system, with assumptions being made about the degree of parallel execution to be achieved, the cost of communicating among the nodes, etc. It has generally been assumed that code written for uniprocessor execution is more performance-portable. This is true to some extent for out-of-order machines, which can rearrange instructions locally and at execution time (having perfect information) for more parallel execution. For an EPIC compiler/processor system, however, it is less clear that there is a good match between typical program construction and efficient optimization and execution.

Schlansker and Rau opine in a 1999 HP Labs Technical Report:

> "Continuing this trend, of ever-increasing levels of performance without re-writing the applications, is the topic of discussion of this report. ... ILP systems are given a conventional, high-level programming language program written for sequential processors and use compiler technology and hardware to automatically exploit program parallelism. Thus an important feature of these techniques is ... they are largely transparent to application programmers." [27]

While the author agrees with Schlansker and Rau that a compiler can expose some degree of parallelism from programs without their being rewritten, programming practice and program structure in many cases, even in SPEC CINT2000, militate against the most effective patterns of ILP development. This will be indicated especially in Chapter 5, dealing with predication, and Chapter 7, dealing with inlining. The application of these two EPIC compilation stand-bys often conflicts with program idioms selected by developers. Programs written with the possibility of EPIC optimization in mind stand a better chance of successful transformation. In reality, this situation may be approaching the complexity of writing some parallellizable programs.

2.2 Limitations of Scope

This dissertation is intended to deal primarily with the problems of extracting ILP from nonnumeric (SPEC CINT2000 and similar) programs on EPIC architectures like

Intel's Itanium. Numerical applications have unique problems, such as loop transformation for cache blocking, which are outside the scope of this work. Likewise, to limit the scope of the problem, certain orthogonal techniques such as program augmentation for data cache prefetching, though profitable for some nonnumeric applications, will be left unaddressed. Data cache latency, for example, a critical element of performance on Itanium, receives a regrettably brief treatment in Chapter 8.

Preferring to explore general limits and interactions, rather than necessarily fast and cheap algorithms, the dissertation will not concern itself with compilation time and space constraints. IMPACT is notoriously slow, taking more than a day to compile SPEC CINT2000 on a single machine at the highest level of optimization. Much of this time is due to optimizations being written in a simple, easily extensible fashion.

Itanium 2 performance in control-intensive, general-purpose programs depends heavily on the compiler's ILP-exposing transformations, which IA-64 supports with predication, explicit control speculation, data speculation, and modulo scheduling aids [57]. The IMPACT compiler currently makes use of all these features except for data speculation. Although IMPACT's aggressive pointer analysis reduces the benefit IMPACT-compiled code could derive from data speculation (perhaps in contrast to production compilers), many potentially fruitful opportunities for data speculation can be observed. In *gap*, for example, pointer analysis is unable to resolve critical spurious dependences in otherwise highly parallel loops. A limited initial application developed by Ian Steiner provided a 5% speedup in *gap*; much more is attainable. Aside from mitigating deficiencies in alias analysis, data speculation can also allow the compiler to manage even "known-sometimes" dependences. Other researchers have shown opportunities to exist for profitable integration of data speculation into optimizations [64].

Since recovery code generation is not available in IMPACT at this time, the general speculation model, as described in Chapter 6, or briefly as the immediate execution of control-speculative, potentially-excepting instructions, with the ability to ignore program-terminal exceptions, is used to implement control speculation. As that chapter will explain, general speculation is a viable and sensible choice of control speculation schema, being potentially more efficient and easier to implement than the alternatives.

Briefly, in the general speculation model, the operating system is instructed to ignore any program-terminal exception on instructions marked control-speculative. Under this model, speculative instructions are executed fully and immediately, rather than deferring prosecution of faults until the instruction's original execution condition is satisfied (as in the recovery code schema). Since the recovery code schema is currently the more popular in production environments, however, all efforts have been made to ensure that either the conclusions drawn here are applicable to the other model or that the differences can be characterized in a useful way.

The dissertation will deal exclusively with code in the presence of profile data, as provided for under SPEC CINT2000 rules. The IMPACT compiler depends on profile in several ways: (1) in procedure inlining; (2) in optimization, particularly block layout, partial redundancy elimination and loop optimizations; (3) in Superblock and Hyperblock region formation [22, 36]; and (4) in control-speculative acyclic and modulo scheduling [65]. In the first, call graph profile information (including indirect procedure call edges) is required. For the latter three, control flow weight and loop (iterations-per-invocation histogram) are obtained. This profile data is today crucial to effective performance for EPIC machines. Practically speaking, representative profiles are often difficult to obtain.[9] Transformations also corrupt control flow information throughout the compilation. However, we anticipate that as the techniques proposed will improve the compiler's general stability, the effects of input variation will actually be reduced as a side-benefit of this work. Additionally, researchers have proposed profile-gathering [66] and optimization [67] models that support profile-based techniques in a more generally applicable manner than that available today. As the opportunity presents itself, the effect of profile variation within the benchmarks is explored, yielding insights for the proper use of bias in the application of ILP techniques. We revisit these issues in more detail in Chapter 7.

[9] Also, SPEC CINT2005, the next generation of integer benchmarks, is understood to forbid control flow profiling, at least for the base configuration.

2.3 High-Level Results

This dissertation work attempted to be driven as much as possible by examination of empirical data from the studied benchmark set, running on the target machine. This led to various interesting investigations in the course of the work that were not encountered in the abstracted simulation approach typical of previous EPIC work. A representative example of the abstracted approach is the critical path reduction (CPR) work of Schlansker, et al. When performance results for these techniques were published in 1999, they were based on compiler estimations of schedule height, weighted by the control flow profile [68]. They thus included no dynamic effects whatsoever. This made for a clean experimental environment, but makes the application to real hardware questionable. The same could be said of [38]. This being the case, it is appropriate to begin with a summary of the performance results achieved by the techniques to be described in the following chapters.

2.3.1 Estimated SPEC CINT2000 performance ratios

Table 2.2 shows estimated SPEC CINT2000 performance ratios[10] (higher is better) for GNU and IMPACT compilers on the system described in Section 2.1.1. The first numerical column indicates the score for Gnu *GCC* version 3.2, a commonly used, free, easily retargetable compiler.[11] The second indicates scores for Intel *icc* version 8.1.[12] The remaining columns show scores for codes compiled with the various configurations of the IMPACT compiler shown in Table 2.1.

The classically optimized **O-NS** baseline already shows a commanding lead (a 42% increase in performance) over GCC, without applying the ILP exploitation techniques

[10] A SPEC ratio, or score, for a given benchmark is computed as $100t_r/t_s$, where t_r is the reference execution time and t_s is the measured execution time of the system under test. The suite-level score is the geometric mean of the individual benchmark scores [5].

[11] Command: `gcc -O3 -fomit-frame-pointer`. *GCC* is run without profile feedback, as it is not yet generally supported.

[12] Command: `icc -O2 -ipo -static -prof_gen|prof_use`. Scores for `-O3` are indicated in Appendix C. `-O3` enables data prefetching and high-level loop optimizations, orthogonal to CFS techniques and not available in the IMPACT compiler, so `-O2` provides a better comparison point.

Table 2.2 Estimated SPEC CINT2000 performance ratios

Benchmark	**GCC**	**icc**	**O-NS**	**S-NS**	**S-CS**	**I-NS**	**I-CS**
gzip	374	629	588	624	696	657	751
vpr	497	666	616	659	719	692	756
gcc	521	988	833	955	1066	977	1030
mcf	333	335	330	335	323	331	338
crafty	489	801	657	695	709	715	745
parser	410	567	532	549	560	546	559
eon	273	895	462	494	530	578	611
perlbmk	472	676	733	728	775	738	772
gap	375	601	569	607	651	597	630
vortex	550	1080	867	1220	1393	1193	1382
bzip2	414	662	610	624	654	737	763
twolf	557	798	738	755	816	812	884
GEOMEAN	430	697	609	656	699	682	730

enabled by Itanium's EPIC architecture.[13] The reasons for this are several: even in this configuration, IMPACT performs aggressive cross-file inlining, interprocedural pointer analysis, careful instruction scheduling, and significant peephole optimizations for the Itanium platform. While GCC performs a very competent level of traditional optimizations, it is not equipped to deliver even minimal levels of ILP on Itanium 2. GCC lacks interprocedural optimization entirely and, because its developers choose to focus on language features and retargetability over single-machine performance, has little ILP cultivation or peephole optimization for Itanium. IMPACT's success here reflects careful tuning and optimization of the baseline configuration to ensure, first, that subsequent stages will have the best possible quality initial code to work with, and, just as importantly, that the results of ILP transformation will not inadvertently be overstated. The importance of performing compiler work such as this within a "validated" performance domain, one comparable with other compilers in use, cannot be overstated, since it is very easy to misattribute performance gains without a solid frame of reference.

[13]For a comparison of IMPACT to Intel's commercial compiler on the Itanium platform [35], see Appendix C.

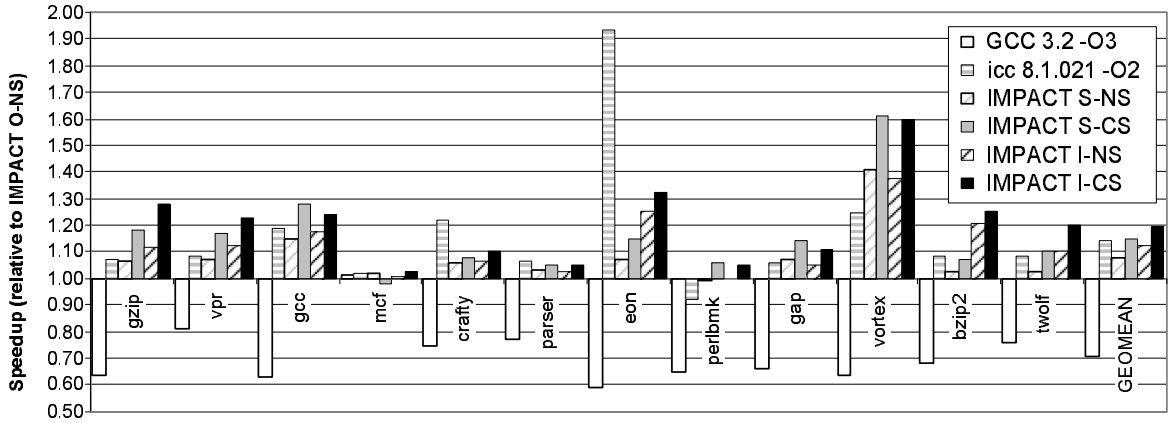


Figure 2.3 Speedup relative to **O-NS** baseline.

Returning to the results of Table 2.2 and adding Figure 2.3 to indicate the relative performance among the various configurations, **I-NS** applies predication and ILP-formation techniques, but not support for control speculation of potentially-excepting instructions, achieving an average speedup of 1.12 relative to the **O-NS** baseline. **I-CS**, finally, adds control speculation, achieving a cumulative average speedup of 1.20 (maximum 1.59) relative to the **O-NS** baseline, and 1.70 on average (maximum 2.51) relative to GCC.[14]

To allow evaluation of if-conversion as a component of the CFS strategy, two intermediate columns, **S-NS** and **S-CS**, are provided. These columns reflect a configuration of IMPACT that does not perform if-conversion (S stands for the use of the Superblock, as opposed to Hyperblock, framework). The **S-CS** configuration achieves, on average, a speedup of 1.14 (maximum 1.61) relative to baseline code.

Finally, the **I-CS** configuration is the "peak" configuration for IMPACT, the one proposed as indicating the current pinnacle of IMPACT's performance. (**I-NS** reflects the same configuration, without control speculation of potentially-excepting instructions.) The **I-CS** configuration achieves a 1.20 average speedup relative to **O-NS** code and a

[14]These results are generated in real SPEC evaluation runs, on real hardware, in the spirit of SPEC's run rules (training/reference inputs, compilation setting consistency, etc.) but are "experimental" in nature. In keeping with SPEC's policy on research use, we therefore label our results "estimated." These results reflect 64-bit pointers, in contrast to most published Itanium 2 results. See Table 2.1 and Appendix B for details on IMPACT's configurations.

1.70 average speedup relative to *GCC*. It also achieves substantial gains (frequently 10-20%) relative to *icc*, the best commercial compiler on the Itanium linux platform (when the latter is configured as described previously).

The high-level performance results indicate that IMPACT's optimizations are meeting with varying degrees of success in the different benchmarks. Benchmark results may be categorized in several ways. First, in *mcf*, ILP optimization appears generally unfruitful, yielding practically no benefit. The benchmarks *parser* and *perlbmk* are not much better off, each garnering about a 5% improvement from transformation. There are several benchmarks for which ILP optimization is successful, but for which **I-CS** and **S-CS** configurations have approximately the same performance (or for which **S-CS** even outpaces **I-CS**). These include *gcc*, *gap*, and *vortex*. Finally, there are those benchmarks for which an **I-CS** configuration (with if-conversion) provides superior performance: *gzip*, *vpr*, *crafty*, *eon*, *perlbmk*, *bzip2*, and *twolf*.

While the value of predication varied from benchmark to benchmark, sometimes becoming quite substantial, when we compare speculative (**CS**) to nonspeculative (**NS**) configurations, we find that control speculation relatively uniformly magnifies the effect of CFS transformation. The interactivity and, to some degree, orthogonality, of these techniques deserve to be pointed out. For benchmarks such as *gzip*, *bzip2*, and *twolf*, the greatest benefits require both predication and speculation (available only in the **I-CS** configuration). Predication and speculation are not interchangeable means to the same end. Neither universally achieves the greater benefit. The diverse nonnumeric applications of SPEC CINT2000 provide a variety of case studies for the various EPIC techniques (and their combinations). This result, while perhaps not surprising in light of previous studies such as [38], underscores the intent of EPIC developers in including these features in combination—none a "silver bullet" solution, they together and in a piecemeal fashion extend the benefits of VLIW execution to various new classes of applications.

Aside from categorizing benchmarks with different properties, there is little more to be learned from the speedup numbers. We will, however, return to explore the reasons behind these results later in the presentation. Let us therefore delve deeper.

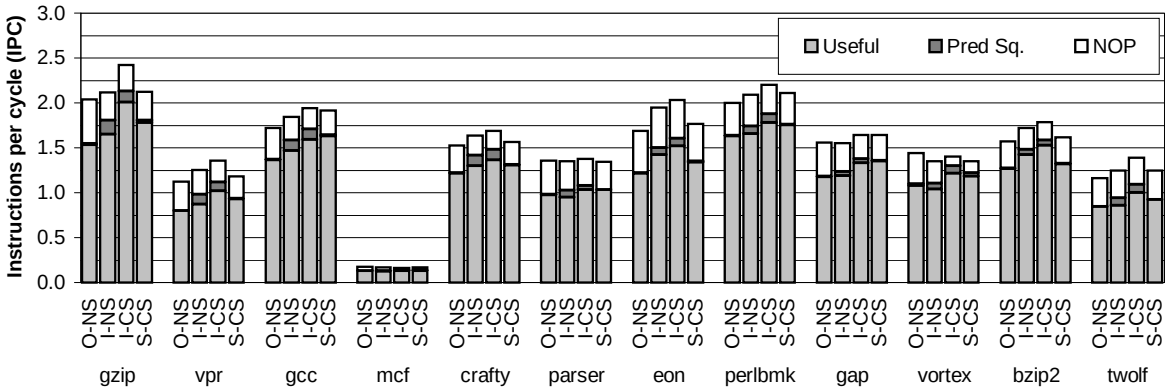


Figure 2.4 Instructions per cycle.

2.3.2 Results in instructions per cycle

The most commonly used metric for effectiveness of software and hardware techniques for ILP exploitation is the ratio of instructions executed to the number of cycles required to execute them, or "instructions per cycle" (IPC). The preference for this measure corresponds more to the dynamic ILP extraction techniques than to those applied in the compiler. In the context of hardware (microarchitectural) optimizations, the number of instructions to be executed is constant. This is not the case for a view that includes the compiler (and we are about to explore the extent to which this is true for these benchmarks). Nonetheless, IPC results are often expected and are useful for explaining some phenomena, as we shall see.

Figure 2.4 shows the IPC results for the various benchmarks across the four compiler configurations. The segments of each bar represent the numbers of particular classes of instruction instances executed per cycle. The bottom segment of the bar reflects non-`nop` instructions that executed with their predicates set to 1. This category is referred to as *useful* IPC. It should be noted that *useful* is a slight misnomer, as off-path, speculative instruction instances are included in this count. In general (and particularly in hardware measurement) there is no good way to distinguish between on-path and off-path speculative instructions. The next segment reflects non-`nop` instructions that were issued but kept from execution because their predicates evaluated to 0. Finally, the top segment indicates `nop` instructions, which serve no purpose but to fill empty issue slots in bundles. Aside from *mcf*, which performs abysmally, IMPACT/Itanium achieves a useful

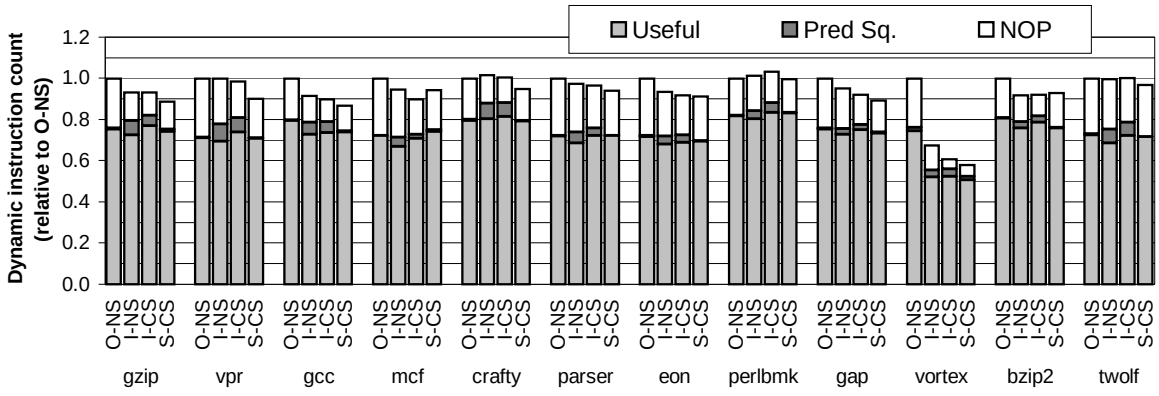


Figure 2.5 Effect of compiler configuration on dynamic instruction count.

instruction execution rate greater than 1 IPC across the benchmark suite, and in some cases up to 2 IPC. The number of predicate-squashed operations issued per cycle is small, and the number of `nop` instructions issued is noticeably reduced in the CFS-optimized configurations. These are all satisfying results, although one might hope to achieve more than 1-2 IPC in a nominally 6-issue machine (more is to be said on this later).

IPC can be a misleading metric for understanding the total performance effects of ILP compiler transformations. The same path specialization (code replication) that allows ILP transformations can also enable optimizations that can have a profound effect on the number of instructions executed. Figure 2.5 shows the number of dynamic instructions executed in each benchmark for each compiler configuration. IPC can mislead in two directions. First, it can understate the benefit of transformations, since specialization can allow greater optimization opportunity, reducing the number of instructions executed. This is the case with *vortex* under **S-CS** optimization, in which the number of useful instructions executed is decreased by 32% and IPC is increased by a factor of 10%, compounding to the observed 62% increase in performance. IPC can also suggest more benefit than is actually being derived, since ILP transformations, and control speculation in particular, can uselessly increase the number of operations executed if compile-time predictions about program bias were wrong. This is the case in *perlbmk*,[15] in which the number of useful (hence the misnomer) operations is increased by 2% while

[15]This benchmark is one of the more egregious cases of profile variation in SPEC CINT2000. Rampant misspeculation and poor spill code placement explain much of this bloat.

IPC is increased by 8%, working out a low 6% benefit for ILP techniques (in the **I-CS** configuration—when speculation is disabled, the number of instructions issued is reduced, but the performance result is also degraded slightly).

One of the unique points of this work is the use of real hardware measurement. This approach has certain disadvantages, as the observation and especially correlation of events in the hardware (not to mention, variation of characteristics such as usable cache size) is much more difficult than in a software-based simulation environment. The insufficiency of IPC as a metric for the effectiveness of compiler-based ILP techniques, though, demonstrates a decided advantage of real-hardware measurement. Real hardware execution is orders of magnitude faster than detailed microarchitectural simulation, allowing the execution of entire benchmarks *with their full reference data sets*, something almost unthinkable on a simulator. The sampling solutions typical for accelerating simulation work to a usable point have very dubious meaning when the code being sampled exists in and is being evaluated across different compiled versions. The inability of these IPC results to provide reliable performance predictions is thus another validation of the decision, however painful, to pursue this work on a real hardware platform.

2.3.3 Results from cycle accounting

To conclude the introductory examination of the high-level results, we consider the accounting of execution cycles to particular machine activities. This accounting is supported by the performance monitoring infrastructure of the Itanium 2, and is reliable because the machine is in-order. This allows a clear definition of what specific activity is preventing forward progress at any given time. Figure 2.6 shows these results. The total height of each bar is the execution time of the indicated configuration, relative to the baseline configuration. The segments within each bar reflect the breaking down of benchmark execution time into the various categories of progress or stall conditions. Appendix Section C.2 gives a complete explanation of the cycle accounting categories and some necessary caveats. For now, we concern ourselves only with a few prominent issues.

First, only the bottom two bars, unstalled execution (meaning instructions are being retired) and floating-point scoreboard (meaning the machine is stalled waiting for a result

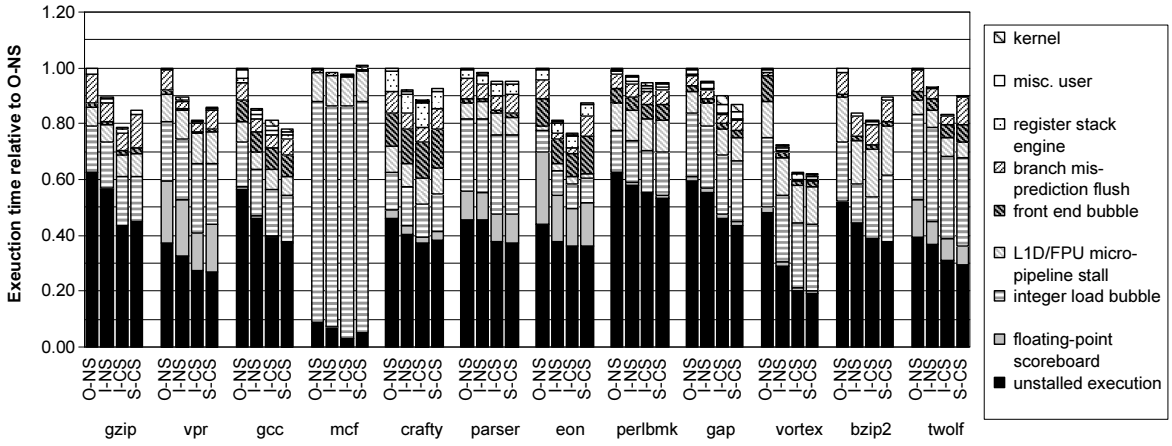


Figure 2.6 Cycle accounting detail.

from a floating-point operation),[16] reflect cycles of activity that are anticipated by the compiler. These generally account for fewer than half the total cycles accounted, posing a significant challenge for an EPIC compiler. A visual inspection of Figure 2.6 reveals that most of the benefit yielded by CFS transformations (particularly with respect to **S-CS**) comes from compacting these anticipated cycles. (Predication in **I-CS** also has a beneficial effect on branch misprediction and front end bubble cycles.)

By far the most important observation from these data is the preeminence of data memory stall time, practically across the board. On average, in one third of cycles (in the **O-NS** configuration) and two-fifths of cycles (in the **I-CS** configuration), the processor is stalled on a value requested from the data cache hierarchy. This can be due to either latency (integer load bubble and, to some extent, floating-point scoreboard) or collisions in data access (L1D/FPU micropipeline stall).[17] In some cases, the domination is quite extreme. 95% of cycles are spent in such a state in *mcf* and 58% in *vortex*, after aggressive optimization. This latency is not generally exposed to the compiler, and is thus immune from optimization by typical ILP techniques. Results show (this will be discussed in more detail later) that the ILP techniques, and specifically control speculation, at least do not generally *increase* the memory latency stall time, as might have been assumed from previous results [38] which suggested that spurious cache misses contributed up to

[16]The Itanium architecture implements some integer mathematical operations in floating-point units.

[17]L1D micropipeline stalls result from events such as memory request recirculations, data TLB misses, queue and store buffer overflows, and other irregularities in the memory subsystem.

half the misses in several applications. The application of these observations in relating this work to previous, simulation-based evaluations is dealt with more fully in Chapter 10.

Another important observation has to do with the role of predication in delivering performance in EPIC systems. Generally, in those applications where **I-CS** outperforms **S-CS**, the benefit from the **I-CS** framework only narrowly exceeds the difference observed in branch misprediction flush and front-end stall categories. In addition to eliminating the effects of mispredicted branches removed, it does succeed to some extent in overlapping independent control constructs, achieving higher ILP even while the machine would not have been otherwise stalled. These instances are not, unfortunately, very obvious at the benchmark level.

More specific observations from these data will be discussed as they become relevant, later in the presentation.

2.4 Introducing the Detailed Presentation

Operating within this experimental context, the next six chapters address specific components of the compiler framework and their roles in producing the indicated levels of performance. Chapter 3 discusses control flow structural optimization at a high level. Chapter 4 introduces more specifics on the implementation and implications of code specialization, something that has not previously been characterized in this level of detail in the modern context. Chapter 5 discusses the predication apparatus of the IMPACT compiler, and the adaptations it required to make it a productive element on Itanium. Chapter 6 discuss control speculation, the area of the compiler that required the most adaptation to render it generally safe and useful in the real-machine context. Chapter 7 discusses procedure inlining, which is generally a necessary fertilizer for subsequent ILP transformations. Chapter 8 covers special topics in implementation and results. Chapters 9 and 10 describe the performance analysis framework that delivered the presented results and some selected related work, respectively. Chapter 11, finally, presents some concluding thoughts.

3 CONTROL-FLOW-STRUCTURAL EPIC COMPILATION

This dissertation refers to the IMPACT compiler's approach to cultivation of instruction-level parallelism as the *control-flow-structural* (CFS) approach. In this approach, a program's control flow is first radically restructured into cohesive regions; parallelism is then cultivated within these regions by means of predication, speculation, acyclic and cyclic (modulo) scheduling, height reduction, and specialization-dependent optimizations. The formation of appropriate regions is thus foundational to the CFS approach. This dissertation's experimental results, to be detailed later, show this region-based[1] approach to be highly effective, with the assumption that representative control flow profile information is made available to the compiler. This chapter lays out the techniques used to develop these basic regions, describing the important features of the heuristics controlling this process and the prominent pitfalls that can decrease its effectiveness.

3.1 Impediments to ILP

Any discussion of ILP cultivation techniques must begin with the obstacles that must be surmounted to expose it effectively. These obstacles form the problem statement, not only for this chapter, but also for the chapters to come. While these barriers fall into

[1]The use of the term "region-based" here is to be distinguished from the use in the work of Hank et al. [69, 70]. In that work, the term referred to a selection of optimization regions across function boundaries, a generalization of procedure inlining, rather than the techniques used to form extended, potentially predicated basic blocks.

four broad categories, we devote the most attention here to the first category, control, which has the most basic connection to the problem of region formation:

3.1.1 Control obstacles

By definition, in control-intensive programs, control operations are frequent (one out of seven instructions is a branch in **O-NS** code). CFS optimization is designed primarily to eliminate them (one out of twelve instructions is a branch in **I-CS** code) or to mitigate their effects. In imperative programs, branches serve two purposes: first, they form the decision-making apparatus of the program, deciding how the control flow graph will be traversed; second, they delimit groups of instructions having the same execution condition. Like any modern architecture, an EPIC microarchitecture usually succeeds for the most part in hiding the latency of the decision-making aspect with branch prediction. Unlike other microarchitectures, however, EPIC does not provide for the runtime intermingling of operations across a branch. Lam and Wilson [10] emphasized the importance of this obstacle to extracting useful amounts of ILP, especially for VLIW machines.

The previous paragraph looked at branches from the perspective of the hardware. Formally, from the compiler's perspective, branches constrain the execution of later operations by *control dependence* relations.[2] Within the context of a (single-entry) trace, the hardware and software views of control dependence are identical. If one considers prior tail duplication, however, the motion of an instruction above a side-exit branch in a trace may not in fact have the effect of executing the instruction in a new control domain (that is, its copy will surely execute in the tail, so it is not, strictly speaking, control speculative). The IMPACT compiler ignores this possibility, treating any instruction as control-speculative that it moves above a branch.

[2]Control dependence here is used in the same sense as in [61, pp. 267 ff.], in which an operation is control dependent on a branch if the branch may determine the number of times the operation may execute.

Predication, as it converts control into data dependence and allows instruction-granularity execution control, enables the compiler to attack both these aspects, simplifying the program control apparatus, removing branches that might otherwise mispredict, and permitting the general intermingling of instructions having different execution conditions. Control speculation allows likely-to-execute instructions to move above their controlling branches. Compile-time, structural code transformations to eliminate control inefficiency are a primary enabler of EPIC performance and the central focus of this study. These compile-time solutions to control obstacles, however, generally require predictions about runtime conditions, leading to the general problem of profile dependence.

3.1.2 False dependences

Memory accesses and subroutine calls can pose artificial barriers to both optimization and scheduling, if their dependences are resolved only conservatively. Alias and array dependence analysis aim to determine the true, minimal set of dependences needing to be drawn among these operations to preserve program correctness. IMPACT applies sophisticated interprocedural pointer analysis algorithms [59] and Pugh's Omega Test [71] to reduce spurious dependences. Informal studies of the highest-execution-frequency, unresolved store-to-load dependences across the studied benchmarks suite indicate that these are usually due to array indirection, manual allocation pooling, or other very difficult cases, and not to unintentional shortcomings in these analysis algorithms. Data speculation support may be useful in addressing these cases, but requires another center of speculative decision-making in the compiler [72].

3.1.3 Occasional dependences

Out-of-order processors today successfully reorder loads and stores, checking for runtime dependences and stalling, buffering, or replaying operations as necessary to preserve program correctness [20]. This allows them usually to reorder "mostly independent" operations, something the compiler cannot do statically without additional hardware support for data speculation [57, 73]. The IMPACT compiler does not currently make use of this feature on IA-64, although it holds promise. Since the frequency of actual dependence

dictates the recovery burden, compile time guesswork similar in concept (but different in substance) to that applied in eliminating control dependence must be applied.

3.1.4 Nondeterminism

Finally, variable-latency and potentially-excepting instructions, such as loads, pose a challenge for statically scheduled machines, as is evident from the fraction of machine stall cycles due to data cache misses. The compiler can manage only those latencies it can anticipate. The dampening effect of data cache stalls on any ILP-oriented optimization technique was noted already in Superblock research [22]. Complicating the basic scheduling problem, when speculated, each off-path instance of these operations becomes a potential performance "landmine." This is of particular concern because the ILP techniques themselves tend to increase the effect, rather than simply be diluted by it, and became a topic of later EPIC research [38]. This dissertation improves on the understanding of this problem. As Chapter 8 will discuss, region formation can have either positive or negative effects on nondeterministic behavior. For example, while Hyperblock formation generally reduces the number of cycles stalled recovering from branch misprediction, experimentation shows that a Superblock-centered formation strategy sometimes increases these penalties. Various software prefetching schemes [54] and microarchitectural additions [28, 29] have been proposed to help "smooth over" these troublesome exposures of latency.

3.2 Control-Flow-Structural Transformation in the IMPACT Compiler

As was just noted, CFS techniques focus primarily on mitigating control barriers to ILP. Like an out-of-order machine, EPIC compilers exploit ILP across branches by relying on the notion that program execution is comprised mostly of a composition of stable, predictable traces through the control flow graph [32]. Compiler-based trace selection and EPIC features together allow for improved region selection, region customization, and decision interleaving difficult or impossible to conceive in the traditional instruction-set architecture. First, by using predication to control the execution of instructions

individually, the compiler may incorporate multiple program paths into a single trace, or *Hyperblock* [36]. This increases the scalability of trace formation, as it counters the (in the worst case exponential) code growth entailed in expanding each path into its own Superblock trace. Furthermore, because the execution of instructions is no longer solely determined by the position of instructions relative to branches, ILP may be exploited more freely among groups of instructions with independent execution conditions. The example of Figure 3.7, to be discussed in detail shortly, illustrates this effect. Finally, the selection of these large, stable, traces at compile-time allows for extensive and efficient optimization of enclosed computation. Code sequences can be specialized for their path context, and speculation can be performed easily and efficiently within the trace by moving instructions up past branches or weakening operations' predicates, allowing them to bypass their predicate definitions in the schedule but permitting them to execute more frequently. This is the essence of control-flow-structural transformation.

Various CFS transformations are employed together in the IMPACT compiler. The techniques are introduced briefly here, with detailed explanations saved for the chapters specific to them. Some recurring themes bear mentioning: CFS transformation specializes code for the purpose of enhanced instruction-level parallel issue. As a side-effect, CFS introduces opportunities for specialization-based optimization. The primary costs of CFS transformation are code expansion and costs of off-path speculative or off-path predicated operations.

We need to introduce briefly the various aspects of CFS transformation in the IMPACT compiler before proceeding to some illustrative examples. In the course of this survey, the important characteristics of the approach are exposed—chief among these is the degree to which each transformation is speculative, hedging that subsequent transformations will leverage its aggressiveness to yield a benefit.

3.2.1 Procedure inlining

The boundaries between procedures are often put in place for reasons other than code compilability: code readability, modularity, etc. [69] The IMPACT compiler uses profile-guided procedure inlining (including inlining of indirect call sites) to create larger

regions within which to apply ILP development and specialization techniques [74]. This is an *a priori* approach, meaning that the inlining of procedure calls is performed before it is known what ILP transformations and optimizations could be performed across the inlined interface. Furthermore, the code specialization effected by inlining of a procedure called from multiple call sites creates uncertainty in the profile used to guide subsequent transformations.

In this dissertation's experiments, inlining delivered an average 13% performance improvement (up to 40% in *eon*) when applied in combination with full ILP optimizations (**I-CS**). Although extensive inlining was performed (resulting in approximately a 1.5× increase in the number of static instructions touched across the suite), performance was not noticeably degraded by code footprint growth. This salutary result has to do both with benchmark code structure, which does not present many opportunities for footprint growth to overburden the instruction cache hierarchy, and also with the fetch efficiency-improving effects of inlining and subsequent CFS transformation. (Transformations reduce fetched `nop` count by 38% across the suite, in both Superblock and Hyperblock configurations). This means more useful instructions are fetched per instruction cache access, combating the footprint growth that ordinarily accompanies inlining and other CFS transformations.

These effects, and other interesting facets of inlining for CFS, are described in detail in Chapter 7.

3.2.2 Superblock and Hyperblock formation

The fundamental element of CFS transformation is a region of code specialized for optimization. The Superblock, a single-entry, potentially multiple-exit region, is the first conception of this type of region that we will consider [22].[3] A Superblock is formed by selecting a single, frequently executed path through a control flow subgraph, then removing all of the trace's side entrances by duplicating all subsequent code in the trace (a process called *tail duplication*), and, finally, rendering all blocks on the trace into

[3] A detailed historical discussion, which provides some useful insights from earlier trace scheduling work, is deferred to Chapter 10.

a single, extended basic block. Within this trace, instructions are free to be speculated without concern for side entrances (contrary to the more complicated case in earlier trace scheduling models that did not replicate code [44]). Superblocks also facilitate increased on-trace optimization, as will be discussed later. The costs of this transformation include the code replication due to tail duplication (generally less than a 10% increase in touched code size with reasonable parameters) and the penalization of execution that does not follow the expected trace. Further details are described in Chapter 4. Superblocks aim to fulfill the apocryphal computing maxim, "Make the common case fast."

If Superblocks make the common case fast, a natural question is how one might make the common case more common. In some program segments there is no single, clearly dominant path. With predication, the Hyperblock [36] introduces the possibility of incorporating multiple traces into a single specialized region. The Hyperblock is simply a single-entrance, potentially multiple-exit region in which individual operations may be predicated. The if-conversion algorithm [75, 76] allows a collection of traces sharing a single entry point to be converted to such a predicated, extended basic block. The use of predication in Hyperblocks has two primary, salutary effects: first, mispredictions attributed to branches eliminated in the if-conversion process are eliminated; second, control-independent instructions may be interscheduled freely within the Hyperblock without code replication or introduction of burdensome control flow. The second effect is rarely mentioned but extremely useful when it occurs. One good example is in modulo scheduling (software pipelining). Pipelining a single predicated, extended basic block loop is much easier than pipelining the corresponding loop implemented with control flow [77]. The details of Hyperblock formation and optimization are reserved for Chapter 5.

The decision-making process of which and how many paths to incorporate into each specialized region is complex and not thoroughly explored. Making regions too specialized (tending toward the single-trace Superblock, for example) may fragment code, generating excessive tail duplications and increased complexity. Incorporating too many paths, on the other hand, may penalize the common case or impede certain optimizations. Heuristics in this area rely heavily on control profile information and, in the case of Hyperblock formation, in which multiple traces of instructions must compete for resources

and co-exist in a single path of execution, on estimates of path dependence height and instruction count. In some cases, compiled-code performance is very sensitive to variations in the parameters controlling this process. Decision-making is complicated by the fact that formation, like inlining, is done prior to the optimization and scheduling that will determine the final characteristics of the code segment.

3.2.3 Ancillary transformations and optimizations

A variety of other transformations assist the region former in including more code into traces. These include loop unrolling and loop peeling (tail duplication is considered an integral part of the region former).

Loop unrolling is applied less frequently in this experimentation than in previous work with the IMPACT compiler. The full-featured support for software pipelining on Itanium has rendered iterative modulo scheduling generally a better solution. The massive unrolling traditionally performed after region formation, in some contexts, allows both overlap of iterations and significant cross-iteration optimization. In SPEC CINT2000, however, experiments show that this optimization has little opportunity to buy performance, and modulo scheduling achieves a better overlap with much less code expansion.[4] Limited unrolling is still performed on single-block loops not marked for software pipelining, and a new schema of unrolling (unrolling under a predicate) has been added to address specific issues in software pipelining loops with specific patterns of memory access. This technique accelerates *bzip2* significantly by reducing spurious store forwarding stalls.

Hyperblocks must have single points of entry; this precludes their incorporating loop bodies. In many cases, it is beneficial, though, to include the first or the first few iterations of an inner loop in a Hyperblock region. This is accomplished by loop peeling, a code replicating transformation that pulls iterations out of loops into the preheader. Chapter 5 addresses the loop peeler, since it is operative only in the Hyperblock framework.

[4]Less than a 1% difference in performance was observed between the unrolled and modulo scheduled versions of these applications in the **I-CS** configuration.

After region formation, additional optimizations continue to increase region size. The most notable of these is branch target expansion. Superblock and Hyperblock formation are limited, in many applications, in their ability to by themselves generate large execution regions, even when aided by loop peeling and unrolling. Branch target expansion finds control flow arcs from the end of one "source" extended basic block to another (the "target") that are frequently occurring and likely to be taken (relative to a given execution of the block). Provided that certain other criteria are satisfied, the target block is expanded (copied) into the source block. If the target block had multiple predecessors, this involves code expansion. In past experiments, this code growth was not meaningfully constrained, with the intent of producing execution regions of at least a minimum size (around 256 operations), if accommodable within the control structure of the program after Superblock and Hyperblock formation. This approach caused unacceptable levels of code expansion in several benchmarks. This occurred most notably in *crafty*, in which a 2.4× expansion in static, executed operations caused most of the Superblock approaches gains to be offset by increased instruction cache miss-related stall time. The controlling of this process, described in Chapter 4, eliminated these degradations without substantially decreasing the gains achievable from this approach, even in an ideal-cache sense.

Finally, the developed regions are exploited to perform enhanced optimization and scheduling. Registers are renamed to eliminate anti- and output-dependences. Induction variables are manipulated to increase parallelism, especially in unrolled loops. Common optimizations, such as copy propagation and common subexpression elimination, observe increased opportunities because of path specialization. Control-height reduction techniques such as branch combining attack control in the more manipulable form it is rendered into within Superblock and Hyperblock regions. These further optimizations can have a substantial effect on total code performance (up to a speedup of 1.12 in *vortex*), although their effectiveness is reduced in this context, relative to previous experiments.

3.2.4 Control speculation

The ability to schedule operations easily with respect to control flow transfers was the primary motivation behind the development of CFS techniques. Mechanisms for

performing and controlling such speculation are thus implied. A general speculation schema for control speculation is provided in the IMPACT compiler. This is employed both during optimization and during instruction scheduling to move even potentially-excepting instructions across branches. Predicate dependences are removed via predicate promotion (the weakening or removal of guard predicates on predicated operations, in effect a control speculation), allowing increased scheduling and optimization freedom and permitting reduction of predicate computation networks. General speculation delivered substantial performance gains in this work, but had to be rendered safe with special compiler techniques. Chapter 6 discusses these speculation mechanisms in detail. Control speculation contributes an average speedup of 1.07 across the suite.

3.2.5 Instruction scheduling and software pipelining

Finally, we return to the topic that motivated the initial development of CFS transformation (that is, in its development from trace-based work)—instruction scheduling. In keeping with the original Superblock philosophy, in the IMPACT compiler, instruction scheduling is a local (per predicated extended basic block) operation. This renders the potential for beneficial instruction scheduling entirely dependent on the formation of appropriate regions, unlike in other compilers that generally employ forms of global scheduling. The IMPACT compiler also performs iterative modulo scheduling [33] on the same PEBB regions. Modulo scheduling delivers occasional benefits in SPEC CINT2000, with the greatest being a speedup of 1.12 in *gcc*.

3.3 Case Studies in Complex Region Formation

As much as any ILP compiler, IMPACT is much more at home in "loopy" programs—programs whose execution time is dominated by manageably sized, high-trip-count loops. A benchmark dominated by these kernels is easy to optimize for ILP because they make the necessary region formation decisions rather obvious and because the secondary effects of intensive optimization generally fall upon unimportant, infrequently traversed code outside the loops. Large, complex programs less dominated by small, high-trip-count

loops make for much more of a challenge. Anyone with experience compiling SPEC CINT92, CINT95, and CINT2000 can easily identify the trend away from kernel-bound programs toward more complex cases, and the author believes this accurately reflects the trends in important nonnumeric applications in general-purpose computing today.

Region formation in control flow structural compilation gets complicated in the presence of intense and irregular (not "loopy" in the classical sense) control. To illustrate the dramatic nature of transformations applied, some examples from the benchmark suite are employed. These examples demonstrate how the various techniques, such as Hyperblock and Superblock formation, loop peeling, and other ancillary transformations unite to produce highly tuned, special versions of code for efficient execution of common cases. In the *gzip* example, in particular, these transformations are crucial for achieving outstanding performance. (IMPACT achieves a SPEC CINT2000 ratio 20% higher than the best commercial compiler on the Itanium Linux platform.[5]) In the success of these techniques, however, there lurks a danger, as well, as the specialization of the assumed-to-be-common paths comes at the cost of a sometimes substantial penalty in code expansion or in direct penalization of those paths adjudged to be less important.

3.3.1 An example from *gzip*

Figure 3.1 shows the main deflation matching loop (originally included in the function `longest_match()`, but considered here in the context of `deflate()`, into which it is inlined) of the SPEC CINT2000 benchmark *gzip*. This loop consumes the majority of the application's processing time, is highly control intensive,[6] and will require extensive transformation to achieve peak performance on an EPIC machine. A control flow profile indicates, fortunately, that not all paths through the loop are equally likely. This bias presents an opportunity for specialization of the common case(s).

There are two factors in selecting and specializing cases: optimization potential and execution frequency. Clearly, if the optimization of a specific path or set of paths is to be

[5]Intel *icc* version 8.1 with `-O3 -ipo -prof_gen|prof_use`.

[6]For those unfamiliar with the C language, the operators "`&&`" and "`||`" short-circuit; that is, if the result of the expression is evaluable without evaluating the second operand, the second operand is not evaluated. This therefore implies additional control flow [78].

```
1 do {
2    match = window + cur_match;                          // block A
3    if (match[best_len]   != scan_end  ||                // block A
4        match[best_len-1] != scan_end1 ||                // block B
5        *match            != *scan     ||                // block C
6        *++match          != scan[1])
7       continue;
8
9    scan += 2, match++;
10
11   do {} while (*++scan == *++match && *++scan == *++match &&
12                *++scan == *++match && *++scan == *++match &&
13                *++scan == *++match && *++scan == *++match &&
14                *++scan == *++match && *++scan == *++match &&
15                scan < strend);
16
17   len = MAX_MATCH - (int)(strend - scan);
18   scan = strend - MAX_MATCH;
19
20   if (len > best_len) {
21      match_start = cur_match;
22      best_len = len;
23      if (len >= nice_match) break;
24      scan_end1  = scan[best_len-1];
25      scan_end   = scan[best_len];
26   }
27 } while ((cur_match = prev[cur_match & WMASK]) > limit // block X
28          && --chain_length != 0);                     // block Y
29 ...                                                    // block Z
```

gzip is public domain software distributed under the GNU General Public License.

Figure 3.1 Code example from *gzip* `deflate.c:397`.

profitable, it must offer some improvement over the generic execution afforded all paths. Simultaneously, however, since optimizing a particular path implies the imposition of a penalty on other, less likely paths, the path(s) to be optimized must dominate the region in terms of execution frequency. The inclusion of more paths in the specialized version increases the version's coverage but may decrease its optimization potential. The latter effect may be difficult to observe at the time of region formation.

The *gzip* example illustrates these principles. Figure 3.2(a) shows a stylized conception of the control flow graph for the loop shown in Figure 3.1. More frequently executed blocks are indicated in darker shades of gray, and control flow is considered to enter the top of the discs, representing basic blocks, and flow out the bottom. The cloud in the center of the inner loop contains additional, irrelevant blocks of code. The "block"

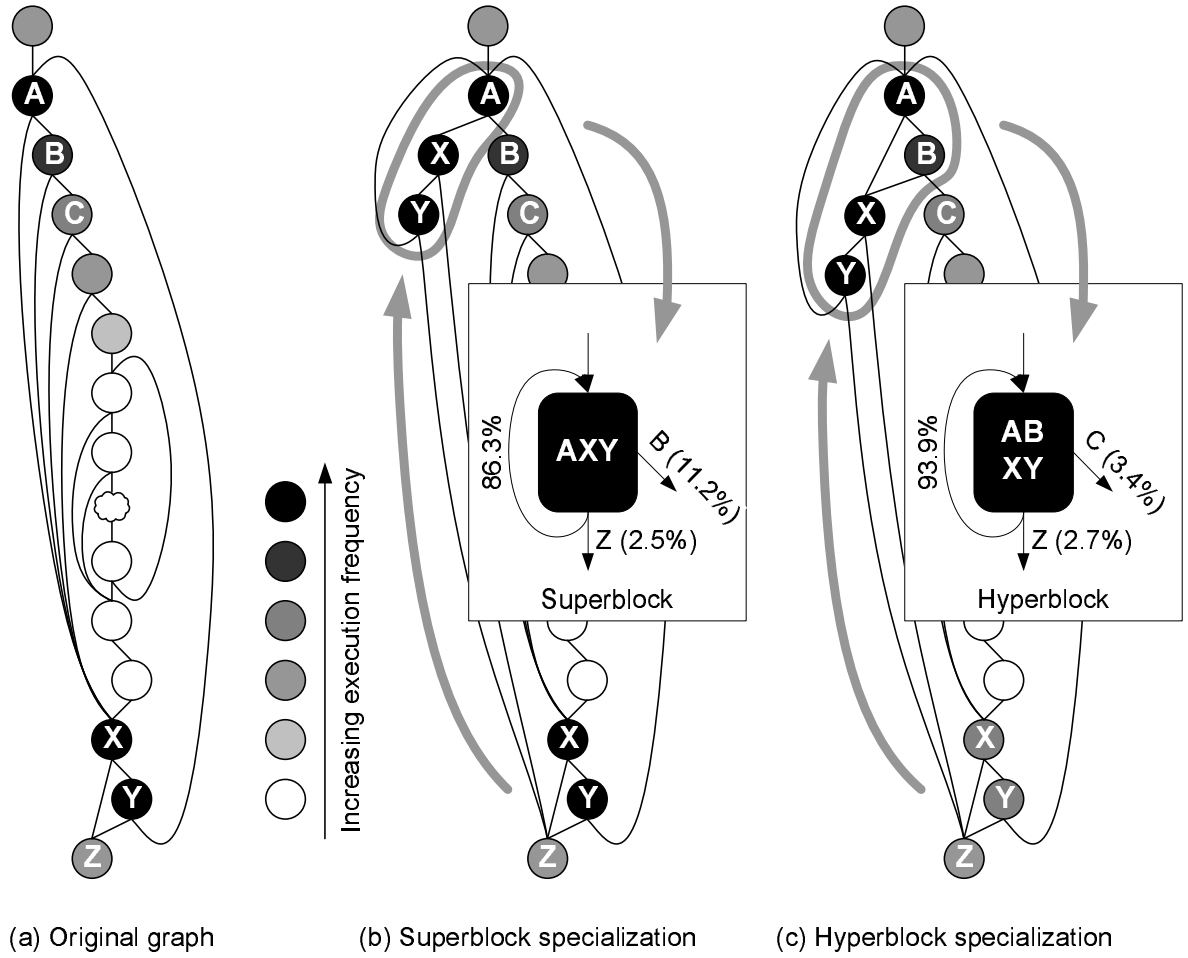


Figure 3.2 Inner loop version in *gzip* `deflate.c:397`.

comments in Figure 3.1 show the correspondence of the key lettered control blocks to lines of code in the original example.

The CFS transformation engine examines this loop for specialization opportunities. We first consider single-path (Superblock) opportunities. Here, we find that the most common path through the loop is $\mathsf{A} \rightarrow \mathsf{X} \rightarrow \mathsf{Y}$. Figure 3.2(b) shows the effect of specializing this path as a Superblock. Since a Superblock must have only a single point of entry, and since block X is entered not only from A but also from other blocks, blocks X and Y must be *tail-duplicated* to include them in the Superblock region. The upward arrow indicates this copying of these two blocks into the transformation region.

The inset in Figure 3.2(b) shows the control flow profile of the resulting Superblock. 86.3% of block traversals end with a continuation to the next iteration of the specialized

```
 1 .deflate_AXY:
 2         dep.z   loc10 = loc1,1,15 ;;
 3
 4         add     loc0 = loc10,loc21
 5         adds    loc2 = -1,loc3
 6         add     loc8 = [loc8] ;;            // match[best_len]
 7
 8         cmp4.eq.unc     p18,p0 = loc8,loc13
 9   (p18) br.cond.dpnt    .deflate_B ;;       // 11.2%
10
11         ad2     loc1 = [loc0] ;;            // prev[cur_match&WMASK]
12
13         cmp4.leu.unc    p17,p0 = loc1,loc12
14   (p17) br.cond.dpnt    .deflate_Z# ;;      //  2.5%
15
16         cmp4.ne.unc     p16,p0 = r0,loc2
17   (p16) br.wtop.dptk    .deflate_AXY# ;;    // 86.3%
18 .deflate_Z:
```

Figure 3.3 Superblock code for innermost loop version from *gzip* `deflate.c:397`.

version. 11.2% require traversal of paths excluded from the Superblock, so these proceed to the nonspecialized remainder of the loop body. 2.5% of traversals, finally, end with an immediate exit from the loop. The Itanium assembly code generated for this specialized version (with `nop` operations removed for simplification) is shown in Figure 3.3.

To maximize performance of this specialized case, this "hot," extracted loop is modulo-scheduled, producing the result shown in Figure 3.4. Control speculation is applied to the potentially-excepting load from the `prev` array (line 8), freeing it from its previous position of control dependence under the branch to block B. This particular schedule achieves the target initiation interval [33], achieving the highest possible performance, at least within this special version. This is not, however, the final solution for this loop nest. Having seen how the common case can be made fast, can it be made more common? A loop continuation ratio of only 83.5% does not likely result in many successive iterations of the loop, making this potentially not a particularly useful version.

Fortunately, predication provides the option of incorporating multiple traces into specialized regions. Figure 3.2(c) shows such an alternative. Referring back to Figure 3.2(a), one may note block B, which follows block A and implements the test on line 4 of the code shown in Figure 3.1. This test is required in 11.2% of the loop body traversals

```
1 .deflate_AXY:
2         // II=3, 2 stages, loc0 -> loc3 rotate
3         add     loc0 = loc10,loc21
4         add     loc8 = loc1,loc11
5         adds    loc2 = -1,loc3 ;;
6
7         ld1     loc8 = [loc8]               // match[best_len]
8         ld2.s   loc0 = [loc0]               // prev[cur_match&WMASK]
9         cmp4.ne.unc     p16,p0 = r0,loc2 ;;
10
11        cmp4.leu.unc    p17,p0 = loc0,loc12
12        dep.z   loc10 = loc0,1,15           // (from iteration i+1)
13        cmp4.eq.unc     p18,p0 = loc8,loc13
14  (p18) br.cond.dpnt    .deflate_B#         // 11.2% taken
15  (p17) br.cond.dpnt    .deflate_Z#         //  2.5% taken
16  (p16) br.wtop.dptk    .deflate_AXY# ;;    // 86.3% taken
17 .deflate_Z:
```

Figure 3.4 Superblock code after software pipelining (*gzip* `deflate.c:397`).

(whenever the test on line 3 fails). Its exclusion from the specialized Superblock case is primarily responsible for early exits from the specialized, single-block AXY loop. The ability to include multiple paths under predication permits the inclusion of this additional test, increasing the coverage of the specialized case. Figure 3.5 shows the assembly code resulting from inclusion of this block, after predicate promotion and modulo scheduling.

Even in this simple example, several factors regarding the performance of Hyperblock-transformed code can be elicited:

- While Superblock formation (and its follow-on partner, branch target expansion) merely synthesize common path traces, controlling Hyperblock growth involves factors in addition to control flow weights, some of which can be difficult to anticipate at formation time. These include the dependence height and instruction count of paths to be included in the parallel structure of the Hyperblock.

- The performance of Hyperblock code depends on control speculation in the form of predicate promotion. The inclusion of the less-common A $\rightarrow$ B path increases the dependence height of the loop, unless the control dependence from the predicate

```
 1 .deflate_ABXY:
 2         // II=3, 2 stages, loc0 -> loc5 rotate
 3         ld1     loc8 = [loc11]              // match[best_len]
 4         ld2.s   loc0 = [loc10]              // prev[cur_match&WMASK]
 5         add     loc4 = loc5,loc17
 6         adds    loc2 = -1,loc3 ;;
 7
 8         ld1.s   loc9 = [loc4]               // match[best_len-1]
 9         cmp4.eq.unc     p16,p0 = loc8,loc18
10         dep.z   loc8 = loc0,1,15            // (from stage 0)
11         add     loc4 = loc0,loc14
12         cmp4.leu.unc    p17,p0 = loc0,loc15
13         cmp4.ne.unc     p18,p0 = r0,loc2 ;;
14
15   (p16) cmp4.eq.unc     p19,p0 = loc9,loc16
16         add     loc11 = loc4,loc12
17         add     loc10 = loc8,loc13
18   (p19) br.cond.dpnt    .deflate_C#         //  3.4%
19   (p17) br.cond.dpnt    .deflate_Z#         //  2.7%
20   (p18) br.wtop.dptk    .deflate_ABXY# ;;   // 93.9%
21 .deflate_Z:
```

Figure 3.5 Hyperblock code after software pipelining (*gzip* `deflate.c:397`).

definition in line 13 to the load in line 8 is broken by predicate promotion, as is the case in Figure 3.5.[7]

- Speculation in the form of Superblocks allows the advance execution of operations along one predicted future control path. In Hyperblocks, however, speculation is multipath in nature. This can result in riskier control speculation than the single-path model. Here, the load operation from the relatively infrequently traversed block B (line 8 of Figure 3.5) is executed even on the *more frequently* taken path A $\rightarrow$ X. This does not occur in the Superblock model, so the reliance of the Hyperblock model on appropriate speculation control mechanisms is greater.

Due to these and other, related concerns that will be made apparent in Chapter 5, the use of if-conversion must be carefully regulated. The sensitivity of *gzip* and the other

[7]As it turns out, it is still possible, at least with human effort, to schedule the loop with the same initiation interval without predicate promotion, but increased control speculation of other operations (specifically, the loads in lines 1 and 2) is required to accommodate the longer dependence chain. A good schedule is definitely easier to achieve with predicate promotion.

benchmarks to variations in the heuristics used to select regions for if-conversion is an important aspect of the results of this dissertation.

The transformation of the loop structure in *gzip* `deflate()` is not complete with the versioning of the inner matching loop. An outer loop also offers opportunity for specialization. Figure 3.6(a) shows the stylized control flow graph for this outer loop, in which the inner loop dealt with in the example of Figure 3.2 is highlighted. Control flow profile information identifies a set of hot paths through this loop body, which are summarized by the trace drawn through the control flow graph. This hot path involves only a portion of the control flow of the outer loop, but includes a (single) traversal of the specialized inner loop version ABXY just produced. To create a Hyperblock from these hot paths, a single-entry version of the hot region must be produced, but this cannot be achieved with an inner loop contained in the trace. Fortunately, loop peeling allows a copy of the inner loop body to be pulled out into the new Hyperblock. Figure 3.6 shows the formation of the hot path Hyperblock. This transformation, enabled by the peeling of the inner loop version ABXY, and ones like it purchase a 5% improvement in benchmark performance for *gzip*, the strongest example of gain from loop peeling in these experiments.

3.3.2 A more complex example from *crafty*

Crafty, a chess program, serves as one of the most control-intensive of the SPEC CINT2000 benchmarks [5]. It includes not only intensively "branchy" code segments but also many reasonably serial and low-iteration-count loops. Exposing ILP requires finding ways both to eliminate branches (by creating efficient predicated regions) and to interleave execution from different loop bodies in an efficient way. These are common features of SPEC CINT2000 programs, but their necessity is particularly pronounced in *crafty*. Complicating the problem is *crafty*'s large instruction footprint, which threatens to erode the benefit of any transformations that result in increased code size, and its extensive use of large lookup tables, which pose problems for speculative execution of loads and their dependent successors.

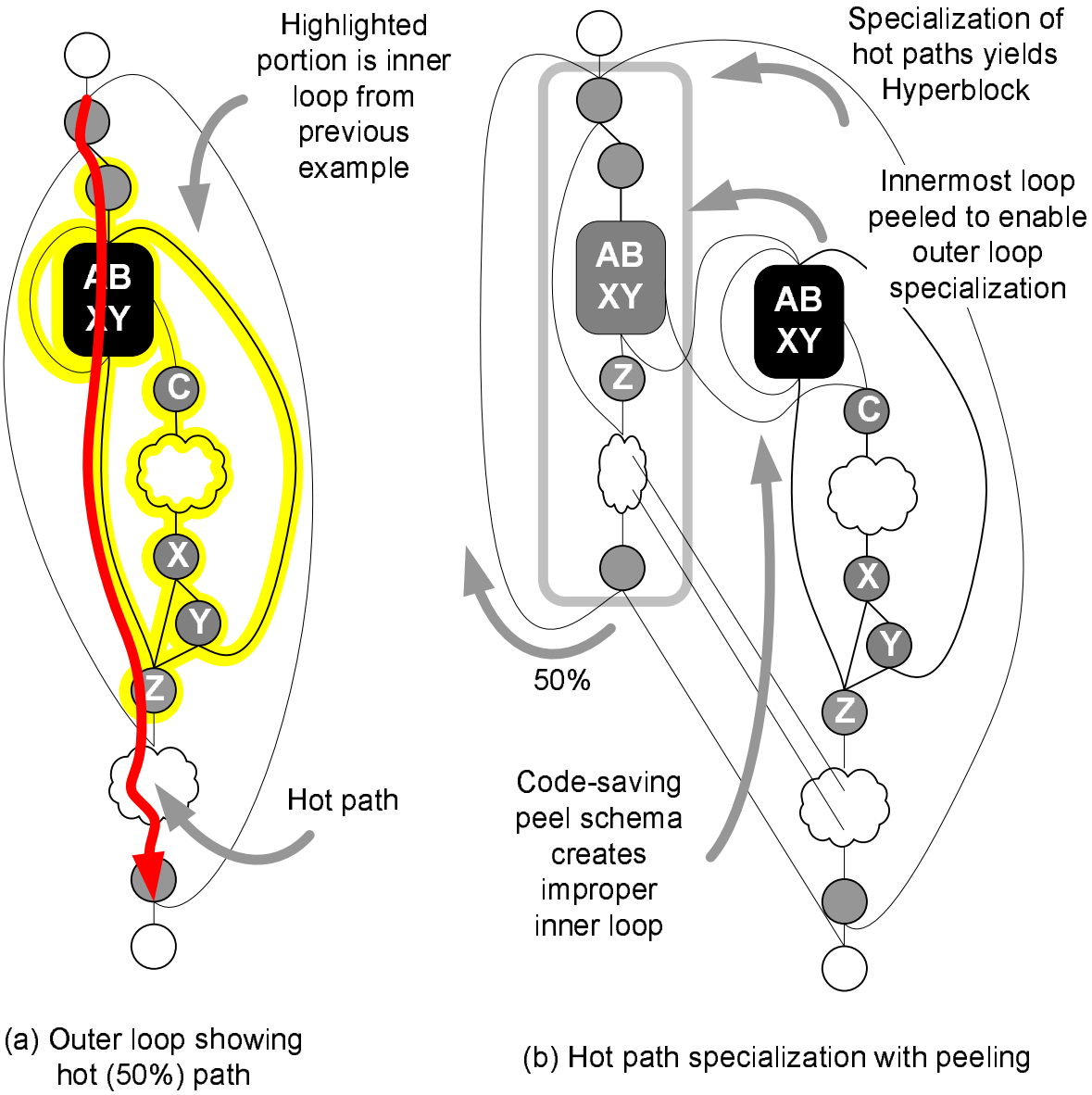


Figure 3.6 Outer loop versioning for *gzip* `deflate.c:678`.

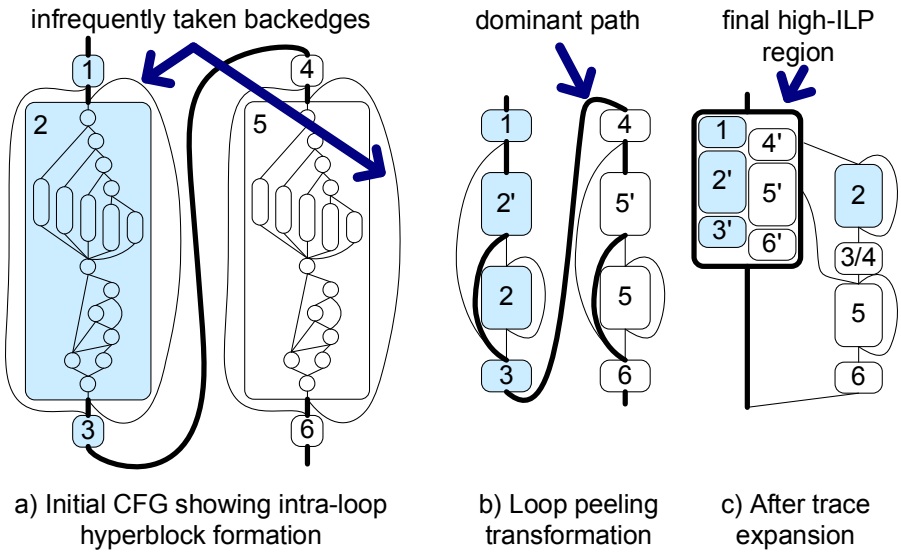


Figure 3.7 Exploiting ILP across *crafty* loops. Code is presented (a) with intraloop Hyperblock formation only, (b) after application of loop peeling, and (c) after trace formation.

The *crafty* function `Evaluate()`, which evaluates the strength of each player's position on a chess board, provides an example of sophisticated region formation. This function contains several sequential `while` loops, two of which are shown in Figure 3.7(a). Both loops contain substantial internal control flow, each loop has little inherent, intraloop ILP due to serial data dependences, and each loop body typically executes exactly once.[8] Simply forming Hyperblocks for each of the loop bodies, as indicated by the enclosing boxes 2 and 5 in (a), prevents misprediction and streamlines instruction issue but does not help develop additional planned ILP (each loop is inherently serial due to data dependences). More aggressive transformations, however, can exploit this situation. The code in (b) has been transformed using peeling; one iteration of each loop has been pulled out. Now, the ordinarily taken path (1 2' 3 4 5' 6) traverses the peeled iterations only; the original loops are left to "clean up" any unlikely remaining iterations. Finally, (c) shows the result of trace formation through the transformed region. The two Hyperblocks, once trapped inside loop back edges, are merged to form a single scheduling region. Predication

[8]These particular loops evaluate the position of the two players' queens; typically, each has a single queen.

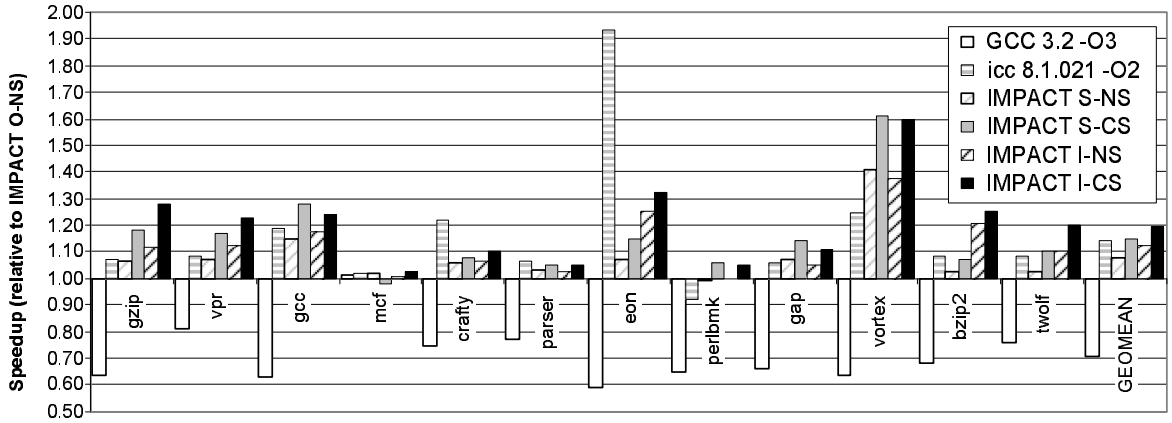


Figure 3.8 Speedup relative to **O-NS** baseline.

allows independent decisions (*cf.* the original control flow within the two loop bodies in (a)) to be made in parallel, and useful ILP is increased.

This example, representative of transformations applied throughout SPEC CINT2000 by the IMPACT compiler, typifies the "structural" approach to EPIC compilation, by which EPIC features enable radical transformation of program control structures in the search for more ILP. The costs of these transformations include an increased reliance on profile information (if the case in which neither loop executed any iterations becomes frequent, many useless, predicate-squashed instructions would be issued) and an increase in code size due to region-related code replication (this becomes significant if the remainder loops are traversed; otherwise, there is no negative impact on instruction cache footprint, and the untouched excess code can be placed harmlessly in a cold location).

3.4 Evaluating the CFS Approach

Now that we have developed a basic idea of the CFS techniques applied in the IMPACT compiler, we recall the speedup figure used in the previous chapter to motivate a discussion of the collective impact of these techniques on particular applications. SPEC CINT2000 provides a variety of sample cases for CFS transformation. For the reader's convenience, the performance comparison figure from the previous chapter, Figure 2.3, is reproduced in Figure 3.8. The two leftmost bars for each benchmark reflect results from *gcc* and *icc*, two other compilers on the Itanium platform. *Icc*, developed by Intel,

is the strongest commercial compiler available. These are added to indicate IMPACT's performance relative to other compilers.

3.4.1 Benchmarks with substantial effect

For four of the benchmarks, Superblock-based CFS transformation (**S-CS**) was remarkably effective, and if-conversion had little to add (and sometimes allowed degradation). These include *gcc*, *gap*, and *vortex*. Chapter 4 discusses the detailed results for this class of applications, as it evaluates single-path specialization.

Another subset of the benchmarks required if-conversion (**I-CS** framework) to achieve the greatest benefit. These include *gzip*, *vpr*, *eon*,[9] and *twolf*. Examples from these benchmarks will drive Chapter 5's discussion of multipath specialization. It should be noted that those benchmarks that do not appear to benefit from predication do often contain individual procedures with substantial benefit, but these gains are often offset by other losses in other procedures. These finer-grain results will be examined in Chapter 5.

Finally, the Burrows-Wheeler compressor *bzip2* benefited substantially from a special loop unrolling technique requiring both predication and speculation, as will be described in Section 8.4. This is an outstanding example of using the CFS techniques in a specially targeted way to mitigate compile-time recognizable microarchitectural penalties. Without this technique, *bzip2* benefits little from CFS transformation, as shown by the unimpressive **S-CS** result.

3.4.2 Benchmarks seeing moderate benefit

The benchmark *crafty* proved a challenge for CFS transformation. The Superblock-based **S-CS** approach achieved a relatively meager speedup of 1.08 relative to classical optimization. **I-NS** delivered a similar result. The **I-CS** configuration fared better, but still not spectacularly, delivering speedup of 1.10. This might suggest that specialization, predication, and speculation are doing little, and doing similar things, in this benchmark.

[9]It must be noted that IMPACT's performance on *eon* is well short of that achieved by production compilers, due to a lack of C++-specific optimizations and poor treatment of structure assignments containing floating-point values. Performance results for this application must, therefore, be scrutinized carefully.

This is, however, far from the truth. In fact, the transformations achieve several different victories but, in the end, surrender many gains to secondary penalties of CFS transformation. These include deleterious instruction cache effects, register oversubscription, and increased data load penalties due to multipath speculation. Profile variation, the effects of excessive code expansion, and nondeterministic behaviors also play important roles here (when compiled with the reference input, crafty achieves a 1.18 speedup rather than 1.10 in **I-CS**).

The benchmark *perlbmk* showed only a 5% improvement in performance due to application of CFS techniques (in the **I-CS** and **S-CS** configurations), but still delivers a respectable result when compared to the production compiler `icc`. Experiments showed that more aggressive application of if-conversion can increase *perlbmk*'s performance by another 3%, yielding a total speedup of 1.08 for CFS with predication. The biggest problem for *perlbmk*, though, is that its training profile input is not very representative of its evaluation input behavior. When compiled with the reference input, instead of the training input, *perlbmk* reaps a speedup of 1.21 in the **I-CS** configuration, underscoring the highly bias-dependent nature of CFS transformation.

Parser also gained only approximately 5% from CFS transformation. In its case, though, the training profile is reasonably representative; it seems simply to be a rather serial application at the instruction level.

3.4.3 Benchmarks with little potential for effect

The execution time of the benchmark *mcf* (on Itanium 2) is is totally dominated by data cache miss stall time. Wu described a stride-detecting prefetching scheme, employed in the Intel production compiler, that allows *mcf*'s data references to be prefetched effectively, even though they are concentrated in linked data structures [54]. This roughly doubles *mcf*'s performance. Even with these techniques, 88% of *mcf*'s execution time would be spent waiting for memory (a remarkable 95% is spent in this fashion in IMPACT-compiled code today), so little benefit from CFS is to be expected here.

3.4.4 General observations and principles

Let us conclude this topic with a brief summary of the challenges to be addressed in the chapters to follow.

- Inlining, region selection, and transformation all depend on the availability and knowability of run-time program biases. In a multipath model (with if-conversion) decisions must also consider path compatibility (issue width and dependence height). Control speculation is assumed in managing these issues, but its effects cannot be fully appreciated at the time regions are formed. The distribution of these decisions throughout the compiler can cause negative interactions.

- Code-replicating CFS optimizations tend actually to improve instruction fetch efficiency. This effect can offset the expected cache-thrashing effect of massive code replication, often yielding net benefits in instruction cache behavior. In some cases, however, clearly excessive replication is performed and instruction cache performance suffers greatly.

- Just as CFS can as a side-effect improve instruction cache performance, it can enable substantial side-effect optimizations. While one would expect to see a typical increase in instructions executed with speculation-laden CFS compilation, one often finds a decrease in total instructions executed due to the success of optimizations in specialized regions. The benefits due to these optimizations are difficult to isolate from "pure ILP" benefits.

- Selection of the desired degree of path multiplicity in specialization regions is a difficult problem. As noted in the above example from *gzip*, there is a fundamental tension in performing CFS transformations between optimizing with the greatest intensity on minimal sets of paths through a region (ultimately, on single paths in Superblocks), running the risk of achieving insufficient coverage, or trying to balance multiple paths in a single Hyperblock region, running the risk of impeding component paths. The remaining chapters will shed more light on this difficult topic.

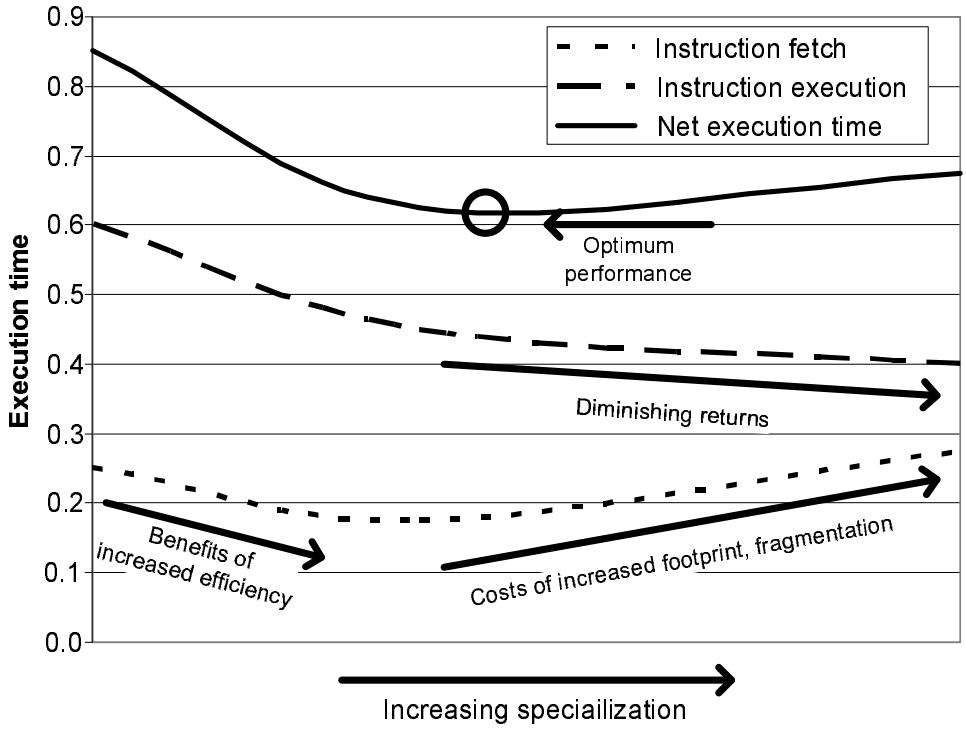


Figure 3.9 Performance sketch for CFS transformation.

- As already seen, CFS transformation depends intensely on control profile information. As increasingly aggressive transformations are employed, achieving higher performance for program executions similar to the one used in compiler training, paths used in unforeseen executions are necessarily neglected or even penalized. This can have performance consequences, even within the controlled environment of SPEC CINT2000.

3.5 Specialization and Instruction Fetch Performance

This chapter concludes with one important, cross-configuration observation regarding the effect of specialization on instruction fetch as a component of total performance. A good perspective on this issue is necessary for understanding the other results, and to justify the approach taken toward achieving them in this dissertation. Figure 3.9 shows a much simplified, stylized representation of the effects of specialization (including inlining, Superblock, Hyperblock and ancillary transformations) on application performance, as the degree of specialization is varied. Two components of end performance are represented

individually: instruction fetch and instruction execution. Instruction fetch represents those cycles wasted because instructions could not be delivered fast enough to keep execution units busy every cycle (and, secondarily, the time spent adjusting the register stack for new procedure contexts). Instruction execution denotes the portion of time that instructions have been delivered and are actually executing. Net execution time is the sum of these two quantities.

As specialization is increased,[10] three interacting processes occur. First, specialization increases the efficiency of instruction fetch and issue by producing customized code for particular, important paths and, in the case of predicated specialization, by removing branch mispredictions. Initially, this beneficial process dominates the effect of specialization on instruction fetch overhead. Gradually, however, the penalties associated with an expanding code footprint of ever more diluted specialized versions begin to dominate, and instruction fetch overhead is penalized. This interaction is indicated by the dotted line in Figure 3.9. At the same time, specialization provides opportunities for efficient instruction speculation and optimization, reducing instruction execution time (as indicated by the dashed line in Figure 3.9). These three processes and two performance components combine to yield the net execution time (the solid line).

These effects can be examined concretely in comparison of code expansion and instruction cache performance data. Figure 3.10 shows the increase in static, touched instruction counts with various types of CFS transformations. In this figure, the central, black column represents the growth of **I-CS** code (relative to the **O-NS** configuration). A typical case allows for approximately a 1.5× expansion in the number of touched operations. Approximately half this increase comes from the tail duplication and peeling involved in predicated region formation (as indicated by the gray bar to the left); the other half comes from subsequent Superblock, branch target expansion, and ancillary transformations (such as unrolling). The white bar on the right indicates the code expansion due to Superblock, branch target expansion, and ancillary transformations in

[10] As will be detailed elsewhere, "specialization" is not a simple, one-dimensional variable, but consists of several interrelated techniques. This chart is simplified for purposes of illustration.

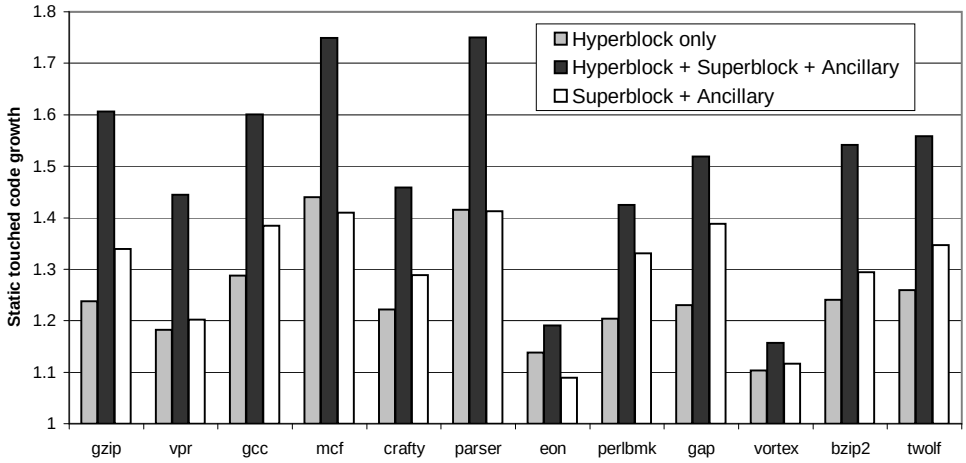


Figure 3.10 Touched static code growth with CFS techniques.

the non-predicated (**S-CS**) framework. The details behind the generation of all this additional code, and the controls over it in the various individual cases, will be discussed in later chapters, as appropriate. The main point here is that a great deal of code expansion is applied to create opportunities for specialization. One might expect this degree of code expansion to have universally negative instruction cache effects. As will soon be demonstrated, however, this is not the case. Although CFS transformations do not directly target instruction cache performance as an optimization objective, they tend either to be neutral with respect to it, or actually to improve it in some cases.

Figure 3.11 shows the number of first-level instruction cache misses as a fraction of the number of instruction cache accesses in the **O-NS** code.[11] Even with the extensive code replication performed in IMPACT-compiled code (recall that even the **O-NS** configuration reflects approximately a 1.40× growth in touched code size due to inlining), the SPEC CINT2000 benchmark suite places little stress on the instruction cache hierarchy of the Itanium 2 processor. A variety of behaviors can be observed. In five benchmarks, instruction cache performance is so good as to be uninteresting. In *gcc*, *eon*, *perlbmk*, and *gap*, the number of misses is only barely affected by CFS transformation. In

[11]For the **O-NS** configuration, this is the L1I miss rate; for other configurations, this represents a normalization of the number of misses to render it comparable to the number for the **O-NS** configuration. Since the numbers of useful instructions and lines fetched vary across the various configurations, comparing the miss rates across all the configurations is not as meaningful. The same general trends, however, are observed in raw miss rates.

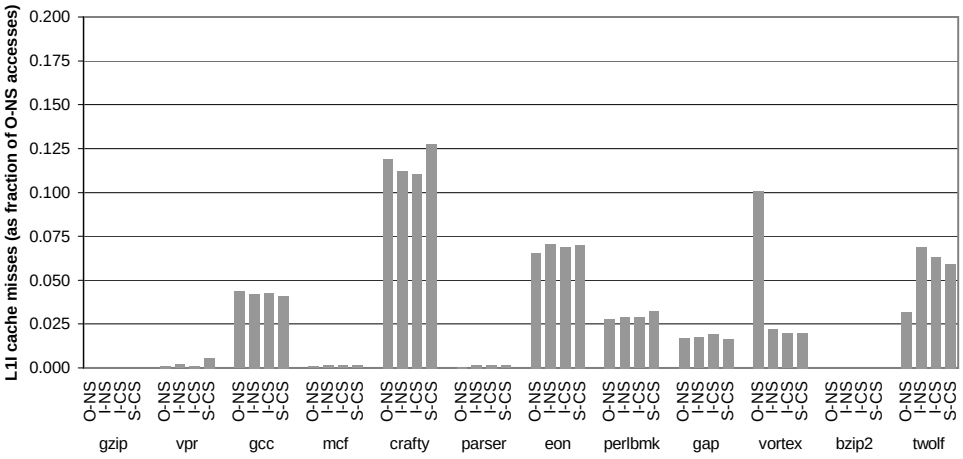


Figure 3.11 First-level instruction cache misses, relative to **O-NS** accesses.

crafty, the number of misses is decreased by predicated instances of CFS and increased by the Superblock-only **S-CS** approach. In *twolf*, all transformations severely impact instruction cache performance, and, finally, in *vortex*, instruction cache performance is dramatically improved by CFS transformation. That being said, the instruction cache is not generally a primary limiter of performance in most SPEC CINT2000 benchmarks on Itanium 2; the instruction translation lookaside hierarchy is so effective as to render changes in its load totally insignificant.

Considering Figures 3.10 and 3.11 together, some paradoxical comparisons emerge. Three benchmarks with noticeable instruction cache miss activity, *gcc*, *eon*, *perlbmk*, and *gap*, incurred pronounced touched, static code expansion in the **I-CS** and **S-CS** configurations, but did not incur a commensurate increase in instruction cache pressure. *Vortex*, though it increases modestly in size, incurs a dramatic reduction in cache misses. *Crafty* in the **I-CS** configuration behaves likewise, but less dramatically so. Only *twolf* (and, to a lesser extent, the **S-CS** configuration of *crafty*) displays the expected behavior, in which increased replication noticeably impacts the occurrence of instruction cache misses. This is indeed good news for instruction-replicating transformations in SPEC CINT2000.[12]

[12]Conversely, though, it means that the SPEC CINT2000 benchmarks may not be a good set for measuring the effect of new techniques for controlling the degree of code replication in CFS transformations.

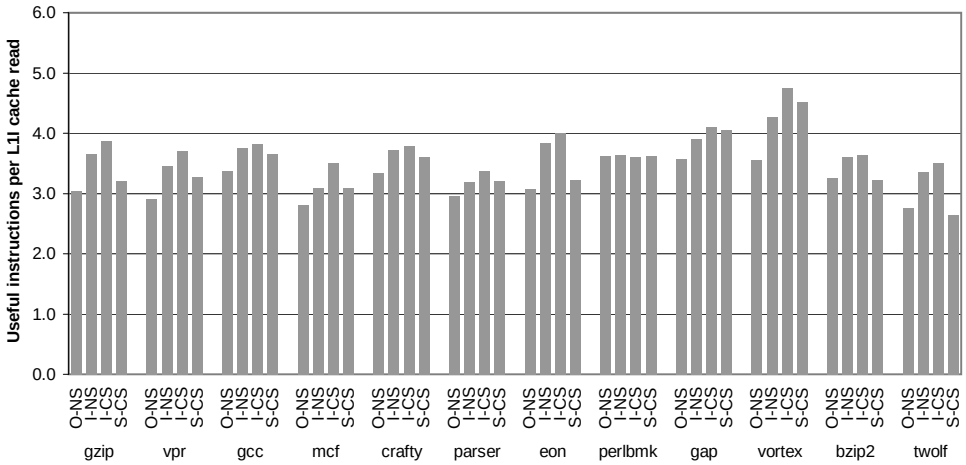


Figure 3.12 Instruction fetch efficiency with CFS transformation.

Several factors offset the expected, negative effects of code replication on instruction cache performance. The foremost of these is that CFS transformations increase the general efficiency of the instruction fetch process in the Itanium architecture; that is, CFS causes more useful instructions to be delivered to the execution core per instruction cache access. Figure 3.12 shows the number of useful instructions obtained per L1I cache read for the various compiler configurations. The theoretical maximum is 6, the number of instructions that fit in an L1I cache line. As indicated by the difference between the height of bars for the **O-NS** and **I-CS** configurations, CFS transformation often allows more useful instructions to be packed into each cache line fetched. Although CFS potentially creates several specialized versions, increasing overall code size, the versions created are more compact, potentially even improving instruction cache performance. This is a result of better instruction packing (more available ILP means fewer explicit `nop` instructions in the program) and improved instruction layout (more straight-line code means fewer fetch redirections and hence fewer partially-used lines). The benchmark *vortex* receives special treatment in Section 4.3.

Figure 3.13 shows one tangible effect of this compaction—CFS typically reduces the number of instruction cache accesses by 18%. As we shall see shortly, this sometimes contributes to the net performance benefit of CFS transformation, and is a larger than expected offsetting factor for the potentially grave performance cost of widespread code replication.

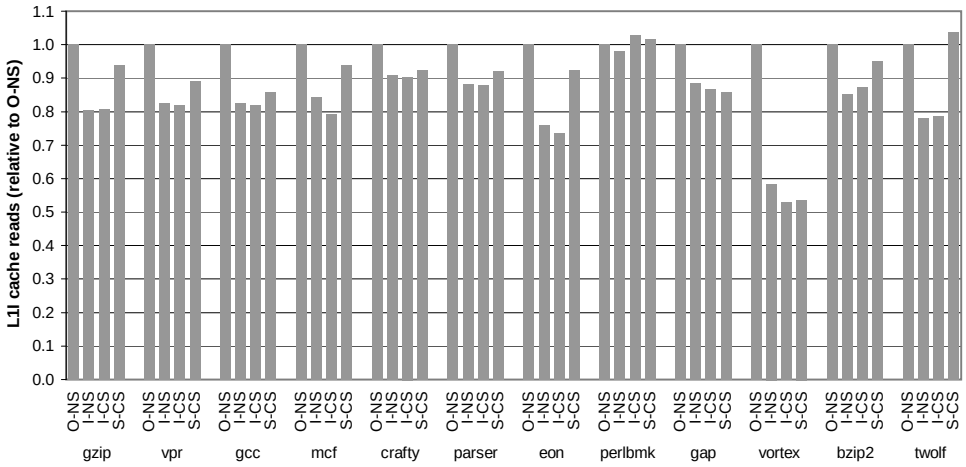


Figure 3.13 Instruction cache accesses.

Code expansion due to EPIC compilation techniques is tolerable if it causes no instruction cache footprint in the program to be spread beyond the capacity of the enclosing cache. Code replicating transformations that condense "hot" segments by ejecting "cold" copies (e.g., by excluding never-visited paths from a Hyperblock, creating zero-weight tails), therefore, generally improve performance, since the cold copies only infrequently enter the cache. Replicating transformations that generate volumes of "lukewarm" code (code that is traversed with some frequency), however, can cause instruction cache thrashing when these copies compete with each other and with other nominally resident code in their enclosing footprint. This can offset or reverse any gains from specialization, parallelism, or misprediction elimination. This is the case with *twolf*, as will be described, together with the finer points of **I-CS** instruction cache performance, in Section 5.7.2. In the aggregate, first-level instruction cache misses are reduced 5% by **I-CS** transformation and 1% by **S-CS** transformation, both relative to **O-NS**.

Given this perspective on the various components of this problem, let us return to consideration of the simplified Figure 3.9. Generally, the combination of continuing (though diminishing) returns in decreased instruction execution time places the point of optimum performance somewhere beyond the point at which instruction fetch performance has begun to be degraded due to increased footprint. Since the processes that perform the specialization and those that perform the subsequent optimization and scheduling occur at diverse points throughout the compiler, the control of this process for optimum

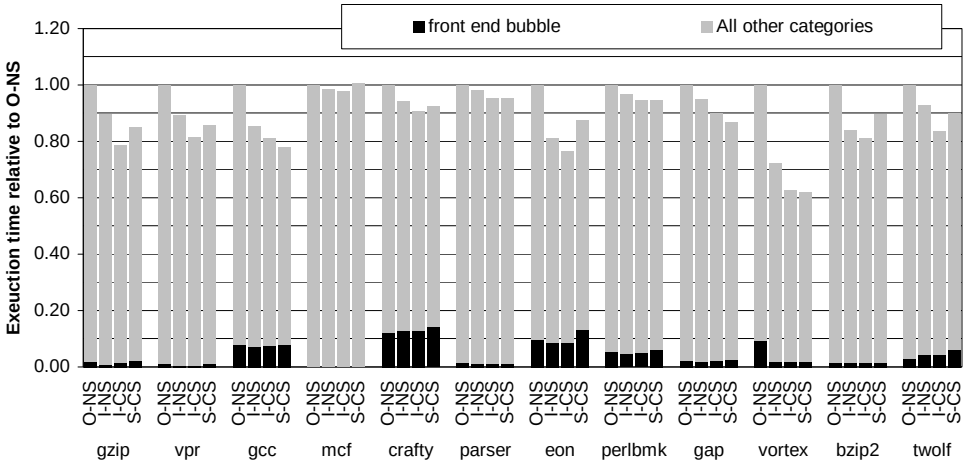


Figure 3.14 Contribution of front end stall to performance.

performance is a difficult problem. A fundamental understanding of the interaction of the mechanisms underlying these performance-producing processes is a first step on the path to better control of CFS compilation.

Some initial steps were taken in this dissertation work to rein in the more egregious examples of excessive code replication. Two examples include a re-engineering of loop peeling, to be discussed in Chapter 5, that replicates much less code in many cases, and a priority-driven, bounded branch target expansion module that limits the degree to which this ancillary transformation can explode code size. Prior to these techniques, instruction cache performance was often (needlessly) injured by out-of-control code replication processes. *Crafty*, for example, in **S-CS** mode, incurred a 2.65× code growth ratio (compared to 1.3×, as indicated in Figure 3.10). This *doubled* the number of front end stalls and caused **S-CS** to render a net loss of performance relative to **O-NS**. With these modifications, **S-CS** has been rendered a reasonable transformation model, and the **I-CS** model has been stabilized.

To conclude this topic, Figure 3.14 shows the contribution of front end stalls to total benchmark execution time. (This graph reflects the same data as Figure 2.6, but has been reformatted to emphasize the front end stall component.) Front end stall cycles, as reported here, generally are dominated by instruction cache miss latency.[13]

[13]In *gzip*, *vpr*, *parser*, and *bzip2*, front end stalls are dominated instead by branch retirement queue and branch resteer stalls (but are also insignificant as a component of total performance).

Given the improvements due to the relatively straightforward changes just described, the performance effect of instruction cache is generally small and relatively insensitive to CFS transformation. The **I-CS** configuration, though it actually involves more code size growth due to lax regulation of code expansion in Hyperblock formation, actually generally performs slightly better than **S-CS** due to the fact that, since predication allows the co-optimization of compatible paths, it spreads execution weight across fewer code versions than does a Superblock-based approach.

3.6 When Compile-Time Predictions Go Awry

Given the heavy bias of CFS transformations towards the anticipated-common paths (and the utter reliance of EPIC system performance on these transformations), the problem of run-time deviance from the biases exploited in the compiler is of particular concern to EPIC compiler developers.[14] Profile variation is a significant real-world problem for any application in which it is difficult to anticipate use-time execution biases from a "representative profile." SPEC attempts to model this problem slightly by providing "training" and "reference" inputs. Although these inputs are generally quite similar in terms of their execution biases, a few notable variations occur.

Measuring the resiliency of CFS transformations to use-time variation in execution bias (as well as variations in data cache latency brought on by changes in data set size) is a difficult-to-define problem. The practical effect of these variations, however, can be assessed in a limited way by compiling benchmarks using their reference inputs for training, and then measuring their performance on the same inputs. Such an experiment resulted in substantial changes in performance for only two benchmarks: *perlbmk*, the performance of which improved by 15%, and *crafty*, for which the improvement was 7%. The pronounced effect of input variations on these programs is not difficult to anticipate. The exercise of *perlbmk*, as an interpreter, will vary widely, depending on the script used as input. (*Perlbmk* is also very sensitive to accuracy of profile information after inlining;

[14] EPIC compiler writers will identify with the words of the great Scottish poet, Robert Burns: "But Mousie, thou are no thy-lane, In proving foresight may be vain: The best laid schemes o' Mice an' Men, Gang aft agley, An' lea'e us nought but grief an' pain, For promis'd joy!"

IMPACT currently uses two profiling runs to address this. See Section 7.4.5.) Likewise, the biases of *crafty*, a chess program, will depend on the style and progression of the game being played.

Even programs that could be assumed to have highly anticipable behavior, such as compressors and decompressors, may have subtle susceptibilities to profile variation. This behavior does not seem to be prevalent in SPEC CINT2000. SPEC CINT95, on the other hand, contained a prominent case in the benchmark *compress*. This benchmark relied on an internal-chaining hash table. In the training input, the table is relatively empty, so collisions are infrequent. Peeling a single iteration of the loop body is therefore sufficient to keep execution within a specialized Hyperblock on well over 90% of loop iterations. In the reference input, however, the table is relatively full, so the inner loop checking the internal chains frequently iterates many times. The specialized Hyperblock created with reference to the training input is no longer very profitable.

Historically, these distinctions (such as profiling with a meaningfully different training input than that used to evaluate execution time) have been ignored in most previous EPIC work, which often used the same inputs for both training and reference, and even often used further-shortened input sets to reduce simulation time. This can contribute a host of inaccuracies to modeling, if the goal is to predict SPEC performance of compiler or microarchitectural techniques. In keeping with the real-world flavor of this dissertation, they have been used appropriately in this work. This dissertation points out in bold relief the degree to which successful EPIC transformation relies on execution bias, even in common benchmarks. Profile variation is thus a very serious problem for EPIC compiler writers, but is unfortunately one that a study of only SPEC CINT2000 has limited ability to illuminate.

3.7 Focusing on Particular CFS Components

The following three chapters, "Chapter 4: Code specialization in the Superblock," "Chapter 5: The value and application of predication," and "Chapter 6: The value and application of speculation," present a detailed explanation of the methods and effects of the elements of the CFS outlined in the foregoing overview. The order of the following

chapters is designed with pedagogical intent, and does not necessarily reflect the phase ordering or physical dependence among the described techniques (for such a view, see Appendix B). The chapters build on each other as follows: The formation of optimization and scheduling regions enables the exposure of ILP through CFS compilation. Chapter 4 focuses on procedure inlining and Superblock formation, two fundamental specialization techniques. Chapter 5 explains the use of predication in Hyperblock formation as an extension of the Superblock techniques (and, incidentally, in other optimizations and techniques). Finally, Chapter 6 presents control speculation as a technique that mitigates the effect of control dependence within Superblock and Hyperblock regions. The presentation of methods in these chapters is interspersed with examples from the SPEC CINT2000 benchmarks.

4 CODE SPECIALIZATION IN THE SUPERBLOCK

One of the important innovations of the control-flow-structural model (the historical development of which is described in Section 10.2) is the specialization of common code paths for aggressive optimization. Specialization occurs in two primary ways: through procedure inlining and through Superblock / Hyperblock formation. Inlining and Hyperblock techniques will be treated in their own chapters (Chapter 7 and Chapter 5, respectively). This chapter will focus on Superblock formation and its ancillary transformations and their behavior in the experimental context. As will be seen, Hyperblock formation is most easily understood as an extension of the Superblock approach.

Earlier presentations have not always distinguished between the performance of the Superblock (single-trace) and Hyperblock (predicated) models of CFS-optimized code, or between the effects of the basic Superblock algorithm and those of subsequent or ancillary transformations. The author's earlier work, for example, showed only the performance of CFS-optimized code relative to basic block code, lumping the effects of Superblock-style and Hyperblock-style optimization [52]. The work of this dissertation showed that, for a subset of the SPEC CINT2000 codes, Hyperblock formation has little to add to the performance of the older Superblock techniques. Furthermore, this set of benchmarks that are relatively unresponsive to if-conversion is a much more substantial subset than might have been expected from [38] and other previous work. This chapter explains this result.

One important facet in understanding the performance of Superblock (and, to some extent, Hyperblock) optimized code is the effect of specialization on the effectiveness of subsequent optimizations. Superblock formation does not simply create an opportunity for improved scheduling, as trace and wavefront scheduling do. It creates specialized code traces which then may be optimized individually, potentially to great benefit. This effect plays an important role in some of the results presented, most notably in the dramatic speedup achieved in *vortex*. It should be noted, however, that the contribution of these effects is diminished relative to that observed in older works [79].

4.1 Superblock Formation

The IMPACT compiler's Superblock formation algorithm remains essentially as described in [22]. It is summarized in the pseudocode of Figure 4.1. Mutually exclusive traces through the control flow graph are first selected. Then, they are ranked by execution frequency and greedily converted into Superblocks until either the specified code expansion ratio is exhausted or no further opportunities exist. There are several factors involved in selecting valid trace successors and predecessors. The two most influential of these are the predecessor and successor execution ratios. The predecessor ratio[1] specifies the minimum fraction of execution frequency of a block that must be directed to the trace for it to be prepended to the trace. The successor ratio[2] is the symmetric condition for forward trace growth. The settings of these parameters, within reasonable bounds, were not found to have a substantial effect on total code performance (largely due to a subsequent transformation, branch target expansion, which we will shortly describe). Code structure, these two factors, and the maximum tail duplication code growth budget (1.50 in the experiments presented here) define the degree to which Superblock traces will be constructed.

The tail duplication cost of Superblock generation is strictly limited to a factor of two, as each block may participate in only a single Superblock trace and, therefore, may be tail duplicated at most once. In practice, the code expansion ratio is much lower, as

[1] the `Lblock` parameter `trace_min_cb_ratio`, here set to 0.50
[2] the `Lblock` parameter `min_branch_ratio`, here set to 0.70

```
1: procedure FORM-SUPERBLOCKS(G = {B, E}, expansion-factor)
                                        ▷ G = {B, E} = procedure control flow graph
2:     T ← {}                                   ▷ Set of Superblock candidate traces
3:     size ← PROCEDURE-CODE-SIZE(G)
4:     max-size ← size × expansion-factor
5:     for all b ∈ B do
6:         visited[b] ← 0
7:     end for
8:     while b ← GET-HIGHEST-WEIGHT-UNVISITED-BLOCK(B) do
9:         T ← T ∪ GROW-SUPERBLOCK-TRACE(b)
10:    end while
              ▷ Manufacture Superblocks greedily as tail duplication budget permits
11:    while t ← GET-HIGHEST-WEIGHT-TRACE(T) do
12:        cost ← TAIL-DUPLICATION-COST(t)
13:        if size + cost ≤ max-size then
14:            CONSTRUCT-SUPERBLOCK(t)
15:            size ← size + cost
16:        end if
17:        T ← T − t
18:    end while
19: end procedure
20: procedure GROW-SUPERBLOCK-TRACE(b)          ▷ Grow from seed block b
21:    trace ← (b)
22:    head ← tail ← b
23:    while next ← BEST-VALID-TRACE-PREDECESSOR(head) do
24:        trace ← PREPEND(trace, next)
25:        visited[next] ← 1
26:        head ← next
27:    end while
28:    while next ← BEST-VALID-TRACE-SUCCESSOR(tail) do
29:        trace ← APPEND(trace, next)
30:        visited[next] ← 1
31:        tail ← next
32:    end whilereturn trace
33: end procedure
```

Figure 4.1 Superblock formation algorithm.

determined by the expansion parameters and code characteristics. In the experiments reported here, the ratio was generally less than 1.10 and, on average, only 1.06. As will be seen in Section 4.2.1, this simple transformation alone (without branch target expansion) often garners most of the benefits of the *S-CS* framework. Where this is the case, execution through important code segments is strongly dominated by single, prominent execution paths, which easily turn into successful Superblocks. This is consistent with the goal of Superblock formation—selecting these heavily biased, primary paths for subsequent specialization.

Other benchmarks benefit from more extensive specialization through a subsequent transformation known as branch target expansion. And, as we will see in the next chapter, still others require the multipath optimization available only with the use of predication. For an example of how this can take place, consider the control flow graph of Figure 4.2(a). Shown is the control flow graph of a loop that constitutes approximately 5% of *vpr*'s execution time. Except for the loop header and footer, the blocks are colored white and light gray. The two colors indicate that two distinct data structures are being manipulated, the gray blocks manipulating structure `x` and the white blocks manipulating structure `y`. The control flow and computation of the two sets of blocks are entirely independent. (This will become of chief importance in the next chapter.) Control flow frequencies are indicated by three styles of control flow arc. The heavy, solid lines are most frequent, followed by the heavy, dashed lines (at approximately half the solid lines' weight), followed by the light solid lines, which are relatively insignificant. The loop is dominated by the unbroken heavy path marked as the dominant path in the figure (slightly more than half of loop iterations progress along it), but the marked secondary paths are, taken together, equally significant.

We consider the Superblock framework's approach to this loop. The Superblock former can select a single path to exploit through this loop. It, of course, selects the path marked as dominant. Figure 4.2(b) shows the Superblock formed, together with the tail duplicated region making up the remainder of the loop. While the Superblock succeeds in capturing about half the control flow through the loop, there are some problems with this approach. Where secondary paths depart from the course of the main path (indicated by

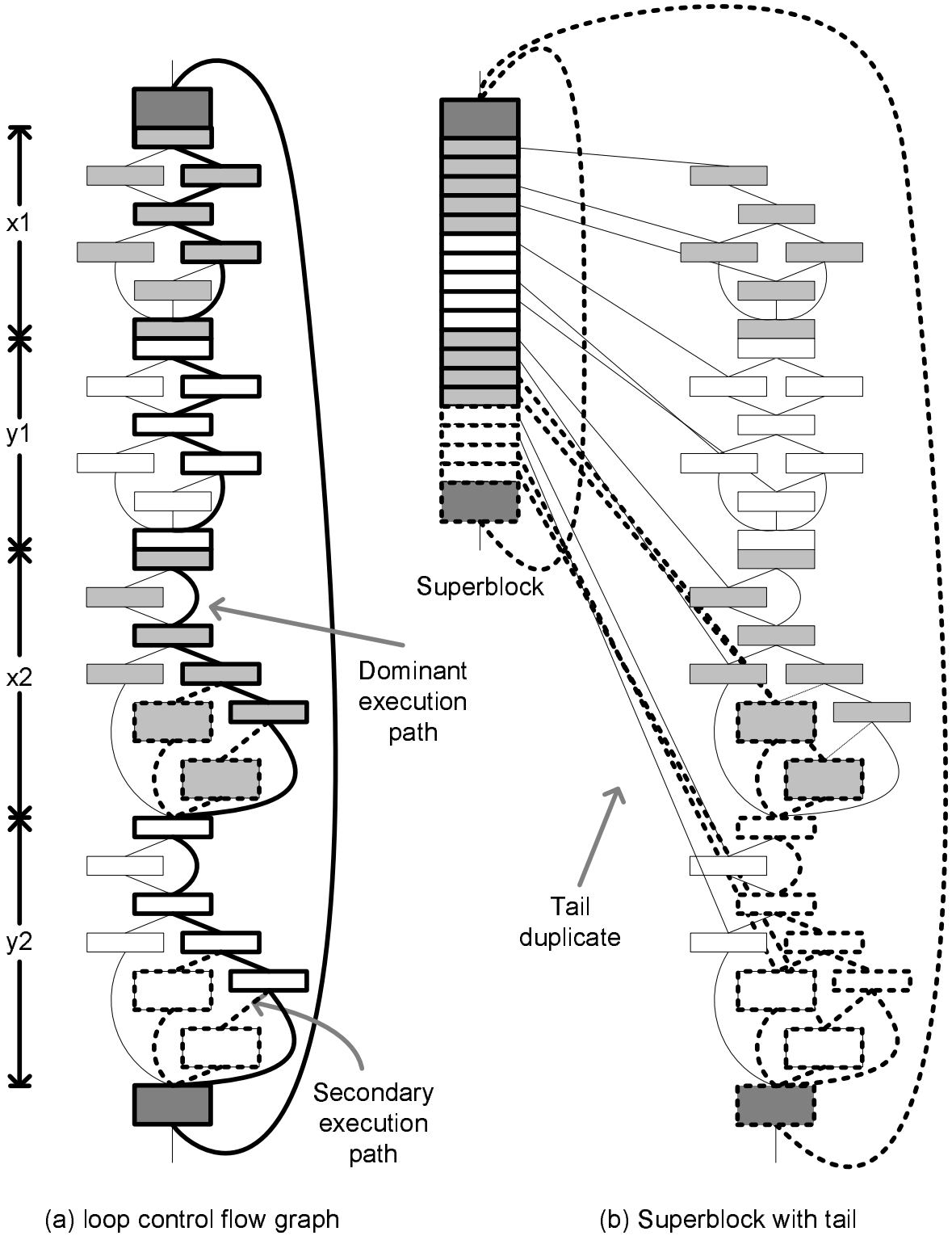


Figure 4.2 Superblock formation in *vpr* `get_bb_from_scratch()`.

the heavy, dashed lines), control flow weight is "lost" to the tail duplicated region. Since the Superblock is single-entry, this flow cannot re-enter the specialized trace. This renders the Superblock loop not a good candidate for software pipelining, since it is unlikely to retain execution for more than a single iteration. Furthermore, any operations speculated above one of the frequently taken exit points perform extra work that is wasted, even if these operations were from a point in the control flow graph where the excluded, secondary path would have already rejoined the main path. Finally, the code in the tail is left unimproved with respect to ILP opportunities. As was just mentioned, branch target expansion will seek to improve on the last problem, but the remainder of the issues cannot be addressed generally without the multiple-path specialization of Hyperblocks.

4.2 Effect of Ancillary Transformations

The effect of ancillary transformations, in particular branch target expansion, in delivering performance in the Superblock framework is of some importance. An understanding of **S-CS** performance is not complete without a description of at least a few of these techniques.

4.2.1 Branch target expansion

As we have seen in the example, branch target expansion allows ILP transformation of code left out of the program's original Superblocks, a concession to the fact that code often has more than one important path. It also allows replication and concatenation of existing Superblocks to create even larger exploitation regions. In branch target expansion, the code in a target block at the end of a control flow arc emanating from the end of a (source) block is expanded into the source block. If the target block had multiple predecessors, this entails code expansion. The goal of branch target expansion is to create larger regions for subsequent optimization. It goes beyond (performs transformations beyond what would be achieved in) the Superblock algorithm, in that it may allow *multiple* replications of a single piece of code (in the Superblock scheme, code may only be tail-duplicated once).

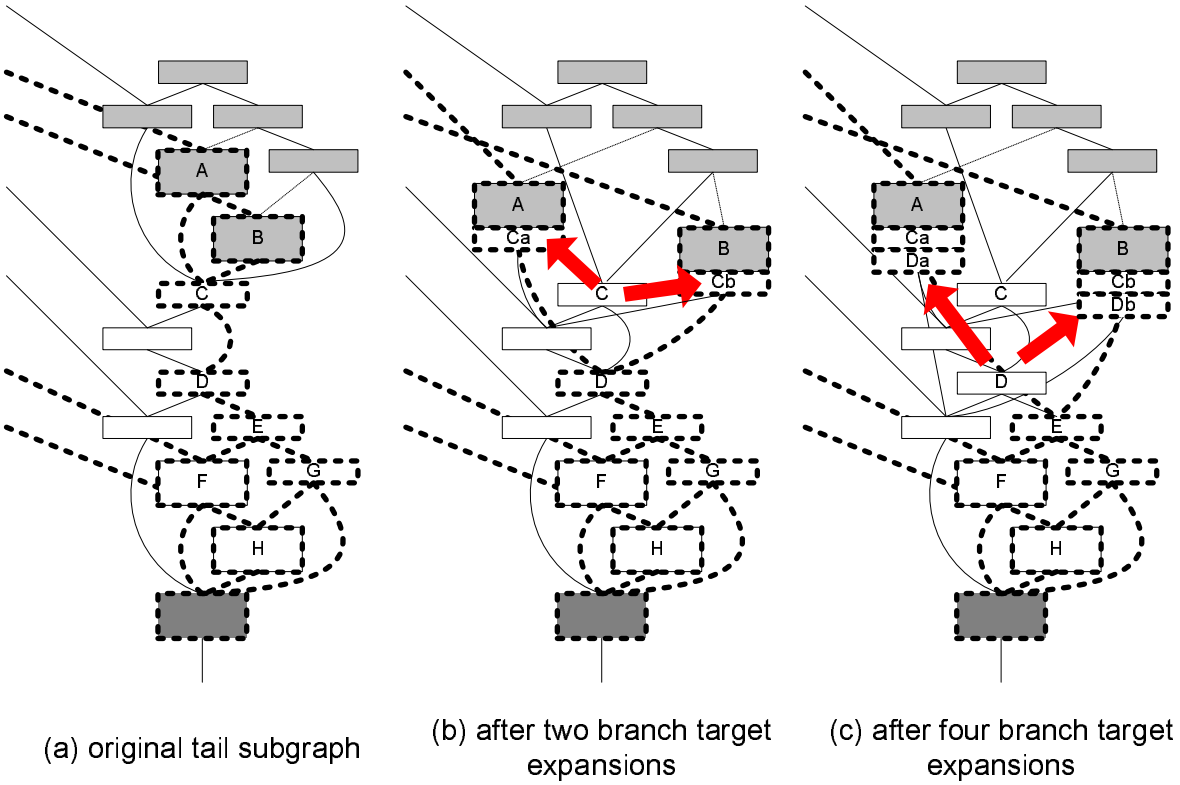


Figure 4.3 Branch target expansion in tails in *vpr* `get_bb_from_scratch()`.

Figure 4.3 shows the early activity of branch target expansion in the code from the previous example. Figure 4.3(a) shows the tail portion of the control flow graph that resulted from the formation of the dominant loop Superblock. Blocks A through H are marked for clarity. Arc weights are as indicated before. Figure 4.3(b) shows the effect of two branch target expansions, of the control flow arcs A → C and B → C. Here we note that block C is replicated into each source block, extending the number of operations among which ILP can be sought in the two extended basic blocks. This also, however, makes for a total of *four* copies of C (one in the original Superblock). Three of these copies are "lukewarm"; that is, all three are occasionally executed; one of the copies (labeled C) is cold; this is of less concern. Figure 4.3(c) shows a subsequent two instances of branch target expansion. This process can be continued indefinitely, potentially causing an exponential increase in code size, even in lukewarm code. Lukewarm code expansion is one of the primary problems with the Superblock framework. Over-expansion of lukewarm code can substantially increase the active instruction footprint, causing increased misses

and degrading performance. This effect was visible in the increased front end stalls noted in Figure 3.14 for *crafty*, *eon*, and *twolf* in the previous chapter.

Before the work of this dissertation, branch target expansion was applied with only the following important restrictions:[3] (1) it explored candidate arcs in six linear passes through the control flow graph in the course of CFS optimization. This potentially limited the growth of a given trace, no matter how heavyweight it was or how likely it was to complete, and allowed essentially all blocks an equal opportunity to expand (up to a maximum size of 256 instructions). No limit was imposed on code growth from branch target expansion. (2) Expansion was not permitted through Hyperblocks (due to inadequacies in an old predicate analysis mechanism).

The combination of these effects resulted in varied results. Increased Hyperblock formation, even of benign Hyperblocks, could prevent important branch target expansion later. Variations in Superblock parameters could, likewise, indirectly affect opportunities for branch target expansion. The most serious problem was the opportunity for explosive code growth in applications with complex control flow. An example of this behavior occurred in *crafty*, in which explosive code growth deriving from essentially unconstrained branch target expansion degraded Superblock results by nearly 10%, actually delivering a result worse than that for classically optimized code. The frequent Hyperblocks created in *crafty* artificially curtailed this growth, so the **I-CS** configuration did not suffer from this problem to the same extent. This resulted in an "unrealistically good" showing of Hyperblock code relative to Superblock in this application. Opposite results occurred elsewhere, where profitable expansion had been blocked by Hyperblocks. (This misadventure underscores the importance of thoroughly investigating differences in performance to determine their root causes.)

To rectify this situation, branch target expansion was reimplemented to eliminate the more arbitrary restrictions (per-block transformation count and prohibition of Hyperblock expansion) and to install a prioritized expansion of the most important edges, up

[3]Other restrictions, including branch and block frequency and probability limits were applied, but these often played little role in limiting code growth. Similar limits are applied in the new approach, as well.

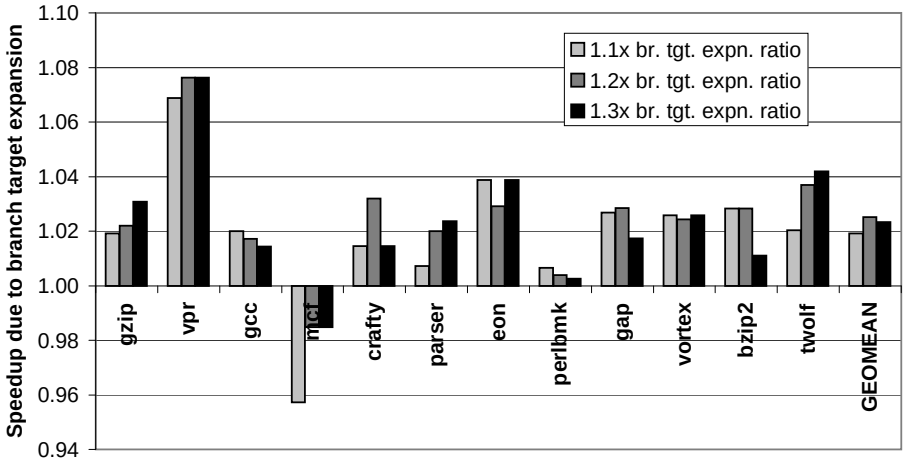


Figure 4.4 Effect of branch target expansion on **S-CS** performance.

to a fixed code expansion limit. Until this code expansion limit is reached,[4] dominant, block-ending control flow arcs are expanded in priority order, where the priority function is $\textit{flow-weight}/\sqrt{size_{source} + size_{target}}$. This experimentally determined priority function tends to expand those branches that are more likely to be executed, while also preferring to equalize the size of regions to a certain degree (rather than creating only a few, very large regions, and leaving many, very small but still relatively important blocks).

Figure 4.4 shows the effect (on **S-CS** performance) of varying the expansion ratio between 1.00 (no expansion) and 1.30 (aggressive expansion). A branch target expansion ratio of 1.10 delivered necessary gains for **I-CS** code without unduly admitting penalties; a higher ratio of 1.20 was necessary to deliver peak performance in **S-CS** code, due to the more stringent limitations on specialization in the original Superblock construction phase. These ratios are applied in the **I-CS** and **S-CS** results reported here. The data of Figure 4.4 provide a concrete example of the behavior described using the abstracted Figure 3.9. The front end stall results presented previously in Figure 3.14 revealed that only *gcc*, *crafty*, *eon*, and *twolf* exhibited significant instruction cache miss behavior. In *gcc* and *crafty*, further increasing specialization (beyond the 1.20 mark) increases harmful footprint effects more than it improves performance in other respects. In *eon* and *twolf*, however, increased specialization purchases continued (although slight) improvements,

[4]Expansion of arcs that lead to single-predecessor blocks is not counted against the limit, since such transformations do not actually replicate code.

although to do so does increase the instruction cache penalty component of execution time. These constitute nice examples of the behavior described in Figure 3.9 in the previous chapter—with CFS transformation, total performance may continue to increase beyond the point where the instruction cache begins to be seriously penalized.

4.2.2 Loop unrolling and/or software pipelining

Loop unrolling, the replication of a loop body inside a loop, has long been associated with Superblock techniques [22]. Unrolling has three benefits: it reduces the frequency of likely-taken control flow (or potentially eliminates some branches in certain counted-loop situations); it provides for overlap in scheduling across a given number of loop iterations; and, finally, it allows for cross-iteration optimizations, such as induction variable expansion, which can reduce effective dependence height. Loop unrolling does, however, incur a degree of code expansion and cannot by itself achieve steady-state overlap between successive loop iterations in the manner of software pipelining (it only approaches this limit). Software pipelining, on the other hand, does not admit the possibility of cross-iteration optimizations, since it does not, in the rotating register schema employed on Itanium 2, actually replicate the body of the loop. For this reason, the performance of an unrolling-based approach was compared (in the **I-CS** configuration) to that of a modulo-scheduled approach that used unrolling only for loops that were not modulo-schedulable, and then only to a small degree. This study found that no benchmark performed better in the unrolling-based approach than in the modulo scheduling approach, indicating that the improved overlap and code size efficiency of the modulo scheduling approach outweighed the effectiveness of unrolling-based optimization in delivering net performance.[5]

4.2.3 Branch combining

In previous work, a transformation known as branch combining has proved effective. This transformation collects adjacent side exit branches in a Superblock (or Hyperblock) trace for combination into a single summary branch, whose condition is computed with

[5]In Chapter 5 an application of unrolling to software-pipelined loops will be described. This achieves a significant speedup in specific situations, but not because of traditional unrolling-based optimizations.

a predicate definition network constructed from the branches to be summarized. Speculable instructions among the branches are pushed above the summary branch; other operations (such as stores) are either guarded with a predicate reflective of their control dependence position among the branches or are pushed down below the summary branch. The summary branch leads to a "decoding" block containing the original branches and any extruded, nonspeculable operations. The benefit of branch combining is that it reduces the number of branches executed and can allow a degree of height reduction if the predicate network can be compressed. This technique is described in [79].

Passing experimentation with this technique found it to be only occasionally effective, and then only slightly. In *perlbmk* it degraded performance by 5%; other benchmarks had smaller, varied gains and losses. Given the variability of these results, and the fact that it requires predication, branch combining was applied only in the **I-CS** configuration. Those instances where branch combining, and not if-conversion, is primarily responsible for the **I-CS** configuration's performance gain (i.e., *eon*) will be pointed out as they arise.

It should be noted that Itanium's ability to retire three branches in a cycle reduces the appeal of this technique. Furthermore, branch combining uses parallel (*or*- and *and*-type) predicate definitions, which are not available for all comparison operations in the Itanium architecture (they support only equality, inequality, and comparison to zero) [80]. When a parallel compare must be generated for another comparison operation, more than a single instruction is required. This considerably increases the overhead of the branch combining technique.

Critical path reduction [68], a later generalization and systematization of branch combining, was not implemented but seems worthy of future investigation in light of these results.

4.2.4 Postformation instruction transformations

In IMPACT, the `Lsuperscalar` module attempts at length to perform specialization-enabled optimizations after region formation. In previous works, these optimizations have contributed significantly to final code performance. In these experiments, however, their role has been dramatically reduced. Omission of this phase of optimizations reduces

performance by 11% in *vortex* and by 2-3% in *parser*, *bzip2*, and *vpr*. The profound effect of post-formation optimization in *vortex* is due to the optimization of many store-to-load forwarding opportunities (and subsequent, enabled optimizations) in the most critical loops of the application. Exclusion of paths with aliasing stores permits many more applications of this transformation in key Superblock loops.[6]

4.3 Evaluation

One exceptional case deserves special note. The benchmark *vortex* shows spectacular benefit from CFS transformation. This benefit reflects the formation of very large, stable, single-path execution paths which provide effective coverage of all common benchmark execution patterns. The specialization inherent to formation of these regions provides opportunities for removal of loads and stores (due to elimination of aliased accesses), branches (due to control redundancy), and other operations (hence the importance of post-specialization optimization to *vortex* mentioned previously). The number of useful operations executed is reduced by nearly a third. The total number of operations executed is decreased by 40%, owing to a reduction in `nop`s due to improved scheduling in Superblock regions. This results in a substantial reduction in front end stalls (as indicated in Figure 3.14) in addition to a large, overall performance benefit. In the most prominent case, the dominant loop of the procedure `SaFindIn()` and all its common callees (which accounts for 25% of *vortex*'s execution time) are optimized together to produce a Superblock loop. Within this loop, substantial specialization is performed. The execution time of this function is reduced by over 60% relative to the **O-NS** code version. As will be discussed in Chapter 7, substantial inlining is required to enable this effect in *vortex*. A substantial investment must be made to enable reaping the benefits of specialization.

[6] An adaptation of partial redundancy elimination could arguably perform these optimizations apart from the formation of regions, performing the on-trace store-to-load forwarding and adding compensation code to the off-trace paths.

Superblock-based optimization **S-CS** transformation, without predication, achieves an average performance increase of 1.15 relative to a classically optimized baseline (**O-NS**). (A limited degree of Superblock and branch target expansion subsequent to Hyperblock formation also measurably aids the performance of **I-CS** code.) In *gcc*, *parser*, *perlbmk*, *gap*, and *vortex*, the **S-CS** configuration provides results approximately the same as those achieved when the **I-CS** configuration is applied. Superblock performance relies substantially on some subsequent, ancillary transformations, such as branch target expansion, that create more versions of code than the Superblock algorithm itself would generate. These subsequent transformations must, however, be controlled to keep the code footprint penalties of this model from outstripping gains from increased planned instruction-level parallelism. Relatively straightforward controls are usually effective in preventing noticeably negative performance implications, as program structure tends not to lend itself to explosive lukewarm code growth. This is an encouraging result for CFS transformation. As we will see in the next chapter, multipath optimization, where program structure accommodates it, is better able to support ILP without deleterious cache effects.

5 THE VALUE AND APPLICATION OF PREDICATION

Explicitly parallel instruction computing systems provide support for *predication*, the addition of a Boolean operand to instructions that controls whether or not the instruction executes and commits its result. The Intel Itanium architecture provides a full predication model [81], in which the vast majority of instructions have a guard operand, which may be set to reference one of 64 architected, single-bit predicate registers.[1] Predicates are defined using dedicated predicate define instructions (with mnemonics based on `cmp`) and are fully bypassed to be available in the cycle after their computation.[2] The Itanium architecture's predicate defining instructions are modeled after those of the HPL-PD architecture [4] and are described in detail in [80].

Most would consider predication to be a topic secondary to speculation. Since predicated code is subject to control speculation (via promotion), since predication is an integral part of the control-structural approach to ILP formation foundational to this work, and since many of the interesting and important cases of speculation involve predication, this dissertation places predication first.

This chapter presents IMPACT's Hyperblock-based predication framework, as improved to support the class of programs used in this dissertation's experiments, and explains the performance results it provides. This dissertation improved peeling heuristics, improved heuristics to include relatively infrequent paths into Hyperblocks to reduce

[1] Predicate register `p0` has a constant value 1, or *true*.

[2] Floating-point compares take two cycles instead of one to execute. Branches may consume a predicate computed previously in their same instruction group.

lukewarm (and sometimes even cold) tail duplication, and added a new form of loop unrolling that increases the applicability of predication. Additional work directed by the author during this dissertation work in the IMPACT group by Shane Ryoo extended its use in general optimization. Still, predication is found to be useful to Itanium 2 performance in only a subset of benchmarks having particular types of control structure, as will be explained in this chapter. These benchmarks include *gzip*, *vpr*, *crafty*, *eon*, *bzip2*, and *twolf*, half the SPEC CINT2000 benchmarks. After a brief introduction to methods of predicated region formation employed in the IMPACT compiler, these results will be considered in detail.

5.1 This Work's Approach to Predication

IMPACT's approach to predication must be positioned relative to two bodies of work. First, the Hyperblock approach applies the idea of path selectivity (from the Superblock approach) to the procedure of if-conversion. Before Hyperblocks, if-conversion was applied to all paths of nested code diamond or hammock regions [76, 82]. This was useful only in situations where the paths were well-balanced in path length and compatible in instruction content—not a situation generally available in nonnumeric applications. Hyperblocks are somewhat *less structured* than the complete if-conversion approach, and therefore applicable in a broader range of programs. On the other hand, some compiler frameworks ordinarily use predication to guard operations percolated above branches, as an alternative to control speculation [50]. Hyperblocks are *more structured* than this *ad hoc* application of predication, and provide for a wider range of benefits. The percolation approach, for example, merely moves instructions relative to control flow; it does not eliminate branches. Some attempts at characterizing the performance of predication in real hardware contexts, most notably [39], have considered the minimally profitable (or even injurious) endpoints of this predication spectrum without examining the fertile middle ground in which IMPACT seeks to apply predication.

This dissertation work treats predication in a manner typical of other, previous IMPACT work, in that predication is used early and extensively in the compilation framework to implement both control flow restructuring and optimizations. This work advances

on previous IMPACT studies in three primary ways: it provides a concrete evaluation of predication in the presence of real-machine constraints; it reflects both generalization and stabilization of IMPACT's predication techniques to deliver more practical, more consistent, and more far-reaching applications of predicated execution, particularly in the presence of low-trip-count loops; and, finally, it explores the interactions of predication with other compiler phases, such as control flow profiling and procedure inlining, and with common programming styles.

As mentioned in the introduction to CFS transformation, predication has two primary, positive performance effects: first, the removal of branch and misprediction penalties and, second, the enabling of overlap between independent blocks of control. To these, this dissertation adds a third, minor benefit—an easing of instruction cache pressure (relative to similarly aggressive Superblock formation) due to the ability to perform multi-path specialization in the Hyperblock context. The IMPACT compiler strives to take advantage of all these aspects with aggressive transformation that does not specifically target only presumably hard-to-predict (relatively unbiased) branches.

5.2 Predication Benefit Example

Figure 5.1 shows an example to illustrate both the complexity of Hyperblock formation decision-making and the potential benefit of choosing the right region. The loop indicated in (a) is the same one employed in the example of the previous chapter (Figure 4.2). Recall that the desired goal of CFS optimization for this loop body was allowing the overlap between the distinct dependence chains of the regions manipulating "`x`" values and those manipulating "`y`" values. This parallelism having been exploited, it would also be desirable to software-pipeline the loop.[3]

In the Superblock version, only a single path through the loop could be turned into a monolithic scheduling region. The control structure of the loop, however, with prominent

[3]Incidentally, modulo scheduling of this loop reduces benefit relative to a nonpipelined Hyperblock approach (increasing loop execution time by 15% relative to the **I-NS** approach) due to the scheduling of two loads likely to be serviced in L2 from the same L2 cache bank in the same cycle. This problem does not detract from the example, but is illustrative of the complex, and often unfortunate, interplay of CFS transformation and dynamic events. See Appendix C.3.2 for details.

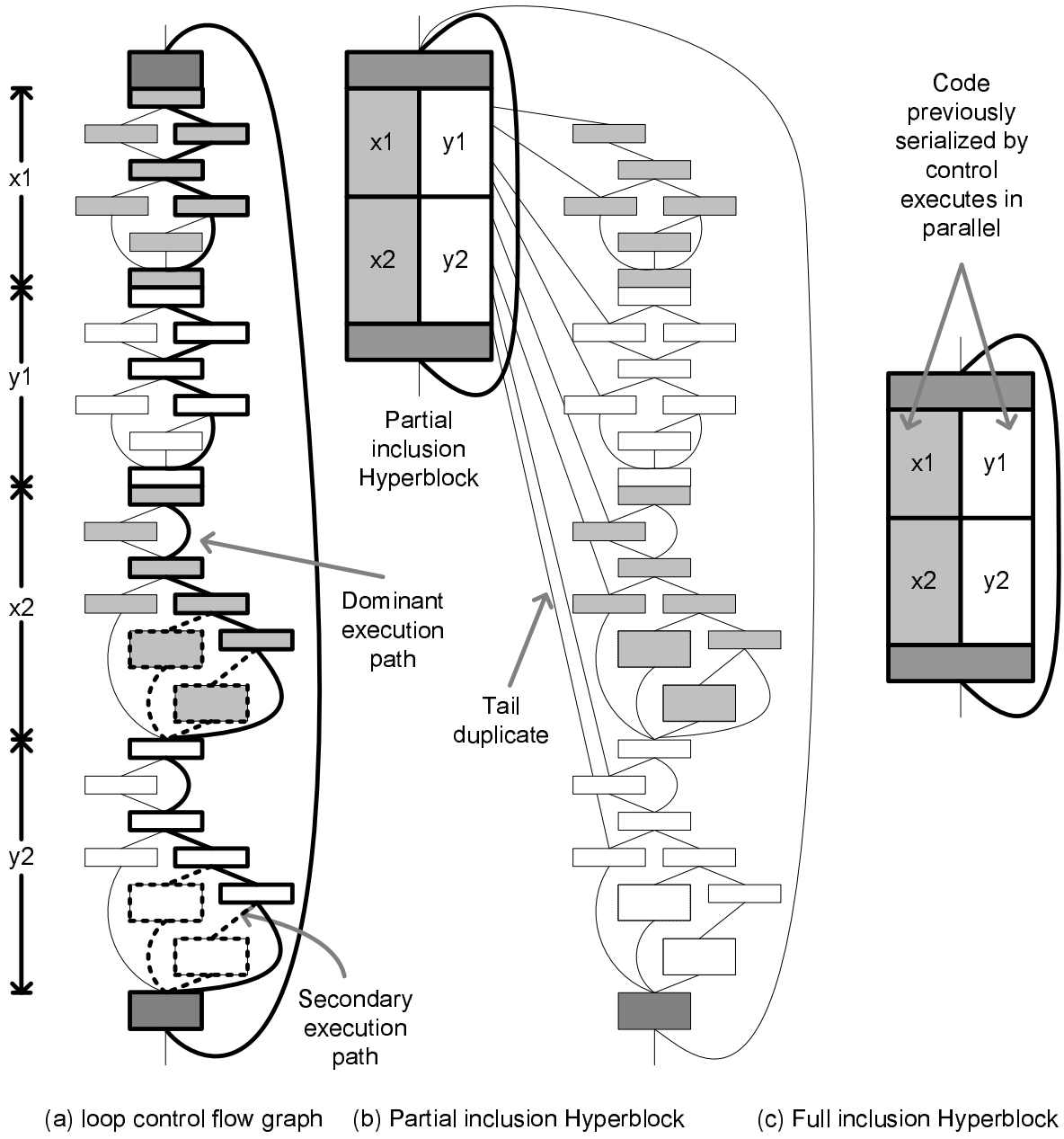


Figure 5.1 Hyperblock formation in *vpr* `get_bb_from_scratch()`.

secondary control paths, prevented the formation of a Superblock that captured even a simple majority of total loop iterations. Subsequent branch target expansion provided some opportunity for combination of blocks in the Superblock tail, but this came at the cost of extensive lukewarm code dilution and left potentially difficult-to-predict branches. Furthermore, single-path speculation from below control flow merge points could have caused work to be performed twice when the predicted, primary path was not taken.

Figure 5.1(b) shows the result of incorporating the primary and secondary paths into a partial inclusion Hyperblock region. In this version, all typically executed paths are incorporated into the Hyperblock. As in the Superblock, included instructions are able to interschedule as their dependences permit, but in this case, all common paths are covered, rendering speculation more efficient. A strategy that included common paths was the original heuristic conceived for Hyperblock formation [36]. Since uncommon paths are still excluded, however, a large amount of code is tail-duplicated. Fortunately, this code is infrequently traversed, so it is unlikely to significantly impact instruction cache performance. Furthermore, the decision and branching overhead necessary to divert execution to this region in the case that an unlikely path is traversed (with all these branches' associated dependences) remains. If this code is ever traversed with any frequency, performance may be negatively impacted.

Another option, shown in Figure 5.1(c), is to include some or even all uncommon paths to reduce the code replication overhead and to improve resiliency to changes in execution bias. In many cases (actually in all cases, in this example) low-frequency paths have negligible instruction count and dependence height impact on the common-path Hyperblock, and can be incorporated without negatively impacting performance. One of the key extensions of the Hyperblock heuristics in the work leading to this dissertation was the addition of mechanisms to include low-cost, infrequent paths into Hyperblock regions. This provides the stated benefits, but poses new opportunities for problematic behavior. Since control speculation of infrequent paths must generally be allowed to minimize their impact on dependence height, misspeculation of these only infrequently on-path instructions can have catastrophic performance implications if any misspeculation cannot

be handled efficiently. This problem will be revisited in Chapter 6, in the discussion of efficient control speculation mechanisms.

The **I-CS** configuration of IMPACT studied here in fact renders a Hyperblock very similar to that shown in Figure 5.1(c), one that includes the vast majority of paths through the loop. This version, when modulo-scheduled, achieves a speedup of 1.30 relative to the Superblock (**S-CS**) version of the loop (and, if the previously mentioned bank conflict is averted, a speedup of 1.58). This example has demonstrated the primary goals and pointed to some of the pitfalls of predicated region formation. Let us now turn to consideration of the mechanisms involved.

5.3 The Hyperblock Framework

Predicated region selection, the collection of basic blocks into Hyperblocks using predication, is perhaps both the most performance-dictating[4] and least understood part of the ILP compilation process. While the basic concepts have been presented [36] and some techniques for improving the accessibility of code to Hyperblock formation have been described [83], little has been said in the literature regarding supporting the process with effective decision-making mechanisms. The compiler is confronted with a virtually infinite number of choices in transforming code into predicated regions, all of which have potential benefits and costs, and some of which do not become apparent until a much later compilation phase. Today this process relies on somewhat fickle heuristics which often make suboptimal decisions (but, in practice, rarely terrible ones). So bad was the situation (at one time) that some means have been proposed to allow the compiler to make up for bad decisions later in the compilation process, within some limited scope [48].

The basic predicated region selection algorithm outlined in [36] remains today, though it has been adjusted in some ways. The basic heuristic considers blocks in regions naturally bounded by loop back edges or single-entry/single-entry points as candidate blocks for Hyperblock formation. Each path through such a candidate region is examined for dependence height, resource utilization, and execution frequency. Compatible paths (paths

[4] in programs with inherent structure presenting interesting options for the application of predication

that do not unduly slow down other important paths and which are estimated to cohabitate peacefully within available resources) are selected for inclusion; blocks not on these paths are excluded and tail duplication is performed to remove resulting side entrances into the region. This heuristic has been modified to improve its scalability (the number of region paths increases exponentially with serial branches included in a region) and to reduce the amount of tail duplication-related code expansion incurred (by including some decidedly unimportant but relatively "inoffensive" paths into the Hyperblock). These heuristic improvements have reduced compile time cost and run-time transformation overhead, and were an important part of making IMPACT's Hyperblock mechanism practical for Itanium 2.

Since optimizations subsequent to Hyperblock formation have a tendency to reduce resource pressure and dependence height within the predicated region, heuristics have been set to "over-predicate" (i.e., include more paths than might be obviously advisable) on the assumption that subsequent optimization would be able to achieve a beneficial result in the end; in other words, predication allows crossing a sort of energy barrier. Sometimes subsequent optimizations are not as successful as was hoped; Partial Reverse If-Conversion (PRIC) was proposed to "undo" Hyperblock formation to some extent, cleaning up at least the path-penalization aspect of this problem [48]. This model was shown to be effective in the presence of very aggressive if-conversion, though perhaps not as elegant as one might hope. Today, however, profitably predicatable regions are more difficult to identify, and often require preliminary transformations such as loop peeling. These transformations are not easy to perform efficiently, and are not easy to undo completely once done, rendering PRIC a less complete solution.

Much of the work done to make IMPACT a useful compiler for modern general-purpose benchmarks on Itanium involved massaging these region selection routines. This work has reduced the fickleness of the heuristics to some degree and has added new, less code-size-expensive options for cultivating large regions, but it has not fundamentally changed the approach to the problem. Fortunately, it has yielded an understanding of the problem that will be useful in developing future solutions. To summarize:

- The applicability of if-conversion (in a productive way) depends significantly on program control structure and bias. Not all programs exhibit a structure amenable to this kind of transformation. Predication must be viewed as a feature that extends EPIC benefits to a certain class of applications, not as a technique applicable to the optimization of all programs. In doing this it is quite successful, but the scope of programs enabled is limited. Relative to competent Superblock specialization, the benefit of predication is generally associated with reductions in branch misprediction and front end penalties.

- In programs suited to it, Hyperblock formation tends to provide increased planned ILP with less impact on instruction cache structures than similarly aggressive Superblock approaches, although this is not one of its explicit goals (i.e., heuristics do not attempt to optimize for this). More effective control of code specialization in region selection requires knowledge of the enclosing instruction cache footprint. Within a given working set, the creation of excessive "lukewarm" code as a side effect of specializing hot paths can negatively impact cache performance.

- If-conversion practically demands predicate promotion in nonnumeric programs, and stands to benefit strongly from the speculation of potentially excepting operations. Since predicated regions include a "main path" (if one is even prominent) and potentially several "secondary paths," speculated operations will often be off-path. This creates opportunities for frequent and unforeseen penalties, necessitating a new approach to control of potentially expensive speculations, as described in Chapter 6; given these new safeguards, in practice, this seems not to detract significantly from the benefit of Hyperblock formation.

- The heuristics established over the years for selecting Hyperblock regions, when appropriately configured and extended in ways explained in this chapter, are generally effective at making good formation decisions for the SPEC CINT2000 benchmarks (they capture apparent opportunities and do not often form performance-injurious regions). The fact that, in many cases, the minority of program latency cycles are directly visible to the compiler, however, is a serious cause for concern.

- Results are reasonably insensitive to reasonable changes in parameters, more due to limitations in program control structure (lack of compatible paths) than due to easily solved limitations in implementation of the algorithms. While the heuristics generally work well, there is no single setting of today's parameters that results in the best performance for all programs. A more systematic approach might be able to produce better results, but would require much more information than is available to the compiler today. Furthermore, extensive experimentation suggests that, in SPEC CINT2000 and the current microarchitectural context, the head room for improvement is too low to enable meaningful experimentation along these lines.

- Effective control also requires more understanding of the interaction between region formation and what are today separate, previous, and subsequent optimizations. Formation of regions is by no means the end of the interesting part of the compilation process. In addition to traditional optimization and height reduction techniques applicable to Hyperblock regions, there is a developing body of work that performs specific optimizations within Hyperblocks, attacking predicate computations, predicate dependences, dependence merging penalties, etc. [84–86]. Predication also offers opportunities for instruction merging (sharing common operations between predicate paths), a largely unexplored and potentially profitable technique. These transformations can have a substantial impact on the quality of a selected region, but their effects are not anticipated in today's region selection algorithms. Programs appear natively to contain many such opportunities, as programmers appear to favor blocks of near-replicated straight-line code to control flow designed to optimize the number of program statements (as in the *crafty* source example). Perhaps this is an artifact of the perception that branching is expensive and to be avoided.

- Small changes in the predicate network created during region selection can have a large impact on final code performance (for example, the ordering of branches and other instructions in a Hyperblock can be significant). Today, only superficial

changes are practical after regions have been formed. As indicated in the second example of the previous section, speculation (in the form of predicate promotion, the weakening or removal of predicates guarding the execution of operations) reduces critical path length but introduces additional instruction executions and affects register usage and scheduling. It is only one of a number of path height reduction techniques that can affect the benefit of predication.

- The analysis and optimization of heavily predicated code continued to pose compiler scalability and accuracy problems. A promising new Boolean-expression-based approach to data flow in predicated programs has been developed and shown to provide an accurate and efficient solution.

Having pointed out the central threads of the description to follow, let us briefly survey the techniques employed in forming Hyperblocks before proceeding to examine performance results in detail. The Hyperblock formation procedure, as embodied in the module `Lblock`, engages in two rounds of Hyperblock formation. The first focuses on loop and diamond regions, also performing loop peeling as necessary to allow inclusion of selected iterations of nested loops. The second repeats diamond formation and adds general formation to clean up any blocks that did not have the opportunity to be included in the first round (including some tail duplicated regions from the first round of formation). The relevant details, and problems with, this approach are as follows:

5.4 Hyperblock Selection Heuristics

Existing predication heuristics are essentially extrapolated from Superblock heuristics, in that they still largely focus on selecting a main path and adding alternative paths to it. In addition to execution bias, though, multipath optimization must be concerned with the compatibility of paths in terms of resource utilization and dependence height [36]. Hyperblock selection heuristics operate in three fundamental modes: the *path enumerative*, *nested diamond*,[5] and *block-based* (also known as *general*) modes.

[5]In the typical IMPACT nomenclature, this has been referred to as *hammock* mode. In more conventional nomenclature, a diamond may have two conditional blocks (`if ... then ... else`) and a hammock is a degenerate diamond with only one conditional block (an `if ... then`).

These three approaches are used in different program contexts to select those regions of code to be included in Hyperblocks. This section documents these heuristics and the modifications made to them in support of this dissertation work. While the experiments in some programs suggest an opportunity to improve or generalize on these heuristics, they also show that SPEC CINT2000, particularly in the context of Itanium 2, offers inherently little opportunity for dramatically more productive Hyperblock optimization.

In the work leading to this dissertation, Hyperblock formation heuristics were in general improved in two relatively obvious ways to improve the stability[6] and their management of code replication. First, a large degree of reconvergent control flow within an if-conversion candidate region has historically proved problematic for path-based region selection techniques. Such control flow can lead to an explosion of paths (n sequential diamonds, for example, may generate 2^n paths that need to be examined for compatibility). In the past, when an unmanageable number of paths arose, a simpler block-based region former was invoked, but this often generated poor results. Setting the number of acceptable paths very high reduced this problem, but caused region formation to take an unreasonable amount of time. The path-based elements of Hyperblock selection heuristics were therefore given the ability to identify "choke points" that are control-equivalent to the beginning or end of Hyperblock regions.[7] A complex region can be split into subregions at these points, dramatically reducing the degree of path explosion. While this generates slightly more conservative Hyperblocks (since it constrains growth based on dependence height point(s) in the middle of paths as well as at the end) than a full path enumeration, it renders the problem of Hyperblock selection much more practical. With this practicality comes increased stability, since the relatively unsatisfactory block-based selection mechanism is avoided.

The second problem had to do with relatively low-weight paths. These in the past were uniformly excluded from the formation of Hyperblock regions. Often these low-weight

[6]For our purposes, *stability* is loosely defined as the resilience of transformations to changes in coding. For example, a loop containing sequential diamonds should experience the same type and quality of path selection no matter how many times it is unrolled. Stability is important to drawing generalizable lessons from compiler work.

[7]Whether the choke point is with reference to the beginning or the end depends on the mode in use.

paths, however, involved only a few additional instructions (relative to the important paths) and had no impact on dependence height. In this case, it is actually more cost-effective to include the path than to include a branch and, often, significant tail-duplicated code, to support its exclusion. Heuristics have been extended to include these relatively infrequently-taken but harmless paths. This has reduced the degree of code expansion incurred in a typical Hyperblock formation pass without, in general, negatively impacting specialization. Such practices do increase the importance of controlling the negative effects of off-path speculation.

5.4.1 Loop path enumeration mode

The first formation mode, *path enumerative*, is the most obvious derivative of the Superblock model. In this mode, all paths through a single-entry, multiple-exit region that terminate with the selected exit point[8] are enumerated and evaluated with reference to the main path (that which would have been the Superblock) for inclusion into the region.

This model assumes that most loop iterations conclude with traversal of the loop back-edge (or at least reaching the loop-back branch). The Hyperblock formed will focus explicitly on this goal (to allow, for example, for efficient software pipelining). If the loop tends to have only one or a handful of iterations, however, and especially if it tends to terminate with a side exit (one that does not involve the loop-back branch) such iteration-focused transformation will be unprofitable. SPEC CINT2000 (and other common programs) abound with infrequently iterated loops. Loop selection heuristics were therefore modified to avoid treating such "unloopy" loops, since peeling can generally be employed to incorporate commonly executed iterations into surrounding blocks, increasing the overlapping of loop latency with surrounding code (as in the example of Figure 3.7).

[8]The region may have multiple exits, but only one exit is considered the terminus of the region. This point, which in loops corresponds to the tail of the (single permitted) loop back edge, is required to be the terminus of all candidate paths.

5.4.2 Nested diamond mode

In nonloop code, Hyperblock formation relies on the assembly of candidate regions from nested and sequential arrangements of diamonds (and hammocks). Such regions may potentially have side exits, but must have a single entry point. Within these regions, paths are evaluated for profile importance, dependence height, and resource utilization, just as in the loop selection mode.

5.4.3 Block-based mode

A final mode assembles irregular predicated regions from blocks not included in other Hyperblocks. It starts with a seed block and iteratively includes successor blocks, potentially along multiple paths, if they meet certain execution weight and other requirements. General formation is useful but unstructured. It can result in parallel, but not coterminous, paths coexisting for a duration of at least several cycles. This is often of dubious benefit if resource pressure or dynamic effects contribute to the region's execution time.

5.4.4 Loop peeler

The three formation modes are aided simultaneously by one important ancillary technique (in addition to tail duplication). With profiling information it is possible to transform small, short-trip-count loops so they can be parallelized with surrounding code. This transformation is called "loop peeling." In loop peeling, a loop body is replicated in the loop preheader, surrounded by appropriate control flow, to make it possible to include a given number of loop iterations into a Hyperblock. The benefit is that ILP may be sought between these now-straight-lined loop iterations and surrounding code. The costs include the effects of the requisite code replication and the potential inclusion of loop iterations that will be squashed away via predication at run time. Allan, et al. documented the ability of loop peeling, at least in concept, to increase the degree of available parallelism in the immediate neighborhood of loops, in the context of a program dependence graph [87]. The original implementation of loop peeling in IMPACT is described in [83], in which it was shown to produce occasional benefit.

Loop peeling required two important changes to make it a practical and reliable tool in this work. The first was with respect to the code replication performed in peeling. As we have seen in previous examples, loop bodies often disproportionately favor particular included paths, often to the extent that the majority of the code in the loop body is either not executed at all, or is very infrequently invoked. The original loop peeling algorithm, when it went to peel a loop, replicated the entire body of the loop, not just the "hot" blocks to be incorporated into the outer region. This led to a substantial amount of code expansion when the peeler was used aggressively, but had one salutary effect—it avoided making the remainder loop improper.[9] Improper loops have not historically been supported by the IMPACT region formation system.

To reduce the code expansion associated with loop peeling and allow its more frequent use, the region formation techniques were enhanced to operate on improper loops and the loop peeler was modified to replicate only the loop paths actually to be peeled into the outer region. This delivers the same degree of loop peeling effectiveness without creating unneeded copies of loop code.

The second modification to the peeler was in the heuristic used to control the peeling decision. Originally, the loop peeler focused on totally peeling inner loops—if a vast majority of loop invocations could not be totally subsumed in the surrounding region with a given degree of peeling, the peel would not be performed. Furthermore, rarely used iterations could be peeled into an outer region if this increased the number of invocations totally subsumed [83]. Analysis (of SPEC CINT2000 and other benchmarks) revealed that a different set of objectives yielded better typical performance. New loop peeling heuristics peel iterations that are likely to be used on more than a given fraction of invocations. A lower bound is retained on the fraction of invocations the peeled loop must cover, but this bound is substantially reduced from that in the previous approach (it exists only to prevent pointless peeling of many-iteration loops).

[9] An improper loop is one that has more than one entry point, or header. Peeling a partial region and connecting outgoing flows to the successor blocks in the original loop body would render the loop body improper.

Extensive peeling (like unrolling) within predicated regions tends to generate many independent predicate domains. This causes serious problems for IMPACT's traditional approach to data flow analysis, as will be detailed in Section 5.6.2. The improvements to data flow analysis techniques suggested there will allow more general experimentation with peeling.

5.5 Optimization in the Predicated Context

The incorporation of predication into the compiler's internal representation offers unique challenges and opportunities for subsequent optimization passes. Much work has been done in IMPACT to ensure that predication does not hamper subsequent transformations and that it is itself optimizable. Please note that discussion of one of the most important post-if-conversion transformations, predicate promotion (the predicate-domain equivalent of control speculation), is deferred to the next chapter. The use of predication in unrolling for modulo-scheduled loops, undertaken to provide temporal separation for interfering loads and stores, is deferred to Section 8.4.

5.5.1 Partial dead code elimination

Specialization of hot paths in loop bodies often entails promoting variables stored in memory to registers for the duration of a loop. When said variables are live outside the loop exit, it is necessary to store the temporary value back to memory after the loop has completed. If the store was conditional within the loop, simply pushing the store out the loop exits causes speculation of a store—an illegitimate transformation if the store is not to a known-safe location. Furthermore, when cold paths (excluded from the specialized loop body) contain potentially aliased uses and definitions of these variables, it is necessary to insert compensatory stores and loads around these operations. To allow this transformation without speculating stores, IMPACT makes use of generalized predication, predicate defines, and guards that are (1) not generated by if-conversion and (2) live across control block boundaries and loop back edges. IMPACT's sophisticated predicate analysis system transparently handles analysis and some optimization of these

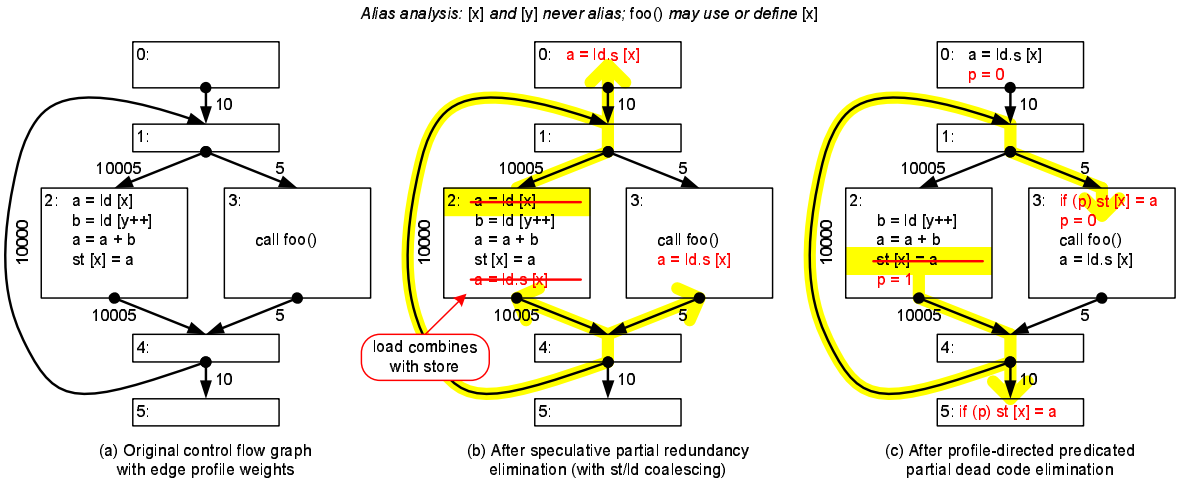


Figure 5.2 The use of predication in partial dead code elimination.

unusual predicates. This technique has been integrated into systematic optimizations such as partial dead code elimination. Figure 5.2 shows an example of the application of this optimization. Figure 5.2(a) shows an initial code segment; Figure 5.2(b) shows the application of partial redundancy elimination to remove a load instruction from the indicated loop body. The corresponding store is removed with the help of predication, as shown in Figure 5.2(c). Predication allows the conditional store instruction to be sunk out of the loop, where a nonpredicated implementation would require insertion of control flow or speculation of a store. Details are given in [88].

5.5.2 Optimization of predicate definitions

Several works have targeted the optimization of predicate definition networks themselves [84, 86, 89]. This work incorporates some lightweight predicate optimization techniques that remove redundancies and enable parallel compares (to a limited degree) in the predicate network generated by if-conversion. These techniques do not, under the circumstances studied here, appear to be very important to performance. With more aggressive use of predication they might have some significance. It should be noted that the limited (relative to that available in the IMPACT EPIC or HPL-PD environments, and in particular to that employed in [84]) predicate define structure of Itanium to some extent hampers optimization of predicate definition networks. This irregularity

also caused problems for branch combining, another predication technique once shown to have substantial benefit (See Section 4.2.3).

5.6 Predicate Relation and Data Flow Analysis

Many elements of the compiler that operate subsequent to the formation of predicated regions (through if-conversion or by some other means) require accurate resolution of control relations such as dominance and postdominance and data flow analyses such as live-variable and reaching-definition [49]. The classical approaches to the analyses that provide this information must be revised in predicated code.

5.6.1 Predicate relation analysis

Analysis of control relations in predicated code is useful in its own right (for notions of dominance and post-dominance) as well as being foundational to the derivation of local and global data flow analyses. The IMPACT compiler uses an approach to predicate control relations, referred to as the Predicate Analysis System (PAS), developed by David August and this dissertation's author, as documented in [55]. This framework, based on the canonical representation of predicate define networks in binary decision diagrams (BDD) [90], performs fully accurate analysis of the control relations among a procedure's predicates. It provides a convenient interface through which the rest of the compiler, including the data flow analysis module, can pose queries about predicate relationships to the analysis BDD it has constructed. Such queries indicate whether, for example, a pair of predicates are disjoint (never simultaneously true), complementary (disjoint, but together subsuming all possible execution conditions), etc. The author extended this system to include relations among the conditions used to define predicates (that is, if `p1 = (r1 > 5)` and `p2 = (r1 < 5)`, assuming both comparisons referenced the same assignment to `r1`, the system would acknowledge that `p1` and `p2` are disjoint. This work is documented in the author's master's thesis [91].

Aside from minor extensions to improve the scalability of this system (modifications were made to allow automatic, sifting BDD variable reordering to be enabled, preventing

undesirable BDD growth in a few cases), the Predicate Analysis System very competently supported the experiments of this work, even with predication very aggressively employed.

5.6.2 Data flow analysis in predicated code

One of the compiler's primary consumers of predicate analysis information is the data flow analysis engine. Data flow analysis is employed in optimization, scheduling, and register allocation phases of the compiler. In its most commonly used application, live-variable analysis, it indicates whether, at a given point in a program, a particular register contains a value that may be used in the future [49]. For an analysis consumer such as register allocation, which must determine the extent of live ranges of values throughout a procedure so it can arrange them, without conflicting with each other, into available machine registers, this analysis is required to be computed for every point in the program, for all machine registers.

The classical formulation of live variable analysis performs the parallel computation of liveness for all procedure registers[10] across the entire extent of the procedure. To accommodate rapid, parallel computation, a single bit in a bit-vector (or variable-sized set in the IMPACT implementation) is assigned to each register, to indicate its liveness. A bit-vector is assigned to each relevant program point (one or two per control block and one or two per instruction) to hold the final result.

Live-variable analysis is monotonic (it can be computed iteratively in a straightforward way) and backward (liveness propagates from later instructions to earlier ones). The effect of each instruction is to generate liveness "upward" toward previous operations for the registers it reads and to kill livenesses, rising from subsequent operations, for those registers it defines. The analysis is typically accelerated by the observation that the process can be split into a local and a global phase, with the global phase processing only summaries of the liveness generation and killing behavior for entire control-equivalent regions (basic blocks). The procedure thus has three phases: First, a linear pass over

[10]These may be virtual registers, which are practically unlimited in number (typically hundreds or a few thousand), rather than machine registers, which are generally fewer in number.

instructions to generate block summaries; second, an iterative global phase to determine liveness at block boundaries; and, third, another linear pass over instructions to propagate liveness locally, within blocks (if instruction-granularity results are desired). The basic equation for live-variable analysis, which may be applied to either a single instruction or to a control-equivalent region, is

$$\mathbf{LIVE}_{in}[i] \leftarrow \mathbf{GEN}[i] \cup (\mathbf{LIVE}_{out}[i] - \mathbf{KILL}[i]) \tag{5.1}$$

In straight-line code, the $\mathbf{LIVE}_{out}$ of one instruction receives the $\mathbf{LIVE}_{in}$ of the subsequent one. When this equation is applied iteratively to all instructions until no further changes occur, liveness has been computed. Where control flow splits and merges exist, however, confluence operations (conservative combinations of multiple paths of liveness) must be performed. If a register is live-out on one path (potentially consumed by some operation later in a program path), it is considered to be generally live-out. Considering a basic block b with multiple successors (perhaps ending in a conditional branch), its iteration equation is modified to be

$$\mathbf{LIVE}_{in}[b] \leftarrow \mathbf{GEN}[b] \cup \left(\bigcup_{s \in \mathrm{succ}(b)} \mathbf{LIVE}_{in}[s] - \mathbf{KILL}[b] \right) \tag{5.2}$$

This equation still does not apply directly to IMPACT's internal representation, once CFS transformations have been applied. First, IMPACT treats code in predicated extended basic blocks (PEBB), the contents of which are not all control-equivalent (due to their potentially containing predicated instructions as well as side-exits). Thus, the merge over all successors is unacceptable, since the exact **KILL** set for each successor may depend on its arc's point of exit from the PEBB. Furthermore, since IMPACT's representation may include predication, individual instructions within a block may have different control conditions. Whereas the extension to Superblocks is, as we shall see, straightforward, the extension of data flow analysis to predicated code is anything but. While control flow arcs among basic blocks determine the necessary paths of propagation for data flow information (via the successor relation), no such arcs exist among predicated operations.

There have been a number of approaches to data flow in predicated code. IMPACT has taken a reverse-if-conversion-based approach, first based on the Predicate Hierarchy Graph (PHG) [79], and later on the predicate analysis system (PAS) [92]. In this approach, referred to as the Predicate Flow Graph (PFG), the analysis attempts to generate, from the instructions in a Hyperblock and from knowledge derived from predicate analysis, a control flow graph that emulates the data flow behavior of the predicated code. The goal is to generate a "stand-in" control flow graph on which standard bit-vector data flow can be run without conservatism. In doing so, two relations must be maintained: instruction ordering and control faithfulness. First, in any traversal of the generated graph, instructions must be visited in the order in which they occur in the actual code. Second, the generated graph must be faithful to the execution conditions of the instructions in the original predicated code segment. If it is not, data flow will be incorrect. A third property is desirable—that this conversion of predication to control flow is complete and free of spurious paths. If the analysis graph contains any remaining predication, the analysis will be conservative (predicated definitions will not kill properly). If spurious paths exist (a *spurious* path is a traversal of the control flow graph that includes two instructions that could not have executed together in a single traversal of the original predicated block due to their predicate relations), data flow will also be conservative. When this final criterion is not achieved, the analysis allows leakage of data flow (in the case of liveness, spuriously long live ranges).

Figure 5.3 shows two applications of the PFG approach to data flow analysis of predicated code. Figure 5.3(a) and (b) show a straightforward example.[11] The numbers in the PFG of (b) indicate the location of instructions from (a). Instruction ordering and control faithfulness are maintained, and the resulting graph need contain no predication. No replication of instructions or control flow is necessary to do so. For simple, directly if-converted code, in which predicated instructions are grouped according to their original control blocks, the PFG generally works well. Figure 5.3(c) shows the same code

[11]The Itanium predicate compare instruction `cmp.lt.unc` sets the two destination predicates to complementary values.

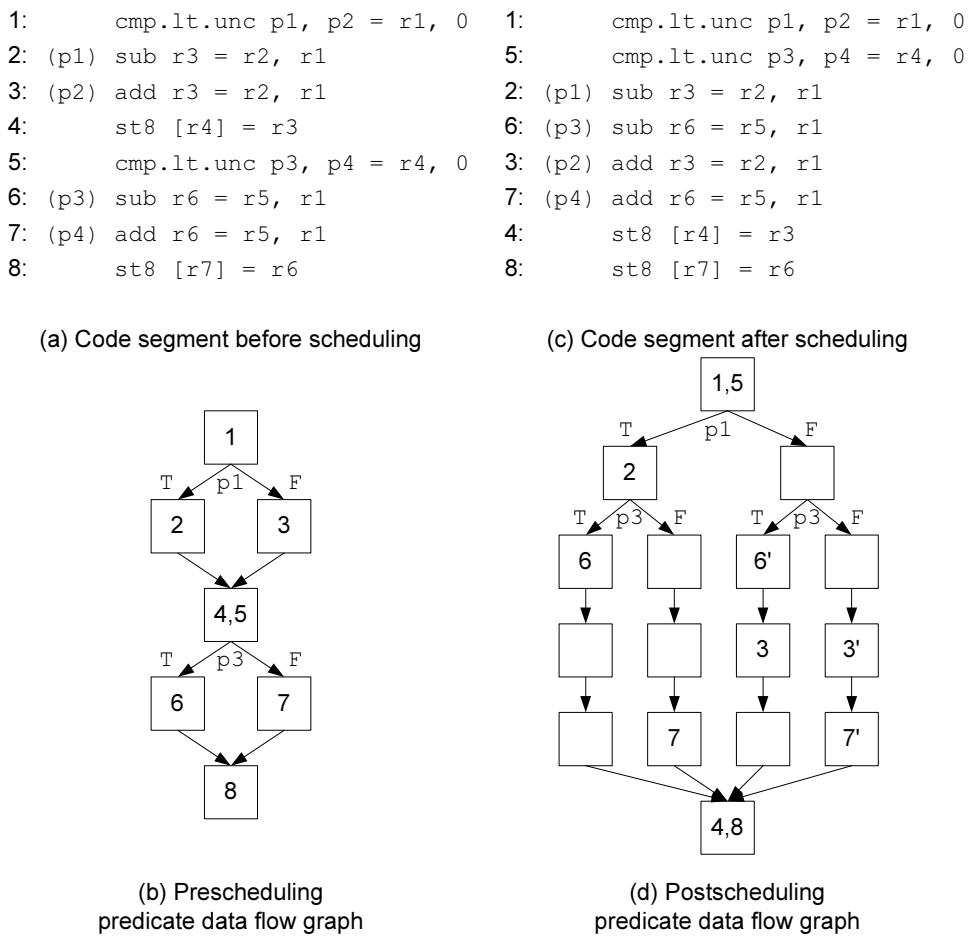


Figure 5.3 Predicated data flow with the Predicate Flow Graph.

after scheduling has occurred. The two sets (1-4 and 5-8) of instructions with independent control flow, implemented as predication, have been interleaved. This is a desired effect of the predicated representation, as has been described. To maintain ordering and control faithfulness in this case, however, the analysis PFG must be more complex. Figure 5.3(d) shows an accurate PFG for this code. On any given traversal of the region, instructions must occur in the order 1, 5, 2, 6, 3, 7, 8, but when 2 executes, 3 should not execute (as these instructions are on opposite predicates). Reproducing this behavior, which is quite simple in predicated form, is not entirely natural in a standard control flow graph. Spurious confluences, as described above, must be avoided, as these will lead to conservatism. Here, four paths must be materialized. In general, in a region of code that meaningfully overlaps n independent predicates, 2^n paths must be enumerated to maintain an exact result. At some point, this growth is unacceptable, and growth must

be curtailed, introducing conservatism.[12] In the case of live-variable analysis, such conservatism can result in a register live range escaping a Hyperblock, potentially causing a register to be wasted for most of a procedure body. This situation occurred frequently in the SPEC CINT2000 benchmarks with the aggressive application of predication (particularly with extensive loop peeling). This resulted in noticeable increases in register stack engine time in *crafty* and other more subtle problems in other benchmarks (for example, it was impossible to experiment with more than a very limited degree of peeling in *twolf* due to an unacceptable degree of path explosion in data flow).

Clearly a different approach is now necessary. Johnson and Schlansker [93] and Gillies et al. [94] addressed the problem of data flow analysis in the presence of predication. They presented a new, predicate-aware data flow framework based on the Predicate Query System (PQS), a predicate analysis representation based on a structured tree of logical domains. By representing the liveness of registers in the nodes of this tree, they could perform accurate data flow in predicated regions without the costly reverse if-conversion and forced approximation of the PFG approach. The PQS, while it provides an orderly structure for such analysis, is not as flexible as the PAS in representing a variety of predicate defining structures accurately [55]. The PAS, on the other hand, lacks an orderly and convenient tree on which to maintain such information. For derivative Boolean manipulations to be performed effectively under PAS, they must take place in the BDD itself.

5.6.3 LED: Efficient predicate-aware data flow analysis under PAS

The author developed a new system of live-variable analysis (initially for register allocation, but extensible to all traditional bit-vector analyses) which operates on predicated, extended basic blocks and requires no reverse if-conversion. It uses the BDD locally (within a PEBB) and standard, bit-vector propagation globally, for efficiency.

[12] This limit is fixed at 256 enumerated paths. Even at this seemingly benign setting, live-variable data flow analysis can consume hundreds of megabytes and take several minutes (and live-variable is run at least dozens of times on a typical procedure). Nonetheless, this bound is frequently exceeded.

```
1:  procedure LED-Block-Setup(B = {I, A})
                                   ▷ B: PEBB with instructions I and successor arcs A.
2:      f_r ← 1                                                    ▷ f_r: reachability
3:      for all i ∈ I in forward order do
4:          f_r[i] ← f_r                                ▷ f_r[i]: instruction reachability
5:          f_e[i] ← f_r ∧ Get-Predicate-Function(i)    ▷ f_e[i]: instruction execution
6:          if Is-Control-Operation(i) then
7:              a ← Get-Associated-Arc(A, i)
8:              f_a[a] ← f_r ∧ Get-Predicate-Function(i)
9:              if Is-Jump-Instruction(i) then
10:                 f_r ← f_r ∧ ¬f_e
11:             end if
12:         end if
13:         srcs[i] ← Summarize-Sources(i)
14:         dsts[i] ← Summarize-Destinations(i)
15:     end for
16:     for all a ∈ remaining flows in A do
17:         f_a[a] ← f_r
18:     end for
19: end procedure
```

Figure 5.4 LED Instruction and arc setup phase.

This approach thus delivers accurate results, provided that data flow need not be sensitive to predicate relations in an interblock sense.[13] This framework will be referred to as Locally Expression-based Data Flow (LED).

Figure 5.4 shows the generic instruction and arc setup procedure for a PEBB. This setup procedure applies not only to live-variable analysis, but to all data flow analyses. Variables specified as f represent nodes, or functions, in the BDD. They can be thought of as generic Boolean expressions, consisting of literals 0 and 1 and (hidden) underlying variables. The function Get-Predicate-Function references the aforementioned Predicate Analysis System (PAS) to acquire such a function for a given predicate. Logical manipulations of these functions, as in line 8, generate new nodes and functions inside

[13] Accuracy will be lost (conservatively) if a register is defined under a predicate in one block and read under a subsumed predicate in another block, and the condition of definition is stronger than (is strictly a subset of) the condition of transit to the block with the use. This limitation could be removed with some degree of effort.

the BDD.[14] Each instruction is annotated with two Boolean functions, f_r and f_e. The former is the reachability condition, the function that most narrowly determines if the instruction will be reached from the top of the PEBB. (The reachability condition f_r is updated in line 10, at the processing of a jump instruction. If the jump is taken ($f_e = 1$), flow leaves the block.[15]) The latter, f_e, is the execution condition, the combination of f_r with the function of the instruction's predicate, as determined by the PAS. For each instruction, summarized lists of sources and destinations are prepared, which express all register uses and definitions. Each listed source and definition is marked with two bits of information: *uncond* and *trans*. The *uncond* flag indicates that the operand is used or defined unconditionally, that is, without regard to the instruction's guarding predicate. This is true of the predicate source itself, as well as the predicate destinations of unconditional-type predicate defining instructions. Such an operand will operate with respect to f_r instead of f_e. The second flag, *trans*, indicates that a definition is transparent; that is, it does not kill liveness; *and*- or *or*-type predicate definitions receive this flag.

Figure 5.5 shows the construction of block **GEN** and **KILL** sets (bit-vectors) in the LED framework. Since the block under examination is a PEBB (potentially having side exit arcs), a distinct **KILL** set must be generated for each exit path, as opposed to the one per block in a classical, basic-block based framework. In the pseudocode, F_g and F_k are maps from register names to BDD functions (expressions) representing the condition under which registers are generated (used before definition) or killed (defined), respectively, for the PEBB being processed. For each source register of an instruction, F_g is updated with the disjunction of the current generation expression and an expression representing the condition under which the register is both read by the instruction (f_r or f_e) and not already defined ($F_k(r)$) (lines 5-10). The corresponding operation is

[14] A reference-counting garbage collection mechanism ensures that these expressions are freed when no longer necessary. The tedious code necessary to ensure that this happens has been omitted from the pseudocode examples, for clarity.

[15] In addition to jumps, control operations may be conditional branches (but only early in the compiler). Since a conditional branch only conditionally causes control flow to leave the block, f_r cannot be updated for these operations. In the latter stages of the compiler, all conditional branches have been converted to predicated jumps, so f_r is always updated.

```
1: procedure LED-Liveness-Gen-Kill(B = {I, A})
                                  ▷ B: PEBB with instructions I and successor arcs A.
2:     F_g = {r → f_g} ← {⋆ → 0}                          ▷ F_g: register GEN functions
3:     F_k = {r → f_k} ← {⋆ → 0}                          ▷ F_k: register KILL functions
4:     for all i ∈ I in forward order do
5:         for all r ∈ srcs[i] do
6:             if uncond[r] then
7:                 F_g(r) ← F_g(r) ∨ (f_r[i] ∧ ¬F_k(r))
8:             else
9:                 F_g(r) ← F_g(r) ∨ (f_e[i] ∧ ¬F_k(r))
10:            end if
11:        end for
12:        for all r ∈ dsts[i] do
13:            if ¬trans[r] then
14:                if uncond[r] then
15:                    F_k(r) ← F_k(r) ∧ f_r[i]
16:                else
17:                    F_k(r) ← F_k(r) ∧ f_e[i]
18:                end if
19:            end if
20:        end for
21:        if Is-Control-Operation(i) then
22:            a ← Get-Associated-Arc(A, i)
23:            KILL[b, a] ← {∀r | f_a[a] ∧ ¬F_k(r) = 0}
24:        end if
25:    end for
26:    for all a ∈ remaining flows in A do
27:        KILL[b, a] ← {∀r | f_a[a] ∧ ¬F_k(r) = 0}
28:    end for
29:    LIVE_in[b] ← GEN[b] ← {∀r | F_g(r) ≠ 0}
30: end procedure
```

Figure 5.5 LED live-variable gen/kill phase.

performed for all destination registers (lines 13-19). These maps (hash tables in the actual implementation) are updated as each instruction is processed in forward order. The **KILL** sets used in global propagation are generated as outgoing control flow arcs are encountered (lines 23 and 27). Here, all registers having a kill expression that *subsumes* the arc's condition of reachability are added to the **KILL** set for the arc. Finally, the **GEN** set for the block is generated in a similar manner (line 29) after all instructions have been processed. Here, though, a satisfiable liveness generation expression (any expression $F_g(r)$ not a literal 0 in the canonical BDD representation) is sufficient to merit inclusion of a register in the **GEN** set.

It should be noted here that there are two opportunities for conservatism in the LED gen/kill phase. The first is with respect to the **GEN** set. If a register is not completely live ($F_g(r) = 1$), its inclusion in the **GEN** set is conservative with respect to the condition $\neg F_g(r)$, since the bit-vector representation is incapable of representing the partial nature of this liveness across a PEBB boundary. Fortunately, this conservatism only manifests itself as a difference in live-variable results if predecessors only define the register r under related predicates. Furthermore, this situation is detectable and, perhaps, even correctable with some additional analysis effort. In practice, given the IMPACT compiler's usage of predication, this potential conservatism does not appear to present a problem at this time.

The second potential conservatism is even less a concern. If conditional branches are included in the PEBB, f_r is not updated to reflect the exit of the taken path from the block. If registers are defined, prior to the branch, under predicates logically related to the condition of the branch, and then used in a control-dependent manner subsequent to the branch, their liveness may escape. This conservatism is easily rectified by the conversion of conditional branches to predicate compare / predicated unconditional jump sequences, as happens in the machine code generation phase of the compiler. Condition analysis [91] can then relate these predicate definitions and remove the potential for conservatism.[16] In practice this second potential form of conservatism is also not a frequent problem.

[16]The same could be done with intact conditional branches, if their conditions were exposed to the predicate and condition analysis.

```
1: procedure LED-Global-Propagation(P = {B})
                                          ▷ P: procedure with blocks B.
2:     repeat
3:         change ← 0
4:         for all B = {I, A} ∈ P do
                                  ▷ B: PEBB with instructions I and successor arcs A.
5:             L ← LIVE_in[b]
6:             for all a ∈ A do
7:                 L ← L ∪ (LIVE_in[s] − KILL[b, a])
8:             end for
9:             if L ≠ LIVE_in[b] then
10:                LIVE_in[b] ← L
11:                change ← 1
12:            end if
13:        end for
14:    until change = 0
15: end procedure
```

Figure 5.6 LED live-variable global propagation phase.

It should be noted that the prior PFG approach suffered from these same modes of conservatism, in addition to its far more serious path limitation problem.

Figure 5.6 shows the global propagation phase of the LED algorithm. This phase is identical to the global propagation phase of classical bit-vector live-variable analysis, with the exception that it has been generalized to handle the multiple, differently located block exits of the PEBB. As shown in Figure 5.5, LED computes a distinct **KILL** set for each exit from a given block, since such exits may depart before the end of the EBB. Otherwise, the algorithm is a straightforward, iterative implementation of Equation (5.2). The global phase can be accelerated by using a work-list, rather than iterating over all blocks, and by processing the blocks in reverse topological order. In such an implementation, the number of iterations of the outer loop is bounded by the degree of loop nesting in the procedure being analyzed [61, pp. 231–235].

Figure 5.7 shows the concluding intrablock propagation phase of the LED algorithm for live-variable analysis (conducted only if instruction-level live-variable results are desired). Here, the PEBB is processed in reverse order, and a liveness map F_v, similar to

```
1: procedure LED-Live-Variable-Block-Propagation(B = {I, A})
                                   ▷ B: PEBB with instructions I and successor arcs A.
2:     F_v = {r → f_v} ← {⋆ → 0}                          ▷ F_v: register LIVE functions
3:     if a ← Get-Fall-Through-Arc(A) then
                ▷ A fall-through arc, if it exists, is not associated with any instruction.
4:         for all r ∈ LIVE_in[dest[a]] do
5:             F_v(r) ← F_v(r) ∨ f_a[a]
6:         end for
7:     end if
8:     for all i ∈ I in reverse order do
9:         for all a ∈ Get-Associated-Arcs(A, i) do
10:            for all r ∈ LIVE_in[dest[a]] do
11:                F_v(r) ← F_v(r) ∨ f_a[a]
12:            end for
13:        end for
14:        LIVE_out[i] ← {∀r | F_v(r) ∧ f_e[i] ≠ 0}
15:        for all r ∈ dsts[i] do
16:            if ¬trans[r] then
17:                if uncond[r] then
18:                    F_v(r) ← F_v(r) ∧ ¬f_r[i]
19:                else
20:                    F_v(r) ← F_v(r) ∧ ¬f_e[i]
21:                end if
22:            end if
23:        end for
24:        for all r ∈ srcs[i] do
25:            if uncond[r] then
26:                F_v(r) ← F_v(r) ∨ f_r[i]
27:            else
28:                F_v(r) ← F_v(r) ∨ f_e[i]
29:            end if
30:        end for
31:        LIVE_in[i] ← {∀r | F_v(r) ∧ f_e[i] ≠ 0}
32:    end for
33: end procedure
```

Figure 5.7 LED live-variable block propagation phase.

the F_g and F_k maps of the gen/kill phase, is maintained and updated with respect to each encountered outgoing arc and instruction. At a flow arc, the map is updated to reflect liveness of all live-out registers[17] under the reachability condition of the arc ($f_a[a]$) (lines 4-6 and 10-12). An instruction's $\mathbf{LIVE}_{out}$ set is computed as containing those registers whose liveness expressions intersect with the instruction's execution condition (line 14).[18] Then, the instruction's destinations kill liveness (lines 16-21) and sources generate it (lines 25-29), and the corresponding $\mathbf{LIVE}_{in}$ set is generated (line 31). Instead of relying on the next instruction's $\mathbf{LIVE}_{in}$ set for an instruction's $\mathbf{LIVE}_{out}$, both sets are provided for each instruction. This is a requirement when the two instructions are on different predicates, because of the intersection performed in line 14, or where the first instruction is a branch or jump. A potential optimization to the algorithm would share some of these sets, which often have identical contents.

A further optimization of the presented algorithm would eliminate the F_g map from the live-variable gen/kill phase indicated in Figure 5.5. Rather than accumulating the generation function in the BDD for each register, the algorithm could instead simply add the register to the **GEN** set in the event of discovering a nonzero intersection. This would accelerate the algorithm somewhat, but the F_g set is useful for determining if the generation is total or partial (to find potential conservatisms).

This concludes the presentation of the LED framework. The algorithms presented generalize readily to other forms of data flow analysis, including reaching-definitions, available-expressions, etc. This generalization is left as an exercise for the reader.

5.6.4 Evaluation of LED and comparison to previous approaches

The LED approach to predicated data flow is clearly superior to the older, predicate flow graph (PFG) approach used previously in IMPACT. It uses less memory, runs faster, and computes more accurate results procedures making complex use of predication. A single run of live-variable analysis across the SPEC CINT2000 suite took 512 s under the

[17] The live-out set of an arc is the live-in set of the destination block.

[18] This is a generally useful, but somewhat arbitrary intersection. In a special mode of live-variable for interference graph construction, this intersection is computed with respect to $f_r[i]$ for predicate-defining instructions with unconditional destinations.

old, reverse if-conversion framework. With LED, the same analysis took only 324 s. LED reduced the size of 1721 live ranges in 91 (out of 7474) procedures. Conservatism spanning more than one PEBB was eliminated in 192 of these live ranges. Such conservatism was resulting in the consumption of excess registers, in a few cases impacting performance (*crafty* was accelerated by 2.7% in the **I-CS** configuration due to the removal of these false live ranges in the register allocator) and in other cases constraining compilation options (e.g., in *twolf*). Considering only those procedures with reduced live ranges, those in which the PFG system suffered reduced precision due to the limitation of path enumeration to 256 paths, LED finished in 90 s the analysis that the PFG approach took 197 s to complete.[19] It should be noted that both the local and global phases of the LED approach could be accelerated substantially with minor algorithmic changes, such as the use of a work-list and topological traversal order in the global phase and a bypassing of the BDD manipulation for blocks containing no interesting predication.[20] These results indicate the LED approach to be a suitable replacement for the PFG method, which frequently lost accuracy and ran slowly in the presence of involved use of predication.

The relation of LED to a few items of previous work merits discussion. Gillies et al. describe a method of global liveness analysis for predicated code using the Predicate Query System (PQS), as noted previously. Their system, within small regions of code, treats control flow and predication identically by representing both in Boolean domains. They then explicitly perform bitwise manipulations on these domains to perform data flow. These manipulations can involve approximations due to the structure of the PQS graph [94, 95]. In PAS/LED, on the other hand, these domains are hidden in and managed by the BDD, so the data flow user manipulates only abstract expressions and never even encounters the underlying Boolean variables. This allows arbitrary improvement in the accuracy or scope of PAS to be translated directly to the data flow analysis, and does not impose the structural requirements on the predicate analysis that constrain PQS so it can build useful domains [55, 91].

[19] The indicated times were measured on a 900-MHz Itanium 2 processor, with IMPACT compiled using the Intel platform compiler, with optimization.

[20] As the vast majority of time is spent in the BDD manipulation phases, such bypassing or optimization of the expression computation routines would be most productive.

Gillies et al. mention the combination of their PQS-based techniques with bit-vector data flow in [94], including methods of extending the useful interaction of predicate functions with the data flow across basic block boundaries. This would, in the author's estimation, result in a system with function similar to the one described here (except that it is constrained by the use of PQS instead of PAS).

In other related work, Eichenberger and Davidson described an expression-based interference graph generation technique for predicated code. Their approach uses a symbolic solver to evaluate the extent of live ranges, but implementation details and indications of efficiency are not presented [96].

5.7 Performance Effects of Predication *in situ*

There are no clear, settled conclusions in the literature on the intrinsic value, best implementation style or preferred means of application of predication, at least for integer applications in general-purpose systems. Different papers offer different viewpoints, often derived in totally different hardware and code generation environments, and often either based on anecdotal examples or incomplete or inconsistent benchmark suites. August, et al. concluded, in a 1998 paper based on the flexible, simulation-only IMPACT VLIW environment and the aggressive IMPACT compiler, that predication offered unique benefits (distinct from those achievable with control speculation alone) but was particularly powerful in most cases only when combined with control speculation [38].

The potential benefits of predication in the abstract are clear. From an instruction issue perspective, predication avoids branch misprediction and fetch redirection. It may also allow co-optimization of compatible paths, rather than creation of different versions, potentially easing instruction cache pressure. From a height reduction or static ILP optimization standpoint, predication allows the overlapping of independent control constructs in a way not possible without significant replication in branching control. Finally, predication renders new, general optimizations more applicable and/or effective. Considering the cycle accounting results of Figure 5.8, where **I-CS** is successful, we should see reductions in branch misprediction flush, front end bubbles, and general dependence height. In this context, the **S-CS** result is also important. Since the **I-CS** framework

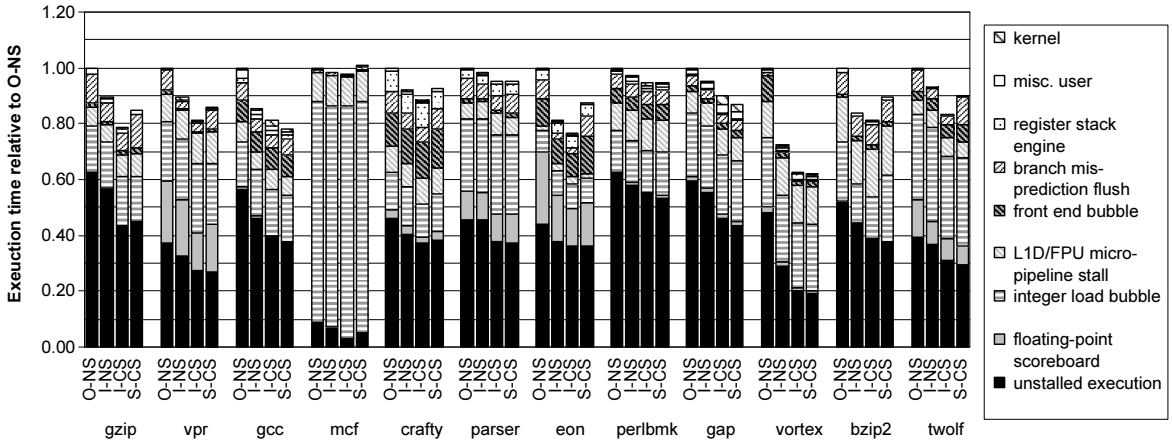


Figure 5.8 Cycle accounting detail.

reflects the effects of both single- and multipath specialization, as it found to be appropriate in different code contexts, comparison with the **S-CS** results allows one to isolate, to some extent, the benefit of predication.

Among the benchmarks in which predication delivers added performance relative to the Superblock approach at the benchmark level (*gzip*, *vpr*, *crafty*, *eon*, *bzip2*, and *twolf*), the difference generally comes chiefly from a reduction in branch misprediction stall cycles and front end bubbles, the former because predication eliminates branches and the latter because multipath specialization creates fewer specialized versions that must compete for instruction cache resources. The benchmark *bzip2* is a special case; its benefit derives from elimination of spurious store-to-load forwarding stalls due to the predicated unrolling technique described in Section 8.4. The following presentation details these positive outcomes and then moves on to consider some of the negative side-effects of predication's application in these experiments.

The constructive interaction of predication and speculation (in the form of predicate promotion) needs to be pointed out. As can be seen in Figure 5.8, the **I-NS** configuration, which has predication but lacks control speculation support, delivers performance that is, except for *eon*, generally inferior to that delivered by a Superblock implementation with

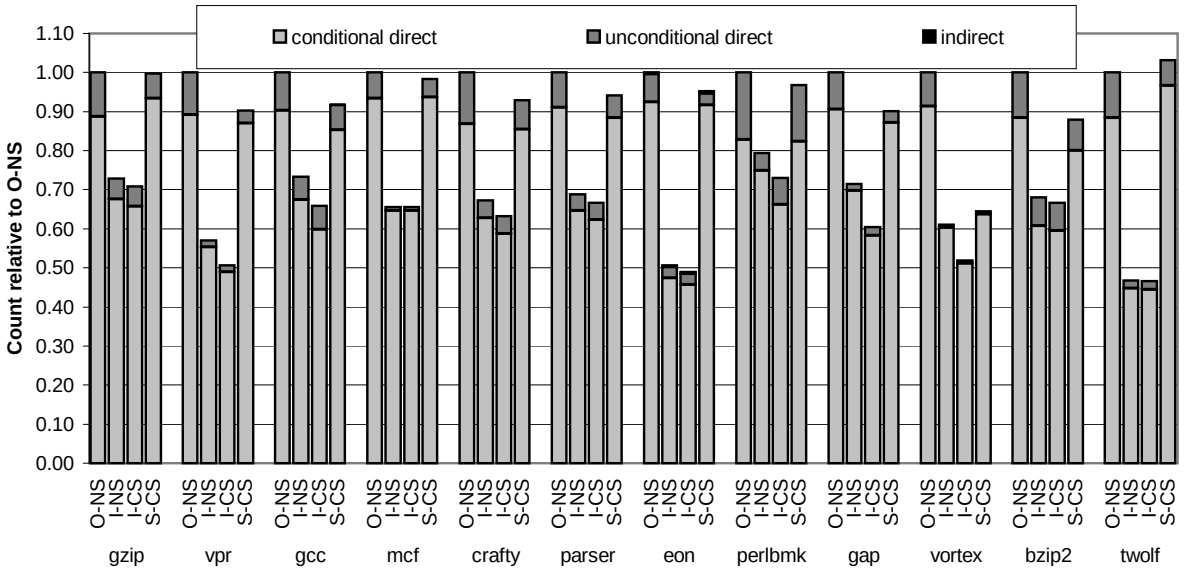


Figure 5.9 Effect of CFS transformation on branch count.

control speculation (**S-CS**).[21] Predication and control speculation have complementary benefits.

5.7.1 Effect on branches and branch prediction

The elimination of branches and branch mispredictions is one area in which if-conversion-based **I-CS** configuration shines relative to the **S-CS** approach. Figure 5.9 shows the effect of if-conversion on branch count. The **I-CS** configuration reduces the number of dynamically encountered conditional branches by an average of 36%, and unconditional branches by 66%. This effect is particularly pronounced in *vpr*, *eon*, *vortex*, and *twolf*, in which fully half of branches are eliminated. These results reveal IMPACT to be an aggressive user of predication. Choi et al. presented a study of if-conversion on the Itanium processor, in which they explored two predicated code generation configurations in the Intel compiler, the default mode and a "maximum" mode. The "maximum" mode if-converts maximally, without regard to profitability expectations—generally with disastrous performance outcomes [39]. IMPACT, on the other hand, even when observing

[21]The victories of **I-NS** in *mcf* and *bzip2* are ignored because of their sensitivity to memory-related events, to which the compiler is oblivious—in a less noisy environment, these distinctions would likely disappear.

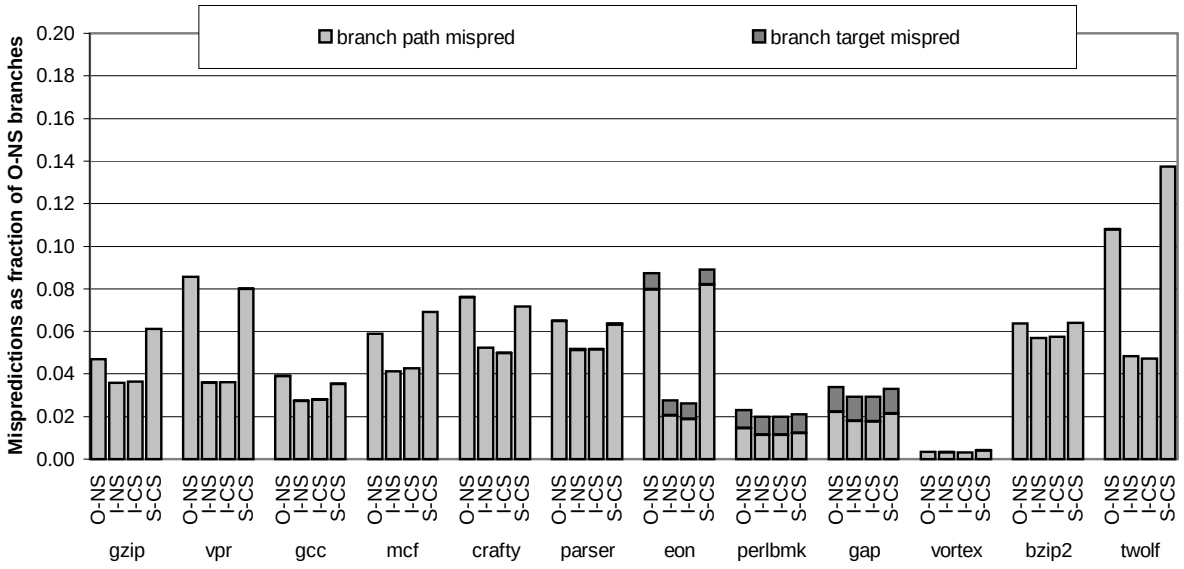


Figure 5.10 Effect of CFS transformation on branch misprediction.

salutary limitations on if-conversion, generally removes a greater proportion of branches than their "maximum" configuration. IMPACT is relatively much more willing to replicate code (by tail duplication and peeling, for example), to create effective predicated regions.

It must be noted that not all of these reductions are due to traditional if-conversion of compatible, comparable paths. In the case of those benchmarks with comparable **I-CS** and **S-CS** performance (*gcc*, *parser*, *perlbmk*, *gap*, and *vortex*), this reduction is largely (but not entirely) due to two factors: (1) the effect of branch combining in Superblock traces, and (2) the incorporation of infrequently executed paths to reduce tail duplication. Branch combining, while it is not traditional if-conversion, uses predication and is therefore not available in the **S-CS** configuration. When applied in the **S-CS** configuration, it was observed not to have significant benefit, although it did substantially reduce the number of branches encountered (see Section 4.2.3).

Formation of predicated regions has a less-pronounced but still significant effect on the number of branch mispredictions encountered (given that the correct prediction rate is generally in excess of 95%). These data are shown in Figure 5.10. The Itanium 2 processor implements a two-level Yeh-Patt branch predictor [97] in a manner tightly integrated into the first-level instruction cache. A secondary history table stores prediction data for lines

evicted from the first-level cache, extending the accuracy of the predictor across large spans of code. This approach results in low misprediction rates, as indicated in the graph, leaving little head room for elimination of misprediction penalties. Nonetheless, in *vpr*, *eon*, and *twolf*, the number of mispredictions is reduced by more than half. Across the benchmark suite, mispredictions are reduced by an average of 35%. Since the contribution of branch misprediction flush to execution time is in no case greater than 10% of execution time, however, even such dramatic reductions typically have only a small influence on net performance.

IMPACT currently does not make use of the exact counted loop prediction available on IA-64, and does not unroll loops to be modulo-scheduled. Enabling either of these features would further improve compiled code control flow efficiency. It should be pointed out that branch misprediction accounts for relatively few cycles on Itanium 2, so its avoidance is not the primary motivation for forming if-converted regions; rather, the regions enable powerful ILP-enhancing transformations.

The effect of if-conversion on branch misprediction has been studied extensively, though many of the studies are becoming somewhat dated. Comparing to the work of Choi et al. on the Itanium processor [39], the only published study on in a real hardware / real compiler setting, we find the **I-CS** configuration to have similar levels of reduction in branch misprediction stall cycles to their "maximum" configuration (and much greater reductions in *eon* and *twolf*). It should be noted again that their "maximum" configuration was undesirable in many other respects; without concerning itself with path dependence height or resource compatibility it often formed Hyperblocks whose *only* appealing characteristic was the elimination of branches. The results of this dissertation show that similar rates of branch misprediction elimination can be performed in a performance-beneficial context.

As the results of [39] show static code size to decrease uniformly with increasing aggressiveness of predication, it appears no code-replicating transformations (such as loop peeling or tail duplication) were performed to cultivate opportunities for region formation in their experiments. This limited possible transformations and potential for gain, resulting in a total reduction in execution cycles by only 2% (tied largely to a 20%

reduction in branch misprediction stall cycles), compared to the 10% reported for our **ILP-NS** configuration.

Other simulation-based studies in various contexts suggested potential improvements in branch prediction accuracy, but are less directly comparable to this work. Mahlke et al. [98] showed for a few Unix utilities and SPEC92 applications that predication (by general if-conversion) dramatically improved branch prediction accuracy by eliminating difficult-to-predict branches. In this study, though, only two-bit counter, BTB-based branch prediction was employed. Another study, performed by Tyson in the context of a variety of dynamically scheduled microarchitectures, showed that predicating only short hammocks removed 30% of program branches and between 30% and 50% of mispredictions [99]. Pnevmatikatos and Sohi showed a similar partial predication scheme to be capable of removing a similar proportion of program branches and characterized the excess issue of predicated-off instructions incurred in doing so [100].

5.7.2 Effect on instruction delivery

One might assume such aggressive predication to dramatically increase the number of instructions issued. In fact, the number of non-`nop` instructions issued in a predicated version of a program (including those squashed) is often no greater than the number issued in the **O-NS** version. Data to be presented in detail in Figure 6.10 in the next chapter will show that, typically, 2-7% of instructions are predicate-squashed in **I-CS** versions of programs, and that the total number of dynamic non-`nop`, non-predicate-squashed instructions is generally increased by only 2-4%. Generally these increases are comparable to or less than the number of instructions removed by optimizations enabled by region specialization, so CFS transformation rarely meaningfully increases stress on processor instruction delivery mechanisms. Not surprisingly, it is in those benchmarks where predication is the most active that it adds the greatest number of dynamic operations. In *gzip*, *vpr*, *crafty*, and *twolf*, predication increases the number of non-`nop` operations (including both useful and predicate-squashed instructions) by between 7% and 14% relative to Superblock-optimized code. These levels of increase generally mean that, even taking into account *nop* instructions which, as we have seen,

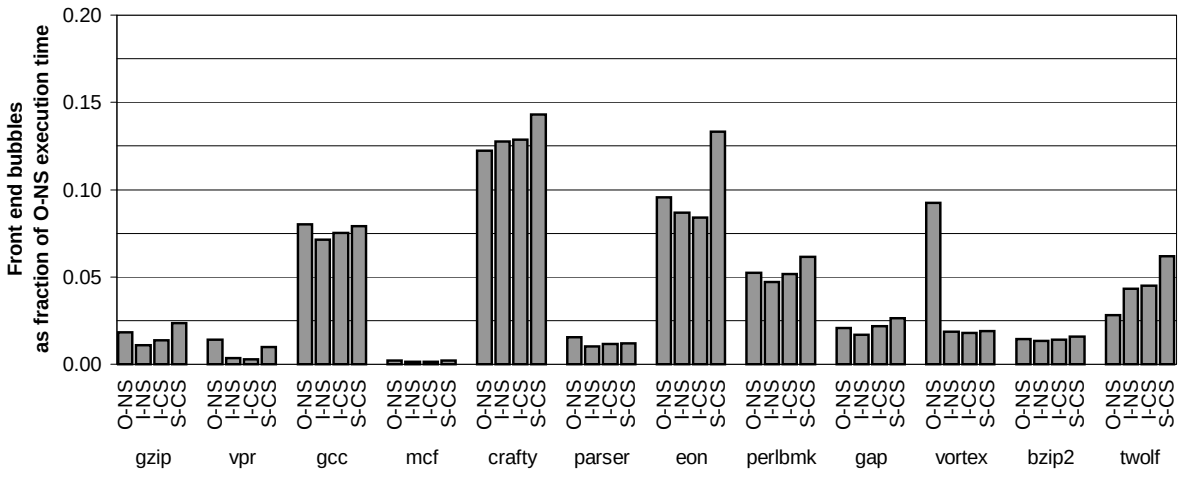


Figure 5.11 Effect of CFS transformation on front end stall.

tend to be less numerous in code with higher ILP, predicated versions of code fetch a slightly greater number of total instructions than nonpredicated versions. Fortunately it is able to maintain a higher level of instruction fetch efficiency, however, due to fewer control redirections (as was shown in Figure 3.12), so this increase in fetched instructions does not translate into worse instruction cache performance.

Choi et al., in published work describing the effect of if-conversion in the SPEC CINT2000 benchmarks on the Itanium processor, demonstrated two levels of if-conversion, one the default mode of the Intel production compiler (at the time), and the other a "maximum" level. At the "maximum" level, which essentially indiscriminately included paths into Hyperblocks, if-conversion dramatically (by more than 50%) increased instruction cache misses in *crafty*, *perlbmk*, and *gap*, with an average increase in misses (across SPEC CINT2000) of 22%, while their much less aggressive "default" level decreased misses by 9% [39]. Their default level removed only 7% of dynamic program branches, so is much less aggressive than the **I-CS** configuration detailed here.[22]

As was noted in Chapter 3, with reference to the data of Figure 3.14, in most cases the **I-CS** configuration improves on, or is at least comparable to, both **O-NS** and **S-CS**

[22]One perhaps significant factor should be mentioned in the relation of this work to that of Choi et al., at least with respect to instruction fetch behavior. That is that their work was on Itanium which, in contrast to Itanium 2, lacked decoupling between the front and back ends. This may impact the cost-effectiveness of if-conversion, since increased back-end interactions may prevent the front-end from progressing.

in its generation of front end stalls. Figure 5.11 shows the number of front end bubble cycles in each configuration, relative to the number of execution cycles in the **O-NS** configuration. In general, **I-CS** reduces on the number of front end bubble cycles in both the **O-NS** and **S-CS** configurations. By treating multiple paths together in a single code region, predicated execution can reduce front end overhead. This improvement does not necessarily come from a reduction in the number of instruction cache misses. In the case of *crafty*, **I-CS** indeed exhibits fewer instruction cache misses than **S-CS**, though slightly more than **O-NS**. Multipath specialization is indeed more cache-efficient than single-path in the context of dynamic, reconvergent control flow. In other benchmarks, however, **I-CS** and **S-CS** exhibit similar levels of instruction cache misses; it appears that other factors are responsible for **I-CS**'s advantage in *eon*. By increasing scheduling freedom in the presence of unbiased branches and by reducing fetch redirection, predication can achieve improved instruction fetch efficiency. The combination of these effects with the increased reliability of prediction in many cases allows the **I-CS** configuration to function with fewer front end stall cycles than other approaches. In two benchmarks, *crafty* and *twolf*, **I-CS** increases the number of front end bubble cycles relative to **O-NS**, in *crafty* by a small margin and in *twolf*, quite substantially. In both these cases, however, it exhibits a reduced number of these cycles relative to the best-performing **S-CS** configuration, which it also outperforms.

Let us briefly consider the behavior of *crafty* and *twolf*. As was noted in Section 3.5, when the code replication due to CFS transformation is largely limited to ejection of "cold" regions, its effect on instruction cache performance is generally favorable. When lukewarm code is expanded, however, there exists a potential for degradation. One such transformation, which is applied only in **I-CS** mode, is loop peeling.

Although *twolf*'s instruction reads are reduced by 21% in the **I-CS** configuration, relative to **O-NS**, due to increased fetch efficiency in densely packed regions, the proliferation of lukewarm code makes the active footprint too large for the cache, causing more misses. In *twolf*, a loop is peeled and the remainder loop, which is itself lukewarm, is then specialized, creating two new, lukewarm regions. Here, front-end stall time is increased roughly 50% by these transformations. In an unconstrained environment, these

would have been appropriate transformations; here, however, some may sacrifice potential performance because they cause the footprint of their enclosing loops to exceed the capacity of the L1I cache. While *twolf* benefited extensively in the net from aggressive transformation, achieving a speedup of 1.20, its gains would have been larger apart from these code bloat effects.

The benchmark *crafty* exhibits similar peeling-related behavior. In both cases, however, peeling proved a net benefit—performance is lost if it is disabled, even though doing so eases instruction cache pressure.

In the aggregate, it is important to note that IMPACT's **I-CS** instruction cache effects are *positive*, delivering an aggregate 5% reduction in first-level instruction cache misses across the SPEC CINT2000 suite (where **S-CS** achieves only a 1% reduction). This contributes, together with generally decreased branch overhead, to a 17% reduction in front-end stall time in **I-CS** code, relative to **O-NS** code. This contrasts with an 8% *increase* due to the **S-CS** configuration—a substantial distinction between the predicated and single-path models.

5.7.3 Effect on planned instruction-level parallelism

In the example of Figure 5.1 it was demonstrated how, in certain circumstances, predication could allow a degree of overlap between independent control constructs that would be impractical to achieve without predication. Such contributions increase planned instruction-level parallelism—relative to Superblock, they do not simply eliminate branch misprediction cycles. This type of behavior can be distinguished by finding cases in which the planned performance increase[23] for **I-CS** exceeds that for **S-CS**. Figure 5.12 shows these data, which are computed from cycle accounting results by subtracting out cycles due to dynamic effects not considered by the compiler. From these it is observed that **I-CS** transformation sometimes makes a shorter, and sometimes a longer, plan of execution than **S-CS**. In *gzip*, *vpr*, *crafty*, and *eon*, the **I-CS** speedup neglecting all dynamic effects

[23]Planned performance takes into account only those cycles the expenditure of which the compiler anticipated, excluding all branch misprediction, cache miss, and other dynamic events. See Section 10.1 for full results.

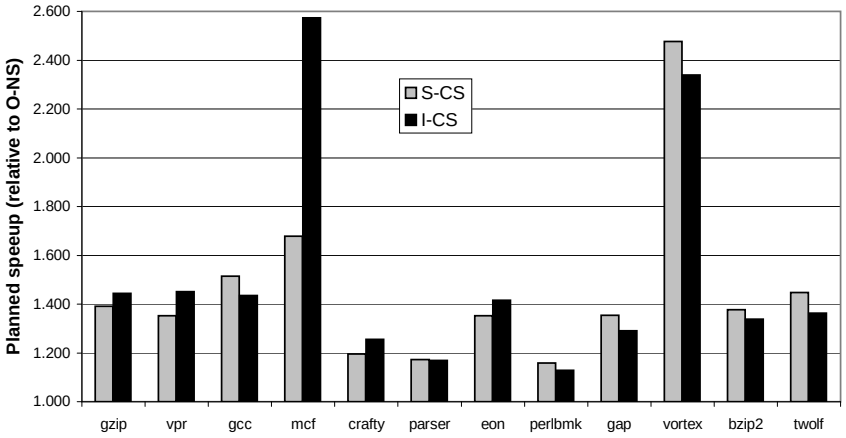


Figure 5.12 Planned speedup comparison of **I-CS** and **S-CS** configurations.

is greater than that achieved by **S-CS**. This is consistent with the generation of a more efficient static plan. In other cases, such as *bzip2* and *twolf*, in which the **I-CS** strategy results in performance increase, its static plan is in fact slower than that for **S-CS**, but it produces a larger reduction in dynamic effects, increasing net performance.

While **I-CS** sometimes produces a more compact plan of execution than **S-CS**, confirming the assertion that predication allows the kind of control structure interleaving shown in the *vpr* example and in previous work [38], this improvement is in the Itanium 2 context dominated by improvements in dynamic behavior, relative to code compiled using the Superblock approach.

5.7.4 Effect on the data memory subsystem

One of the prominent fears with instruction speculation, and particularly the multi-path speculation that occurs in Hyperblocks (in which not only the most commonly executed instruction paths, but also less-frequently executed instructions may be speculated), is that off-path, speculative memory operations will negatively impact performance. They might suffer additional cache misses, cause other loads to suffer misses by polluting the cache, cause extra translation events, or incite costly collisions in the memory substructure. Since this issue emerges most prominently once predicated code is control-speculated, this topic is deferred to Section 8.1.

5.7.5 Effect on register utilization

Exploitation of ILP by overlapping independent strands of computation requires allocation of many register names, even given predicate-aware data flow and register allocation techniques [55, 94] that reduce the number of live range conflicts in predicated code. This cost increases with the promotion of instructions. In certain benchmarks (e.g., *crafty* and *parser*) IMPACT transformations consume many registers in an attempt to expose parallelism. The cost of allocating these registers appears as register stack engine (RSE) activity (**register stack engine** in Figure 5.8) and as an increased number of register spills and fills.

Let us deal with explicit spills and fills first. Integer register spills are dynamically identifiable because of their use of the special `st8.spill` instruction.[24] Spills and fills are tightly correlated to each other, so this gives a good measure of the amount of spill and fill activity in code. A simple counting experiment showed that the effect of CFS transformation and the **I-CS** configuration in particular had a widely variable effect on the absolute number of spills issued, ranging from a dramatic reduction in spills to a dramatic increase. *Gzip*'s spills increased by a factor of 58× in the predicated configurations, and by only 3× in Superblock code. *Twolf*, an aggressive user of predication, suffered 1739× more spills in the predicated configurations and virtually no extra spills in Superblock. *Vortex* suffers a 30-fold increase. Fortunately, however, the amount of this activity relative to total execution of instructions is very small. In the worst case (*twolf* under **I-CS**), spills accounted for less than a half-percent of instructions issued.

RSE activity is a more prominent contributor to execution time in a handful of benchmarks (in most cases, the calls that might instigate RSE activity have been rendered very infrequent by procedure inlining). RSE activity generally increases with CFS transformation. Excluding those benchmarks in which RSE activity always accounted for less than 1% of execution time (and generally much less), *gzip*, *mcf*, *gap*, and *bzip2*, the **I-NS** configuration increased RSE activity by an average of 27%, the **I-CS** configuration by 48%, and the **S-CS** configuration by 16%. These changes are due to the increased overlapping

[24] This is due to the need to preserve `NaT` bits for Sentinel speculation (Chapter 6).

of live ranges in code with higher ILP and, in the case of the predicated configurations, the effects of the inclusion of multiple paths into predicated regions. In *crafty*, in particular, **I-CS** has a profound effect on RSE activity. Recursive functions in *crafty* are extensively inlined, and the inlined regions include predicated, promoted regions that use many registers. This increased utilization, in the context of a recursive function, causes a dramatic increase in spill and fill activity.[25] Thus, this is not inherently a problem with predication, but a problem that emerges as the result of specific inlining and predication decisions in a given code context. These decisions are spread throughout the compiler, making the control of such situations nearly impossible. To conclude, especially when combined with predicate promotion (control speculation), multipath CFS specialization can increase register utilization to an injurious degree, but better holistic management of register resources can likely mitigate this effect in future implementations.

5.7.6 Predication and control speculation

The constructive interaction of predication and control speculation (in the form of predicate promotion) has been pointed out in the literature [38]. This work found predicate promotion of exception-safe operations to be essential to the performance of predicated code. With promotion entirely disabled, performance of predicated code is abysmal, often worse than that of classically optimized code. Hyperblock formation assumes that the dependence height of paths to be included will be absorbed via at least the promotion of the safe included operations; when this does not happen, the main path is penalized. It is apparent that formation without promotion could only generally deliver benefits in machines with much greater misprediction penalties than the one considered here, and then only in applications with secondary paths very compatible with the main paths. The **I-NS** configuration for this reason allows promotion of safe operations by the same promotion mechanism to be discussed in Section 6.4.2. The **I-CS** configuration supports the speculation of potentially-excepting instructions, vastly increasing opportunities for promotion.

[25] See Chapter 7 for further discussion of this issue as it relates to inlining.

Although they interact well to produce gains they could not independently expose, predication and speculation, when combined, have the potential to increase the harmful secondary effects of CFS transformation. The mechanisms behind these effects, while relatively intuitive, have not been described in depth. Given the importance of data delivery latency in an in-order Itanium implementation, it is important to consider the performance effect of off-path loads in the multipath speculation allowed by predicated regions. A discussion of this topic, however, is deferred to Section 6.4.2, which deals with the interaction of control speculation and predication.

5.8 Predication Case Studies Across SPEC CINT2000

To isolate the benefit of the use of predication in if-conversion from the effects of other CFS transformations, we present benchmark- and function-level comparisons of code compiled with the **S-CS** and **I-CS** configurations. The goal is to evoke, at least from this admittedly small sample set of examples, the common cases of benefit from the CFS application of predication, either by conventional if-conversion or otherwise. Examination of function-level results[26] reveals more variation than the benchmark-level results, as some of the benchmarks contain a variety of different behaviors.

Linux kernel support and the *Pfmon* performance monitoring tool allow binning of sampled events by instruction address. Using these, approximate[27] per-function performance comparisons can be performed between two versions of compiled code. This capability was employed to find the examples presented here and to diagnose performance effects of transformations, as benchmark-level performance changes often aggregate too many effects to be useful guides. As an example, Figure 5.13 shows a comparison of **S-CS** code to **I-CS** code for the benchmark *gzip*. The horizontal space taken by a function is its contribution to **S-CS** execution time; the height of each is the ratio of **I-CS** time to **S-CS** execution time (so that the area under the divider represents the run time of

[26]The reader is warned that IMPACT heavily inlines procedure calls, even across files. The results shown for various procedures may be in large part due to optimization of inlined procedure bodies.

[27]See Chapter 9 for details.

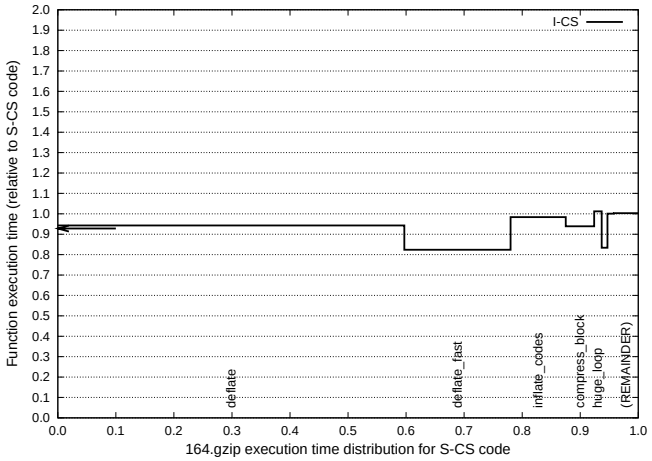


Figure 5.13 Comparison of Superblock (**S-CS**) and Hyperblock (**I-CS**) for *164.gzip*.

the **I-CS** version). The arrow on the left indicates the total benchmark run time in the **I-CS** configuration, relative to **S-CS**.

We have already examined an example (Section 3.3.1) of *gzip*'s benefit from predication. Figure 5.13 shows the relative performance of Superblock and Hyperblock strategies across the benchmark. The two most important functions abound with small, versionable loops in which the incorporation of small hammocks or diamonds of a few instructions improve region coverage, providing relatively easy wins for predication. In some cases, the incorporation of these paths allows for effective modulo schedules with high loop flow retention rates.

Although even without control speculation, Hyperblock formation is more effective than Superblock in extracting performance from *gzip*, control speculation widens the gap between the two approaches, delivering a 1.14× speedup for the predicated approach, compared to a 1.11× speedup for the Superblock code. This increased benefit is due to the improved dependence height and resource accommodation of the incorporated secondary paths. The loads speculated (loads are increased by 19% over a nonspeculative version) tend to hit in cache, causing little increase in load stalls. (The speculative operations from minor paths can cause major disruptions in predicated code, since they are usually "off-path.") Predicate promotion is thus a very important part of a predicated compilation strategy, and works out well in *gzip*.

The inflation algorithm `inflate_codes()` tends to have fewer opportunities for multi-path specialization, so the **I-CS** configuration makes what may as well be Superblocks; hence, little performance difference.

In a few cases across the suite, reasonable variation of Hyperblock formation parameters influenced benchmark performance in positive ways, suggesting that better control of region formation could yield improvements, while confirming that the room for improvement in these benchmarks is relatively small. In *gzip*, the example of Figure 3.2 showed how the inclusion of another path in a key loop Hyperblock could improve the balance of coverage and execution cost for this instance of specialization.

A minor example from the benchmark *vpr* has already been examined. Predication achieves a nearly uniform 5% reduction in execution time for functions accounting for 92% of the program duration. As was seen in Figure 5.12, *vpr* exhibits one of the highest rates of planned speedup for predication, relative to that achieved by **S-CS**. This is consistent with the behavior shown in the previous example.

While *gcc* exhibits examples of procedures in which **I-CS** improves performance by up to 20% (i.e., `regclass()` and `propagate_block()`) largely by enhancing modulo scheduling of important loops containing some control flow, these benefits are outweighed by losses in other procedures. In the cases examined, accounting for a total of 2% performance loss relative to **S-CS** (approximately 1/3 the total deficit), minor scheduling differences in key modulo-scheduled loops, leading to increases in scheduled height or sensitivity to data cache miss, were generally to blame—not systemic losses, but minor perturbations that happened to occur in important places.

Of all the benchmarks, *crafty* posed the most opportunities and most difficulties for predication. This chess program is written in a way that exposes a vast number of control paths, often of roughly equivalent weight, throughout many of its phases. The **I-CS** configuration is in fact able to reduce the number of cycles spent in branch misprediction flush in the **S-CS** configuration by a third, while still reducing the number of planned execution cycles, through an extensive use of predication and predicate promotion. The former removes one-third of branches relative to the **S-CS** code; the latter aids dramatically in reducing the dependence height of if-converted regions, which otherwise proves

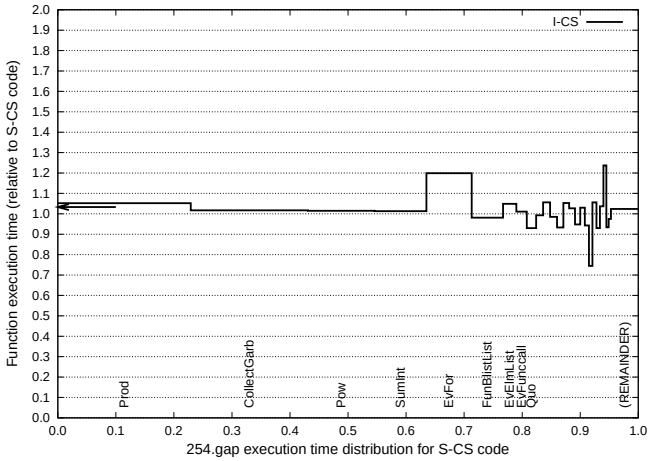


Figure 5.14 Comparison of Superblock (**S-CS**) and Hyperblock (**I-CS**) for *254.gap*.

problematic (*cf.* the **I-NS** result, which is only roughly as effective in terms of net performance as the **S-NS** configuration). As with *twolf*, however, the **I-CS** approach increases front end stall (though not as much as **S-CS** does). As can be seen in Figure 2.6, however, most of the potential of **I-CS** relative to **S-CS** is eroded by an increase in register stack engine overhead, as discussed in Section 5.7.5. Solving this problem would substantially increase the gains of the if-converted code.

Although *eon*'s execution time is spread across a relatively large amount of code, leaving no very prominent examples of highly successful predicated regions, it is clear that if-conversion is extensively employed to remove relatively unbiased branches and to allow path specialization with a lower instruction cache footprint than competitive levels of Superblock-based optimization. While **S-CS** increases instruction cache stalls by 39%, **I-CS** decreases the same category of cycles by 12% (relative to **O-NS** code).[28] The latter also eliminates 70% of *eon*'s branch mispredictions.

The benchmark *gap* includes one of the most prominent examples of degradation due to if-conversion, as indicated for the procedure `EvFor()` in Figure 5.14. As in *crafty*, this problem is due partly to profile variation and partly to an unfortunate assumption in region formation. The predicated region former selects a loop for formation that

[28]Observe the caveat noted in the introduction, that *eon*'s performance under IMPACT optimization is not comparable with production compiler results. This substantial improvement may be eliminated with more effective baseline code optimization.

contains a hammock followed by a side-exit branch, followed by a great deal of other code. The code-bearing side of the hammock contains instructions with a dependence height of at least six cycles, a small fraction of the total dependence height of any loop-carried dependence path, but more than twice the dependence height of the path that leads to the side exit branch. Since the region is a loop, decisions are made relative to the main path that starts at the loop header and ends with the loop continuation branch, so the hammock code is included in the Hyperblock region. Unbeknownst to the compiler, however, the loop frequently exits (more than 90% of the time) early in the first iteration (at the before-mentioned side exit branch). The inclusion of this hammock, which due to the total dependence height of the enclosing region seemed reasonable, heavily penalizes the now-important side exit path. *Gap* is also unfortunate enough to have frequently imbalanced diamonds (or hammocks) in the context of larger candidate regions. Current formation heuristics consider paths through these regions independently, but, in formed Hyperblocks, instructions shared among different paths "bind" the paths together at particular points in the schedule. Without node splitting transformation to remove these binding points, there is a potential for underestimation of expected dependence height for the region to be formed. This is a slightly more general case of the hammock problem just mentioned, and it does not require a profile mismatch for it to impact performance. These cases (and similar ones) occasionally degrade *gap*'s **I-CS** performance. Problems such as this underscore the problem EPIC compilers have with profile variation, first of all, and the unlikelihood of developing static profile estimation techniques that will deliver generally good results in irregular applications—loops in SPEC CINT2000 often are not very "loopy."

Vortex, as shown in Figure 5.15, has component functions that benefit from predication, but these benefits are more than offset by losses in the most prominent function, `SaFindIn()`. If these losses were mitigated, predication would deliver a 2% overall speedup for *vortex*. The loss in this critical function, however, does not seem to be the direct fault of the Hyperblock former. Aggressive loop optimization techniques extract a dramatic amount of loop-invariant or partially redundant code from the main loop in this function, consuming a large number of registers with loop-carried data. A small

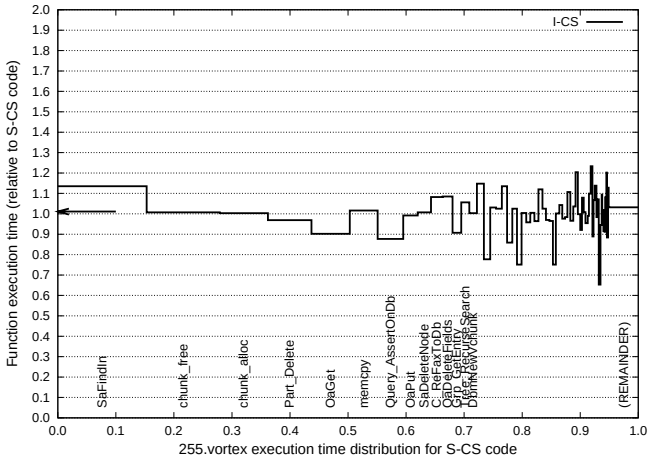


Figure 5.15 Comparison of Superblock (**S-CS**) and Hyperblock (**I-CS**) for *255.vortex*.

increase in registers utilized in the predicated version of the code, however, overloads the available registers and causes performance-detrimental spill and fill code to be inserted into important locations in the function. This problem is compounded by the fact that inclusion of rarely or never-executed paths in the **I-CS** version (to reduce tail duplication) limits the effectiveness of some optimizations in this key loop. Improved predicated code optimization could alleviate this effect; otherwise, the incorporation of these paths needs to be reconsidered.

Bzip2, when compiled with predication, shows a 10% improvement in execution time over the Superblock version. This is not, however, mostly a consequence of standard if-conversion techniques. Rather, this difference is due to the predicated unrolling of critical loops in the function `generateMTFvalues()` to avoid costly, spurious store-load dependence events. The pronounced effect on this important function is indicated in Figure 5.16. Without this technique, **I-CS** performance is closer to **S-CS** performance. A subsequent section will fully describe this transformation. Other, less performance-influencing, functions do exhibit good Hyperblock examples. Generally, these are tight loop versions with multiple paths, in which predication has allowed an efficiently specialized modulo scheduled loop with a very high iteration capture ratio.

Twolf exhibits a near-doubling in CFS benefit with the addition of predication. This gain comes from an increased ability to interleave control-laden paths of computation with surrounding long-latency floating-point operations. In a relatively unusual turn of

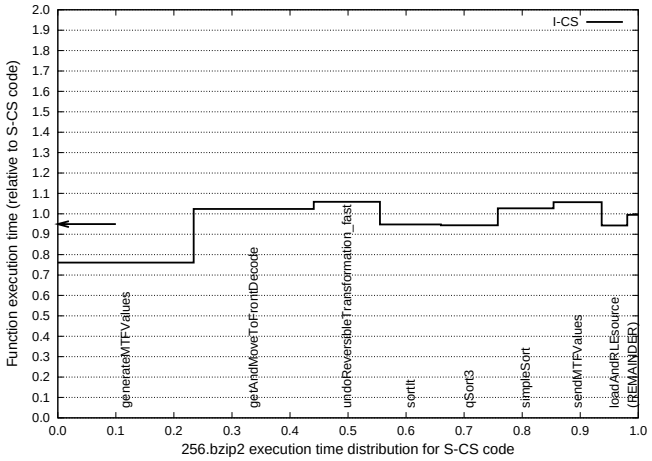


Figure 5.16 Comparison of Superblock (**S-CS**) and Hyperblock (**I-CS**) for *256.bzip2*.

events, the formation of these regions increased dependence height, even with extensive control speculation, as Figure 5.8 shows (notice the increase in **I-CS**' unstalled execution and floating-point scoreboard categories relative to **S-CS**, where successfully predicated benchmarks generally have a decrease). This difference, though, is more than offset by reductions in branch misprediction flush and in front-end stall cycles.

To round out the discussion of the benchmark suite, predication had little to offer *mcf*, due to its domination by data cache stall time, and *parser* and *perlbmk*, due to their tendency not to have exploitable patterns of control flow. These benchmarks show small variation in performance with the application of predication for important procedures (though less-important procedures are often accelerated or decelerated by 10-20%). The performance of predication on SPEC CINT2000 defies summary description. The benchmarks present a variety of contexts, or idioms, as defined by local program structure, execution bias, observable and unobservable dependence height, and surrounding context. Clear examples were presented of how predication extends the benefits of EPIC to less-regular programs by reducing plan height and dynamic effects, but it is clear these situations are not to be found in all programs. While these results show IMPACT's approach to CFS transformation using predication to be generally useful and reasonably stable, in a few cases, general gains from predication were offset or even overwhelmed by losses from lack of generality in the framework or poor decision-making elsewhere. Since

these degradations often result from effects unobservable by the region former, a more holistic approach would be required to generate more consistent and/or effective results.

5.9 Related Work

A few other applications of predication should be noted here.

5.9.1 Wavefront scheduling with predicated hoisting

Predication is used as an alternative to control speculation in wavefront scheduling [34, 50], which is a generalization and adaptation of the trace scheduling technique [44] (as described in Section 10.2.1). IMPACT performs this transformation only opportunistically, through partial dead code elimination after prepass scheduling. Unfortunately, however, the speculative attribute on a load so repredicated cannot be removed, as there remains, in the general speculation model, no record of the load's original execution condition. In a recovery-code based approach, however, this would be possible. It is important to note that the use of predication in wavefront scheduling only schedules around, and does not eliminate, control flow, so it cannot, in general, elicit the branch misprediction flush reduction evidenced here.

5.9.2 Kernel-only or counted-loop modulo scheduling

Specialized, rotating and automatically set stage predicates are used to control the execution of modulo scheduled loops in the kernel-only and counted-loop schemata. Predication in this context prevents unnecessary control speculation and saves code size by avoiding the generation of explicit prologue and epilogue code sequences [24, 57]. The IMPACT compiler does not, however, make use of these features, which are more appropriate for typical floating-point codes than for the control-intensive integer benchmarks studied in this dissertation. Loop iterations-per-invocation are generally lower for integer benchmarks than for the floating point class, making overlapping of prologue and epilogue with surrounding code structures relatively more important to performance. The use of explicit prologue and epilogue, which allow this overlap, as opposed to the kernel-only

schema, which does not, causes a small degree of code size increase (since modulo scheduled loops in integer code typically have a small number of stages, this increase is not as dramatic as it might be in floating-point codes). The counted-loop schema [57] may occasionally prove useful for avoiding performance-injurious instances of misspeculation, as described in Section 6.7.

5.9.3 Special purpose predication

The experiments presented here point to two generally useful aspects of predicated code generation: first, a simplification of control flow and, second, an enabling of path-specialization-based optimizations without the degree of lukewarm code expansion sometimes rendered in Superblock-based optimization. Limited studies have adapted predication to maximize its behavior in these directions, either creating programs with very simple control flow to make better use of loop buffers [101] or using predication in a very aggressive sense (excluding almost no paths) to reduce the code size of embedded applications [102]. Finally, various schemes of partial predication [81] have been proposed to let architectures take advantage of some benefits of predication without the cost of implementing a predicate guard on each instruction.

6 THE VALUE AND APPLICATION OF SPECULATION

Speculative execution means the (potentially partial) execution of an operation before it is certain that all potentially-constraining, incoming dependences are resolved. The dependences broken could be of a control nature (i.e., an instruction is executed that a nonspeculative version of the program would not have executed) or a data nature (i.e., a load instruction executes before a logically preceding store that might write to its address).

Virtually all pipelined microprocessors, and especially out-of-order processors, exploit speculation as a means of increasing the number of instructions available for simultaneous execution. They do this implicitly, generally with respect to both control (branch resolution) and data (store/load address generation) dependences. Hardware features ensure that the effects of these operations, should they begin to execute in a context prohibited by a valid, late-resolved dependence, do not pollute the architectural state with invalid execution results [103–106].

EPIC systems are unique in providing explicit architectural mechanisms for compiler management of speculation, an essential feature for extracting performance from control-intensive programs on a machine that cannot itself reorder operations for issue. This chapter discusses two such forms of speculation, the one, *control speculation*, dealing with control dependences and the other, *data speculation*, with data flow dependences (the former at length and the latter in passing) [38]. Effective use of explicit (or static) speculation is critical to achieving high performance in EPIC systems. As discussed

elsewhere in the detailed benchmark results, control speculation is responsible for a 7% average increase in performance (in both the **S-CS** and the **I-CS** configuration), approximately one third the average 20% improvement due to IMPACT's ILP techniques. It contributes a 10% or higher performance improvement in *gzip*, *vpr*, and *vortex*. Data speculation, while not implemented in the IMPACT framework, has been shown in a very limited initial implementation to deliver up at least 5% additional performance improvement for *gap*. (Due to the successful disambiguation of most critical load / store pairs by IMPACT's sophisticated pointer analysis techniques, data speculation shows less widespread benefits.)

Because in EPIC systems control and data speculation are explicit and may (in fact, frequently do) involve potentially-excepting instructions, architectural, compiler, and operating system accommodations are required to enable speculation. Both forms of speculation entail substantial performance risks, which easily become of great significance if mechanisms are not in place to manage them. Control speculation, when operated using the general speculation schema applied in the IMPACT compiler, can incur spurious page faults on speculative loads, requiring an invocation of the operating system to traverse page tables. In one prominent example, rampant control misspeculation of loads caused a 25% performance degradation in *gcc* (and smaller losses in *parser*, *perlbmk*, and *gap*) before the technique described in Section 6.7 was implemented. This technique turned this degradation into a respectable 5.4% performance increase for control speculation. This approach proved applicable across the suite, making the general speculation schema a useful alternative on the Itanium platform.

Other effects, such as the increased load on the data-side translation lookaside buffer (See Appendix A.1 for details on the DTLB) due to the increase in loads executed in control-speculative code, have more subtle, but also measurable and occasionally significant, effects. This chapter describes the application (or potential for application, in the case of data speculation) of these techniques in the IMPACT compiler, the work that was required to render explicit (general) control speculation a safe and useful technique within the IMPACT environment, design alternatives, and the performance evaluation of speculative execution on Itanium 2.

6.1 Control Speculation

The execution condition of an instruction, the set of circumstances under which the instruction will execute and have its results committed to machine state, is determined by its position relative to control instructions, such as conditional and unconditional branches, and, in a system supporting predication (see Chapter 5), by the value of its Boolean guard operand. The evaluation of the execution condition and its means of influencing instruction execution impose certain constraints on the in-order plan of execution. These are the tangible effects of control dependence. If an instruction is executed before a branch that could have caused control flow to proceed down a different path (that led away from the instruction), the operation's execution condition is altered; likewise, if the Boolean predicate guarding the operation is weakened.[1] The former reflects the breaking of a *control dependence* and the latter a *data dependence*;[2] in either case, however, the result is that the instruction is rendered *control speculative*. Control speculation is the alteration of either instruction position or instruction predicate in such a way that the instruction is allowed to execute in cases where it previously did not [107, 108].

It should be noted that when this chapter refers to control speculation, it is always speaking of control speculation of potentially-excepting instructions. "Safe" instructions, those that have no possibility of causing spurious, program-terminal exceptions, can be control-speculated without regard for these techniques, instruction set or hardware support for speculation. Safe instructions include all instructions that cannot generate program-terminal exceptions and those that can, but can be shown not to do so in the context of the speculation [109]. One example would be a load from a known global or automatic variable. Since the location is known to exist and to be safe to access, such an operation may be speculated without special care. These safe speculations are not subject to the procedures described in this chapter.

[1] The *weakening* of an execution condition allows the controlled instruction freer execution.

[2] In the nomenclature used here, if-conversion [76] converts control dependences into data dependences (unlike, for example, in [75]). Breaking predicate dependences through weakening, however, is *control* and not *data* speculation.

To ensure the clarity of the later material, some terms need to be defined precisely. In the common parlance one might say that control speculation causes instructions to "execute more frequently." Since control speculation is, however, often used to relocate instructions to locations of lesser execution weight (out of a loop, for example, as in Figure 6.3, below), this is a misleading definition. The proper understanding requires a more formal approach. Consider an *instruction instance* to be the execution of a given instruction with a given set of operand values, producing a particular durable transformation in machine state. The existence of an instruction instance implies that it is reached by control flow and that its guard predicate is found to be true. Now consider a *program path* as the dynamic sequence of instruction instances that occur in the course of executing a program. Finally, consider the abstract notion of *all program paths*, the set of all possible traversals of a program. For the purposes of this dissertation, a transformation is control-speculative if it causes, in *any* program path, a *new* instruction instance to appear (when compared to the instances on the program path of the nonspeculative version). A new instruction instance is one that appears in a location where it is not, to use the language of partial redundancy elimination [110], *anticipable* from any other non-speculative instruction instance. Such an instruction, when executed, if it has observable effects (like program terminal exceptions, for example) may change the outcome of the program.

This path-based definition yields some useful derivatives. An *on-path* speculative instance is one that is not "new"; that is, it would have been reached in the nonspeculative program. An *off-path* speculative instance is one that is "new"; that is, it did not simply exist in another position in the nonspeculative traversal. It is these off-path instruction instances that complicate the problem of control speculation, as they can affect both performance and correctness. If an off-path instance is merely a copy of (i.e., has the same basic effect on architectural state as) another on-path instance, then the speculation may have performance but not correctness consequences.[3] If, on the other hand, the off-path instance is *unique*, that is, it is not a copy of some other instance or has a different effect (e.g., it is a load from an address not loaded from by an existing,

[3]This assumes, of course, that dependences other than the control dependence are not broken.

nonspeculative instruction), then it has the potential to generate a program-terminal exception, such as a segmentation violation. It is this latter class of control-speculative instances that requires exceptional care. We therefore consider correctness first, and the various elements of performance later. As will be demonstrated, various techniques of converting the correctness problem into a minimized performance problem have been explored.

In order to maintain program correctness, the off-path instruction instances due to control speculation must not affect the eventual outcome of the program. For some instructions, such as stores and subroutine calls, control speculation is untenable, as these instructions have the potential to modify machine state in such a way that it would be difficult to determine which subsequent results would be affected. We focus instead on one particular class of instructions, *potentially-excepting instructions* (PEI), which may be speculated, but only with hardware support and some care. A PEI is an instruction, such as a load or floating-point operation, which has the potential to raise an exception. Should the execution condition of a PEI be relaxed, additional, *spurious*, exceptions may be encountered in execution of off-path instances, which could irreparably alter program state (possibly even terminating the program, in the case of a page fault). It is thus necessary to provide in both hardware and operating system implementations for suppression or deferral of these spurious events, if PEI are to be speculable. Special care is required if precise interrupts [111] are to be maintained.

6.2 Control Speculation Schemata

Bringmann [109] categorizes PEI speculation mechanisms by their handling of errors (meaning potential, spurious, program-terminal exceptions): they alternately *avoid*, *ignore*, or *resolve* errors. Errors are *avoided* if only provably "safe" instructions are speculated. A safe operation is one guaranteed not to produce spurious program-terminal exceptions, such as a load from a known-valid, fixed address. This speculation model requires no hardware support, but does entail a notion of "safety" within the compiler.

The IMPACT compiler uses such a simple notion of safety for accesses to known, statically allocated and automatic variables.[4] In the second approach, errors are ignored, the assumption being that errors occur infrequently. A nonexcepting, or *silent*, version of an instruction is used in control-speculative situations. Any non-program-terminal exception on a silent operation, such as page faults that resolve to a valid page, are completed immediately. Any potential program-terminal exception generated by such an instruction, on the other hand, is simply ignored. This prevents the manifestation of spurious errors but does not avoid any performance consequences of these events. Finally, in the error resolution approach, speculative instructions can trigger exceptions, including those that are potentially program-terminal, but these are suppressed until a point in the program where the speculation can be checked; that is, until it is possible to determine if the speculative, faulting instance would have occurred in the nonspeculative version of the program. This model requires the most sophisticated hardware and software support.

Given this introductory taxonomy, we can consider various schemata for supporting the speculative execution of PEI, each of which has its own cost-benefit trade-offs. Two of these models, *Sentinel speculation with explicit recovery blocks* and *general speculation*, are available in the Itanium architecture; it is the prerogative of compiler and operating system developers (working in concert) to support either or both of these models [57]. The remaining models are described because of their prominence in the literature and so that the relevance of ideas described in this dissertation to future architectures with features other than those provided in Itanium may be made clear. In each case, the effect of speculating load operations will be described. Generalization to other types of PEI, a straightforward affair, is left to the reader.

6.2.1 General speculation

The general speculation model (also known as the silent speculative operation model) is the oldest and most obvious approach, employed to some extent by Multiflow [56], Cydra 5 [82], HP PA-RISC, and Sun SPARC architectures [109]. Full general speculation

[4] Other means of demonstrating safety, including operation identity and constraint-based systems, are briefly summarized in [109].

support has also long been assumed in experimental work in academia, particularly in that involving predication [81]. In Bringmann's taxonomy, this approach falls into the "ignore errors" category. A potential exception on a speculative operation is investigated immediately. If it is program-terminal, it is assumed to be spurious (off-path) and is simply ignored. If it is resolvable in a non-program-terminal fashion (for example, if it is a page fault to an existing page), it is resolved. This model is available in the Itanium architecture, but is not enabled in current Linux distributions[5] or commercial operating systems.[6]

6.2.2 Sentinel speculation with recovery code

The Sentinel speculation model [47, 107] is the most prominent representative of the "resolve" model of control speculation. Under this model, all exceptions occurring on speculative instructions are deferred. An instruction to be control-speculated is split into two parts: a speculative part, which attempts to perform the operation but stops short of causing any exceptions; and a "sentinel" instruction, a *check* operation in contemporary parlance, which remains under the instruction's original execution condition and pursues exceptions only as they are proven necessary. An "exception" tag (called `NaT`, or "not a thing" in Itanium) on each register propagates potential exception conditions from speculative operations to sentinels.

In the original Sentinel model, the address of the speculative, latent-faulting instruction is also propagated to the check, so that re-execution can be performed using the original instructions (an extension of this model, to support predication and eliminate most explicit check operations, was used in the IMPACT EPIC work that led to the development of this dissertation work [38]). This requires that no dependent, "irreversible" operations be placed between the speculative operation and its check, and that all source operands needed to perform recovery be preserved to the point of the check. The details

[5]The kernel patch and associated utilities necessary to enable this model in Linux, as prepared by the author are, at printing, maintained by the Gelato Initiative and available for download [43].

[6]The ability to silence NULL pointer dereferences [112], a limited form of general speculation, is present for purposes of HP PA-RISC [112] compatibility in contemporary versions of HP-UX for Itanium.

of the eventually resulting inline recovery model are described in detail in [113]. This is the model assumed in the previous evaluation of EPIC using the IMPACT compiler [38].

Instead of performing inline recovery, Itanium designers opted to use the Sentinel model, but with explicit recovery code instead of a potentially complex re-execution of the original, speculated code region, for recovery from deferred exceptions. This comes at the cost of some code expansion, but it is hoped that the recovery code would be very infrequently executed. Explicit recovery code allows some additional freedom in combining check operations and / or rematerializing values required in recovery code that may not be afforded by the inline recovery model. The details of the Itanium architecture's control speculation model can be found in [57, pp. 139 ff.].

6.2.3 Other models proposed in the literature

A variety of other approaches to explicit control speculation have been proposed in the literature, but are now of largely historic interest. One such approach, write-back suppression, was adapted from the Sentinel approach to prevent destruction of source values needed for recovery, after exception deferral [114]. This increased the complexity of the compilation model, but reduced the run-time live range overhead for speculation. Some earlier approaches settled on a great deal of hardware support to avoid potential compiler complexity. Boosting [115] relies on examining the chain of nonspeculative branch resolutions between speculative location and the nonspeculative "home block" of an operation to alternately suppress or recover from speculative exceptions. This requires a potentially costly modification to the ISA and is not trivially extended to allow for speculation in the predicate domain. This approach also requires one or more shadow register files (unless speculation is to be limited to branch shadows) [116].

6.3 Itanium Control Speculation: Selecting an Appropriate Schema

The Itanium architecture supports two basic control speculation schemata: Sentinel speculation (with explicit recovery code) and general speculation. Figure 6.1(a) shows a simple code example that will be used to illustrate the basic features of the two models.

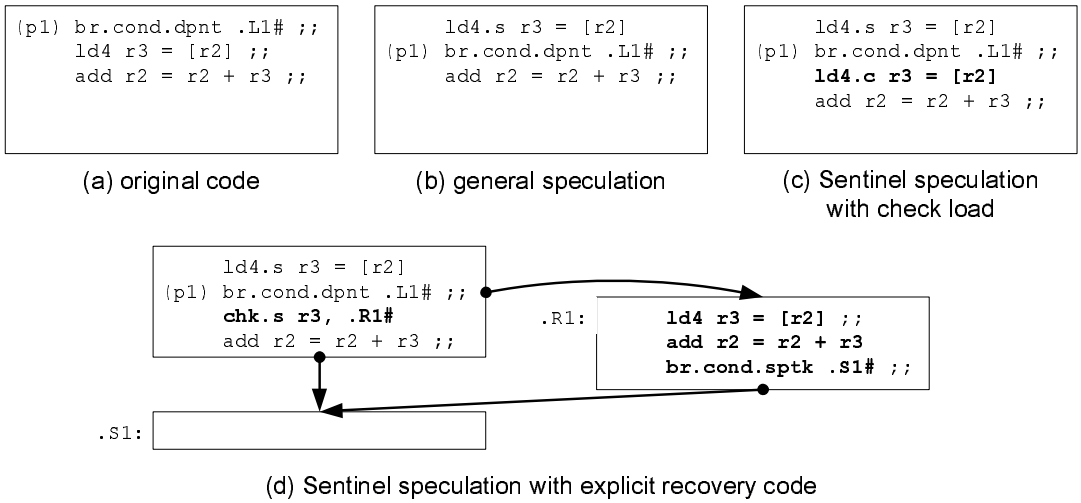


Figure 6.1 Code generation for the two Itanium speculation schemata.

As noted previously, general speculation requires less compiler and run-time effort than the alternative approach. Figure 6.1(b) shows the result of speculating the load operation above the conditional branch. (To simplify the figure, `nop` operations are omitted.) Here, the load is simply converted to its speculative, or "silent," version. Should an exception occur on execution of the load, the exception will be resolved to completion, with the caveat that, if the exception is program-terminal (the requested page is not found, for example), the exception will simply be ignored and the load will set the `NaT` bit in its destination register, `r3`. No bookkeeping is required, and no run-time checking or recovery overhead is incurred. The primary cost of this model is the expense of examining potentially spurious exception conditions. Subsequent discussion will evaluate this cost.

Figures 6.1(c) and (d) show the alternative Sentinel speculation model. Figure 6.1(c) shows the load speculated above the branch, with a *load-check* instruction left in its original position, to execute under its nonspeculative execution condition. At run time, if the speculative load excepts, the check-load will detect the resulting `NaT` in register `r2` and initiate a trap. The trap will execute the load nonspeculatively and resume execution (assuming the fault was not program-terminal) with the subsequent addition. The load-check mechanism is a relatively low-overhead means of speculating load operations, by themselves (with no dependent operations). Nonetheless, it is not without cost—the load-check is an additional operation to schedule; furthermore, the load address in `r2`

must be preserved to the point of the check. In addition to potentially increasing register pressure, this may constrain the scheduling of other operations and also will prevent the use of speculative postincrement loads.

The check-load schema is only a special case. Figure 6.1(d) shows the more typical and more general recovery code pattern. Here, a dedicated check instruction takes the place of the original load, maintaining its execution condition. On a deferred exception, the check received the `NaT` value in register `r3` and branches to a specially prepared recovery code segment at label `R1`. Since, at the completion of recovery, control cannot branch to the instruction immediately following the check, the rest of the check's bundle is replicated. As in the check-load case, values such as the load address must be maintained to the point of the check (or regenerated inside recovery code from other values). Optimization and instruction scheduling are complicated by the need to generate the necessary recovery code while minimizing impact on the common-case path length. A good summary of one system for performing control (and data) speculation in a production compiler environment is given in [72].

While the inline-recovery model [38] has a certain degree of attractiveness relative to the traditional recovery model, it shares with the baseline Sentinel approach some of the same costs relative to the general speculation approach. It requires extension of register live ranges to preserve values necessary for re-execution into the recovery code and requires preservation of a certain degree of the program's original control structure to guard check operations. In some cases, especially where the removal of control structure through predicate promotion is essential to optimization [84], this is burdensome. At the very least, it leaves check instructions trapped after branches, where they may cause a break in issue.[7]

Typically, PEI are load operations. In either schema, control-speculative load instructions are marked "speculative" when relocated by the compiler to a program position (or promoted to a predicate) in which they execute more frequently than in the original program. Every such operation must be treated specially; otherwise, in one of its "off-path"

[7]The Itanium architecture does not allow instructions to issue in slots following a branch instruction.

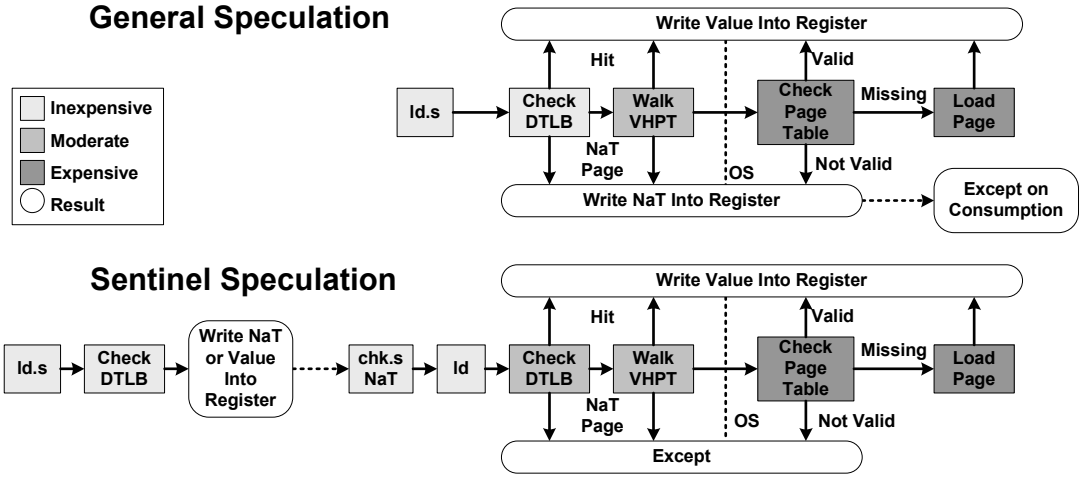


Figure 6.2 General and Sentinel speculation models.

executions (one not dictated by the original program) it might trigger a spurious page fault to a nonexistent page, inappropriately terminating the program.

Control speculation, while it beneficially reduces the dependence height of code, also bears a number of potential negative performance consequences. The IMPACT compiler capably manages these issues, but only after substantial modifications performed in the process of producing this dissertation. It is worthwhile to begin by discussing the trade-offs between the two models available on Itanium.

6.3.1 Performance issues: Sentinel speculation

Figure 6.2 shows the events entailed in completing a speculative load in the two schemata supported on IA-64. In the general speculation model, any speculative load that can be completed successfully (in a non-program-terminating way) is completed at the time the speculative load is executed. A speculative load to an invalid location returns the value `NaT` (not a thing) and does not terminate the program (though doing so may involve an expensive query of the O/S page table). Since nothing remains at the original load site, a predicate used to guard the load, for example, may no longer be necessary, allowing a further optimization. In the Sentinel model (early deferral mode), a speculative load checks only the data translation look-aside buffer (DTLB) for an entry. If one is not found, the load returns a `NaT`. This model defers execution of an

expensive page search (which occurs speculatively under general speculation) but requires additional overhead: a "check" instruction needs to be left at the original load location, to complete execution of speculative loads that missed in DTLB when it is determined that their execution was required. Some state must also be preserved to the point of this check, to support the initiation of recovery code. One study [72] reported that, for the SPEC CINT95 benchmarks, one third the potential gain due to control speculation of potentially-excepting operations is eroded by the overhead of the Sentinel model (when compared to a recovery-code-free general speculation implementation).

6.3.2 Performance issues: general speculation

While general speculation avoids the expense of recovery blocks and state preservation, it incurs a more subtle cost, the magnitude of which becomes evident only in real-machine experimentation. This is that speculative loads may occasionally attempt, for example in the case of programs using pointer/integer union types, to access nonsensical addresses (non-NULL and not in a mapped page). These *wild loads* traverse the page mapping hierarchy and can thus be very expensive (a typical resolution is measured to take at least 500 cycles in a system using the Linux 2.4 kernel, with virtual hashed page table (VHPT) [117] enabled).[8] Since these accesses resolve to nonexistent pages, they result in no update to the virtual memory map hierarchy, so each event causes the same substantial penalty, even if the same bogus address was just probed.

Before implementation of the mechanism described in Section 6.7, there was a danger of this phenomenon occurring in four of the benchmarks (*gcc* in a prominent way, causing it to spend a full 25% of its execution time chasing spurious page faults, and less so in *parser*, *perlbmk*, and *gap*). These costs substantially detracted from and sometimes overwhelmed the benefit of performing control speculation, even at the application level. The literature is peppered with abstract concerns about these potential problems, but this was the first concrete, *in situ* examination of the performance of general control

[8] Common NULL dereferences are handled using a special, architected, 4 kilobyte page at address `0x0000000000000000`; these typically execute (returning a `NaT` value in the load's destination register) with only a 2-cycle penalty in either model.

speculation on nonnumeric applications. If this problem could not have been addressed, the general speculation model could not very well be recommended as a viable alternative to the Sentinel / recovery code model. With the implementation of the techniques to be described shortly, however, this danger is effectively alleviated.

6.3.3 Potential difficulties with the general speculation model

There are three problems with the general, or silent, speculation model, beyond its sometimes undesirable (but manageable) performance implications: debuggability, explicit signal handling, and platform fragmentation. A productization of this technology would have to deal with these issues, in addition to potentially "hardening" the wild load avoidance techniques described below.

Program errors involving pointers often manifest themselves as segmentation violation (`SIGSEGV`) events. In a nonspeculative or Sentinel-speculative system, this event occurs at the offending load operation, speculative or not. This is the idea of the painstakingly preserved notion of precise interrupts [111]. Under the general speculation schema, genuine segmentation violations, such as a `NULL` dereference, on an *on-path* speculative load operation may be suppressed. The programmer is provided with no clear indication that a `NULL` deference occurred at the offending load, and the program may continue beyond the point of the dereference. If this generally converted program-terminal faults to silent ones, this would be a serious problem.[9]

The situation, however, is not quite so bad on Itanium. Recall that the Sentinel speculation model relies on `NaT` bits being propagated from excepting, speculative operations to consumers (and eventually a check). The Itanium architecture specifies that if these values flow to certain operations (generally stores or nonspeculative loads) at which the `NaT` can no longer be propagated without error, a program-terminal exception is raised. In the author's experience, most `NULL` dereferences due to programming errors

[9] In fact, as mentioned earlier, HP PA-RISC systems supported silent `NULL` dereferences, and this caused many debugging headaches and portability problems. A `NULL` dereference on HP PA-RISC returns the value 0.

(or, more commonly, compiler bugs) rapidly propagate to a nearby `NaT`-intolerant operation, facilitating a slightly more complicated debugging process than usual. Whether this is appreciably worse than the usually imperfect ability to debug optimized code [118] probably depends on the specific development context.[10]

A silent exception model will clearly fail if the offending load operation was intended to invoke an activity outside the kernel, since only the kernel page translation routine is explored on a speculative load. Software occasionally is intended to generate a `SIGSEGV` (the segmentation violation signal generated by the kernel on a failed page mapping), which could be suppressed in the general speculation model in a case where it actually was intended to occurred. This might occur in the case of code running under a virtual machine, or code in a dynamically linked environment, in which a `SIGSEGV` is sometimes caught by an explicit handler and used to initiate mapping or remapping of some desired region of the virtual memory address space. Code that relies on this functionality could be broken by a compiler that uses the general speculation schema. At the present time, this is an unsolved problem. It is possible, however, that a set of tests could be developed (particularly in the context of interprocedural or especially whole-program analysis) to identify risk factors such as the presence of signal handlers and `setjmp()`/`longjmp()` or `try...catch` constructs.

Finally, the general speculation model requires a special mode of execution based on a modification to the Linux kernel. Linux for Itanium has been distributed for some time without these features. Vendors would thus likely have to provide different versions of their software for customers with and without these features installed. The desire not to fragment an already modest Linux-ia64 community appears to be the primary reason that, as of this dissertation's printing, general speculation is not supported in the default kernel build.

[10]Since speculative values may occasionally need to be spilled for register allocation purposes, the architecture provides a special `st8.spill` instruction and a complicated mechanism called the `UNaT` for preserving the `NaT` bits on registers that are spilled to memory [57]. The IMPACT compiler, since it does not rely on recovery of these values, does not bother with the `UNaT` mechanism, so spilled `NaT` bits are simply lost. This could prevent detection of some silenced, genuine page faults.

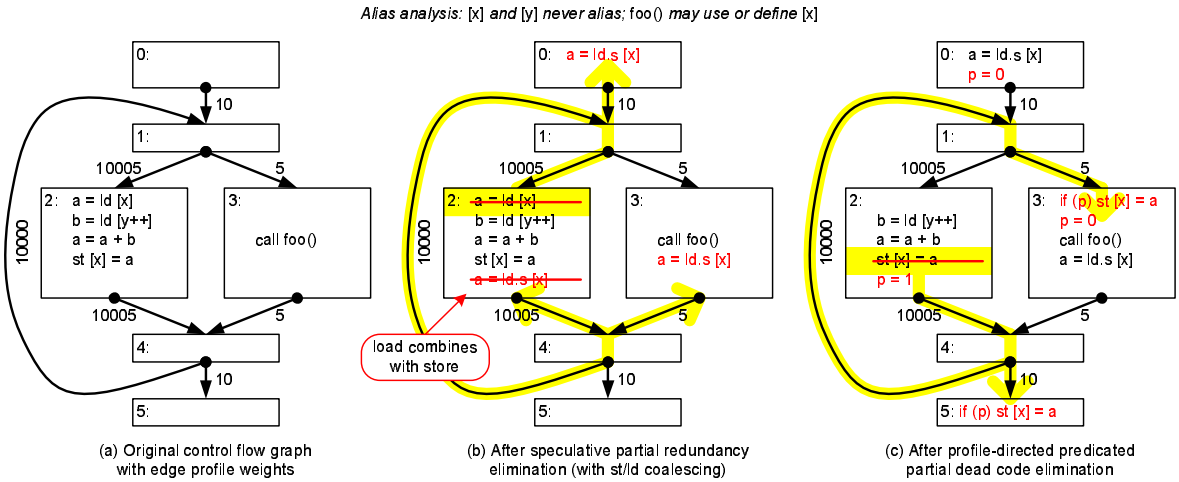


Figure 6.3 The use of speculation in partial redundancy elimination.

6.4 Control Speculation in the Compiler

Control speculation proves an effective tool for ILP enhancement in the EPIC machine. The IMPACT compiler uses speculation in optimization, to reduce predicate control networks, and in instruction scheduling. This section describes the nuances involved in these approaches.

6.4.1 Classical optimizations

As was indicated in the example in Section 3.3.1, the specialization and optimization of prevailing loop paths is critical to achieving high performance in the EPIC machine. Control speculation aids the compiler substantially in performing this difficult chore. The IMPACT compiler has a relatively sophisticated suite of partial redundancy elimination (PRE) and partial dead code elimination (PDE) routines which are applied in the classical code optimization phase (the IMPACT module `Lopti`). This class includes the less general, but still useful, loop invariant code motion and loop global variable migration routines [119], as well as a modern implementation of PRE and PDE [88]. Section 5.5 described the application of predication to extend these techniques; control speculation is also applied to increase their applicability. Figure 6.3 shows an application of partial code elimination to a simple loop. Figure 6.3(a) shows the code prior to optimization. The left-hand path through the loop dominates, and it will presumably be specialized

into a Superblock region in a subsequent optimization stage. Figure 6.3(b) shows an application of partial redundancy elimination that moves the load from the hot path to the header of the loop and the cold path, reducing its execution weight from 10 005 to 15. (Section 5.5 described the additional transformation shown in (c).) This partial redundancy optimization is described in detail in [88].

Although the migrated loads execute much less frequently than before the optimization, there exists a path ($0 \rightarrow 1 \rightarrow 3 \rightarrow 4 \rightarrow 5$) along which the load is executed only in the transformed version of the code; the load is therefore control-speculative. If the load cannot be proven safe (that is, it cannot be ruled out that it could cause a spurious, program-terminal exception) it must be marked as control speculative.

Here the economy of the general speculation model relative to the recovery code model stands out in particular relief. If this transformation were to be performed using Sentinel with recovery code, a check would have to be placed at the original load location. This check would execute 10 005 times. Although a check has zero latency, a slot still must be found in the schedule to accommodate it. This bookkeeping seems to almost negate the benefit of performing the optimization. As usual, however, there is a potential escape for the recovery code model: one could envision a subsequent transformation that would create a second version of the loop, not containing the check. This version could be used after the first traversal through the checked control block 2. This approach, however, could easily become quite complex. (It is, nonetheless, the approach applied by Intel's production compiler [72].) The generality of IMPACT's optimization (and the simplicity of the compiler) is aided by the availability of such a cost-effective form of control speculation.

There is another, more subtle cost of optimization using a recovery-code based schema for control speculation, and that is the difficulty of generating and maintaining recovery code associated with speculative operations (as opposed to for scheduling, which is relatively self-contained and straightforward). Initial experiments with this in the IMPACT environment proved very difficult. An effective solution would require invasive changes to the optimizer infrastructure. As far as the author is aware, furthermore, there has been little work published on the minimization of check infrastructure, which could easily

become quite burdensome. The general speculation model is certainly much more economical to add to a compiler infrastructure than the recovery code tracking approach.

6.4.2 Predicate promotion

As was quite prominent in [38] and as indicated in Chapter 5, achieving maximal benefit from predication usually requires the application of control speculation, in the form of predicate promotion. Predicate promotion is the weakening or removal of the guard predicate on a predicated operation, where weakening means the replacement of one predicate with another that allows the instruction to execute more freely. Promotion removes or relaxes the dependence of the operation being promoted on its controlling operations, allowing it more scheduling freedom and, potentially, the freedom to be optimized together with a broader range of other instructions. Since, however, the instruction is caused to execute more freely than before, it can increase register pressure, cause additional stalls (if one of its source operands' producer instruction is variable- or long-latency), or even cause spurious exceptions.

Since Hyperblocks include multiple paths, and since control dependence is often an important contributor to dependence height, speculation in the form of promotion tends to allow compression of longer paths and, hence, more compatibility among the paths included in regions. This benefit is in addition to the single-path benefit, also achieved, associated with hoisting individual operations above their controlling instructions. These effects were noted early in Hyperblock research [36], but the consequences of promotion in the original form became clear only in real-machine work. This chapter describes the problems introduced into the compiler by this intrusion of control speculation into the predicate domain and the solutions employed in this dissertation work. This work falls into two areas: first, that of predicate promotion itself, and second, dealing with the complex interaction of execution condition and data flow in code using predication and speculation.

Predicate promotion, as described in [36] and as used in [38, 84], has historically been applied very aggressively. Any control speculable instruction was promoted to

the weakest of its predicate ancestors[11] (including to `p0`, the always-true predicate, the degenerate ancestor of all predicates) for which its destination did not overwrite a live value. Finally, for those operations whose destination registers were live under other predicates, preventing their promotion, register renaming was attempted in the hope of exposing an opportunity. Such aggressiveness dramatically reduced the control structure of Hyperblocks, as was its goal. It was applied early in the compilation process to simplify the design of subsequent optimizations, which could work within Hyperblocks mainly as they did in Superblocks—on mostly unpredicated code.

In the real-machine context, however, this aggressive approach causes two problems: First, rampant predicate promotion increases the number of overlapping live ranges (since predicate-aware data flow is able to use a single register to contain two live ranges, as long as the live ranges are mutually exclusive in the control sense [94]). Second, it creates false-escaping live ranges. If an instruction is promoted to a predicate weaker than that of one of the instructions providing its source operands, predicate-aware data flow will detect that the definition does not cover the use. The resulting spurious live range may, in the worst case, be projected throughout the function. Finally, it may allow unprofitable "wild load" speculations, as described in Section 6.7.

These issues caused tangible performance problems in real-machine execution. The author refined predicate promotion to eliminate cases of unsafety (for example, the promotion of loads in such a way that their address operands receive unrelated values) and extended subsequent analyses to make safe judgments regarding the execution condition of speculative operations (more will be said about this later). Additionally, predicate promotion was delayed to the machine code generation stage of the compiler, just before scheduling (and optimizations were enhanced to accommodate this without loss of performance) in preparation for integrating the management of instruction execution condition into scheduling, where more information about dependence and live range count is available to guide it. These changes returned predicate promotion to its status as a

[11] Generally, an ancestor is a predicate guarding one of the instructions in the chain that defines the predicate of the instruction to be promoted. For `or` and `and` type predicates, the only usable ancestor is generally `p0`.

safe and effective (and necessary) partner to predicated region formation and prepared the compiler for a more integrated future implementation, in case that should prove warranted.

6.4.3 Scheduling phases

Finally, control speculation is available to the IMPACT compiler in the scheduling phase. Since IMPACT uses the low-overhead, general speculation model, only minimal constraints are placed on control speculation during scheduling. In fact, instructions are scheduled without regard to control dependence (with respect to branches). Potentially-excepting instructions are marked speculative at the end of the scheduling phase if they have reordered with respect to branches.

The IMPACT scheduler prevents speculation of instructions to a point where they are very unlikely to be on-path (in the documented experiments, to where control flow profile information suggests there is less than a 5% probability that the operation will be on-path). Before the development of the more specific technique described in Section 6.7, this prevented particularly bad cases of wild load creation in *perlbmk*. Since the introduction of those techniques, however, this limit has been retained, as it does not degrade performance in practice and seems a reasonable constraint. The potential importance of curtailing very aggressive (unlikely to be profitable) operation speculation is increased by the sometimes unwarranted aggressiveness of the IMPACT compiler's scheduler.

One might expect that, as control speculation often generates considerable freedom in the scheduling of load operations, the amount of data stall time might be reduced in code compiled with control speculation. In general, though, this is not observed in the experimental data. The reason for this is that, even if slack exists, the compiler is uninformed as to how to distribute that slack along a chain of dependent instructions to achieve the maximal benefit; that is, if two loads are connected in series by dependence and share a certain amount of slack, the compiler does not know which is more likely to miss and therefore should get the more slack. Simple heuristics inserted into the scheduler to employ a portion of available slack for each load operation to hide unanticipated

latencies have not been able to make a very substantial difference in total code performance. This is just part of a larger problem in which the DHASY List scheduler often does not make best use of freedom purchased with speculation (slack distribution) to reduce register live ranges or hide potential latencies.

Software pipelining (using the iterative modulo scheduling technique [120]) relies especially on control speculation to achieve effective overlap among different loop iterations. IMPACT's modulo scheduler uses only a general `while` loop schema (no special accommodations are made for `for` loops or loops with counted, known-in-advance durations. Again, the decision to make this simplification is justified by the generality and simplicity of the general control speculation model (speculation is cheaper than implementing the loops in a less-speculative manner).[12]

6.5 Execution Condition and Data Flow Analysis

Much work has been expended on the IMPACT infrastructure to provide for the derivation of accurate instruction execution conditions in predicated code. As has been well-documented in work by the IMPACT group and others [93, 94, 96], determining the relationships among instruction execution conditions is an important and nontrivial problem. In IMPACT, which uses predication early and extensively in the compilation process, both in structured (if-conversion) and unstructured (optimization, such as the PDE technique described in Chapter 5) ways, this is even more the case than in most compilers. When control speculation is performed, in addition to predication, the situation becomes more complicated, as the *effective* execution condition of an operation may be different than its *necessary* execution condition, the condition under which its results are eventually used. This section briefly describes the operation of the predicate analysis and data flow frameworks of the IMPACT compiler (previously described in Section 5.6) in the context of predicate promotion.

[12] Again, this is a consequence of selecting the general speculation model. Intel's production compiler even uses increased predication (if available) to guard instructions moved above branches to avoid control speculation, due to the cost of check operations [50].

```
1       cmp.gt.unc  p1,p2 = r1, 0
2 (p1)  add  r3 = r1, r2
3 (p1)  add  r5 = r3, r4
4 (p1)  st   [r5] = r6
```

(a) before promotion

```
1       cmp.gt.unc  p1,p2 = r1, 0
2       add  r3 = r1, r2
3       add  r5 = r3, r4
4 (p1)  st   [r5] = r6
```

(b) after promotion

Figure 6.4 Predicate promotion, execution condition, and demotion.

This chapter, at its outset, has defined the concepts of execution condition and control speculation. The previous chapter showed how execution condition is determined for predicated operations, and how this knowledge is applied in adapting traditional data flow algorithms to predicated code; now the impact of control speculation on these analyses must be considered. Classical compiler analyses and optimizations are free to assume that the position and predicate of an instruction stipulate its condition-of-execution in the original program. Transformations are generally considered with respect to this execution condition. This is a fundamental assumption. Live-variable analysis, for example, assumes that whenever an instruction is executed, the values of its operands are important and must be preserved to it. If an instruction has been speculated, however, this is not necessarily the case. Ideally, we would like to minimize the effect of promotion on subsequent optimization, scheduling, and register allocation. For this to happen, instructions must be demotable (that is, able to be repredicated with a predicate more narrowly approximating their original condition of execution).

Figure 6.4(b) shows the result of applying predicate promotion to the code in Figure 6.4(a). Such promotion could allow the instructions in lines 2 and 3 to be scheduled with or above the predicate definition in line 1. Since, after promotion, these operations are unpredicated, conventional concepts of liveness indicate that `r1`, `r2`, and `r4` are always live before instructions 2 and 3. In reality, however, the value computed by these instructions is only used under predicate `p1`. If scheduling should happen to locate instructions 2 and 3 after their original predicate definition, we should like to repredicate them to free additional live ranges for the register allocator to use (assuming, of course, that it has a use for live ranges limited to the condition of `p2`, which is disjoint from the

condition of p1). This reduces the net cost of promotion in those cases where it turned out to be unprofitable. Conventional live-variable analysis will not give us this result.

This problem necessitated an analysis called predicate partial dead code elimination (PPDE). This analysis uses predicated live-variable analysis to discover the strongest possible predicate that can limit the execution of each program instruction, without changing program outcome.[13] This analysis was developed by August and is described in his dissertation [92]. At its completion, each instruction is marked with, in addition to its actual predicate, a demotable predicate expressing this tightest allowable condition of execution.

PPDE was originally based on live-variable analysis. This had the limitation that it could only resolve one "generation" of dead code in a given invocation of the analysis. In Figure 6.4(b), for example, one run of PPDE would identify instruction 3 as demotable to p1, but since instruction 3 actually executes whether p1 is 1 or not, instruction 2 would not be considered demotable unless instruction 3 were actually demoted and the analysis were run again.

To improve on this, creating a single-step analysis that would annotate all instructions with their strongest-possible predicate, regardless of their dependence relation to already-promoted instructions, this dissertation's author developed *critical-variable analysis* (CVA), an enhancement of live-variable analysis. Under CVA, certain instructions which have indisputable or undetectable effect on machine state (stores, subroutine calls, returns, control operations, fill loads, register allocations, and instructions with unsafe macro-register destination operands) are considered to be live and to generate liveness inherently, regardless of their dependent operations or lack thereof. Other operations only become live when liveness propagates to them. In the context of iterative analysis on the predicate flow graph, this generates the desired result—the minimum condition under which the execution of an operation is *really necessary*. This analysis will also remove dead induction variables, a task of which traditional live variable analysis is incapable. The cost of this approach is that local and global phases of CVA are not separable as in

[13]The predicate derived need not necessarily be defined at the program point where the instruction is located at the time the analysis is performed.

```
1       cmp.ge.unc  p1,p2 = r1, 0
2       add  r3 = r1, r2              // promoted from (p1)
3       ld.s r5 = [r3]                // promoted from (p1)
4 (p1)  st   [r5] = r6
```

Figure 6.5 Example illustrating a promoted load and criticality.

live variable analysis. In practice, the analysis is reasonably fast, though, in a work list implementation with proper processing order (and a great improvement over running the previous analysis until no further demotions occurred).

This minimum condition having been determined, it is possible to "abuse" registers that are "live" but not really "critical." For example, in Figure 6.4(b), an instruction under a predicate disjoint with `p1` that defines `r3` could be scheduled between instructions 2 and 3.[14] This reduces the impact of promotion on live range overlap, improving scheduling and register allocation in predicated regions.

6.6 Execution Condition and Safety of Potentially-Excepting Instructions

A subtle danger lurks here, however. Figure 6.5 shows another example code segment like the one of Figure 6.4 except that instruction 3 has been replaced with a load. The load is marked speculative, since it has been promoted from `p1`. Critical variable analysis would determine, as before, that the register `r3` may be abused between instructions 2 and 3 under the condition that `p1` is not 1. This could lead to the load accessing a nonsensical address, depending on the value stored in the conflicting live range. Since general speculation silences spurious exceptions, this will not alter the program outcome, but it will cause a serious performance loss—a wild load. This would be a legal but not a wise transformation. Even worse, if the load was from a provably safe location, it would not have been marked for silencing of its exceptions when it was speculated. In this case, an abuse of its incoming live range would cause a program error.

[14]Dependence-drawing routines in the IMPACT scheduler know how to use the minimum execution condition of each operation appropriately in this regard.

Thus the notion of execution condition had to be modified again to account for the performance-safety of speculative operations. To do this without losing the advantages of the critical variable model (including the potential demotion of potentially-excepting instructions), a modification was made to CVA. Potentially-excepting instructions not included in the list of those with indisputable effect on machine state (mainly loads) are marked "critical" at the start of the analysis. That is, they generate liveness under their full execution condition but are not of themselves automatically live. This allows them to be demoted without assuming that they will be, restoring a correct meaning to the execution condition derived in CVA.

This extension, in combination with the modification to promotion mentioned previously, eliminated a substantial number of spurious exceptions which had not been identified or dealt with before the real-machine work of this dissertation uncovered their deleterious performance effects. The techniques and examples presented here illustrate as perhaps never before the extreme complexity of maintaining proper and performance-salutary concepts of execution condition in an environment with predication and control speculation.

6.7 Minimizing Spurious Exceptions Under General Speculation

One of the most significant challenges in achieving high performance on the Itanium architecture using the IMPACT compiler was the problem of the wild load, a speculative load that accesses an unmappable address. A careful study of the incidences of these wild loads in SPEC CINT2000, as originally compiled using the IMPACT approach, revealed their two dominant causes. First, a chain of operation promotion and demotion (speculation and sinking) events could lead to a load being issued with inappropriate operands. This issue has already been discussed in detail in Section 6.5, so there is no need to belabor it here.

The second mode of wild load creation involves the speculation of operations that load from an inherently unsafe address (that is, the "wildness" of the load is purely due to the speculation of the load itself and not in part to the demotion or sinking of operations on which it depends). Of this mode, there are three identified subtypes: the

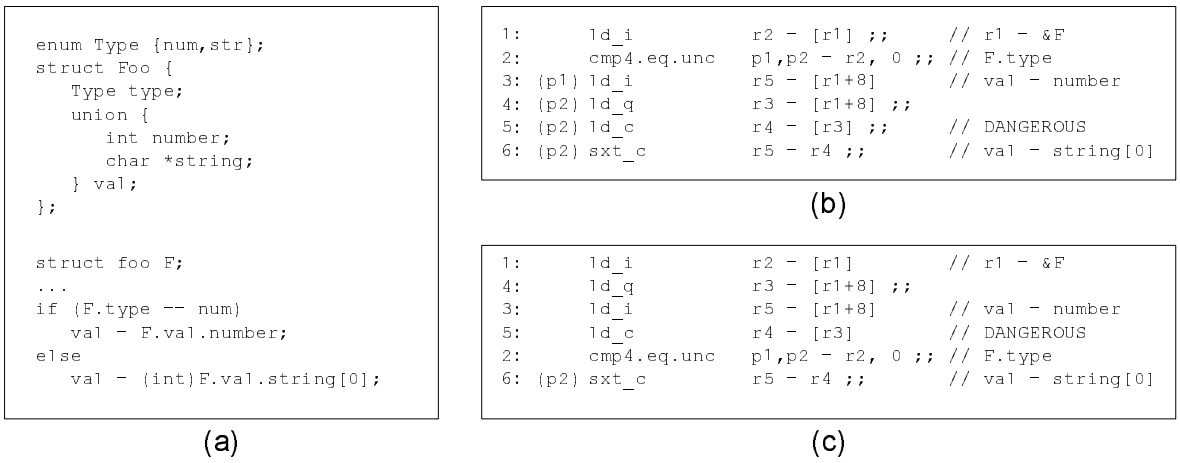


Figure 6.6 Development of a union-field wild load: (a) C source code; (b) initial machine code; (c) after speculation (promotion).

union-field type, the illicit-offset type, and the operational type. These three dominate in *gcc*, *parser*, and *gap*, respectively.

Figure 6.6 shows an example of the union-field form of wild load development. Figure 6.6(a) shows a contrived but representative snippet of source code, consisting of a structure definition and an operation on the structure. The structure is designed to carry two payloads, one an integer value (`number`) and the other a pointer to a character array (`string`). The program determines the type of the structure from the `type` field and accesses either the integer value or the first character of the character string, as appropriate. This snippet is representative of a common case from *gcc* (in the ubiquitous `rtx` structure) and similar cases in other benchmarks. Assuming this section of code is if-converted,[15] the compiler initially generates the machine code shown in Figure 6.6(b).[16] This initial piece of code has a dependence height of five cycles, as indicated by bundle stop bits (`;;`). Operation 3 implements the assignment "`val = F.val.number`," while operations 4, 5, and 6 together implement the assignment "`val = (int)F.val.string[0]`."

[15]This assumption simplifies the example, but ordinary control speculation (the motion of loads across branches rather than the weakening of guard predicates) is also quite capable of manifesting wild loads, and was observed to do so in the experiments.

[16]Loads are shown as having an index, which is not available in Itanium machine code, to simplify the diagram. The address-generating `add` is free of dependence and has no impact on the performance outcome here.

Figure 6.6(c) shows the code after control speculation (in the form of predicate promotion) has been applied and the instructions have been scheduled. The dependence height of the code segment is reduced from 5 to 3, which is appealing. A dangerous situation, however, has been created with respect to instruction 5. This load should be dereferencing a `char` pointer, but the location it accesses contains a pointer only under the condition `(F.type == num)`. The removal of predicate `p2` from instruction 5 has allowed the dereference to occur when the field contains an integer instead. This generates an address translation activity for a nonsensical address, incurring a whopping penalty if the nonsensical address does not happen to fall within a valid, mapped page or the special 4 kilobyte `NaT` page at address `0x0000000000000000`. This mode is referred to as the "union-field" wild load type.

Another common mode of wild load creation is due to the use of computed offsets of a pointer, particularly negative ones.[17] If the pointer is `NULL` and this offset is applied, causing a subsequent dereference to fall outside the mapped `NaT` page, the arduous page hierarchy traversal will ensue. This is referred to as the "illicit-offset" type.

A related problem occurs with respect to unaligned accesses, which also trigger expensive exceptional handling. Programmers sometimes use the least-significant bits of a pointer value to store Boolean flags. One prominent example is in a common implementation of binary decision diagrams (BDD), the Colorado University Decision Diagram (CUDD) package [121]. In this application, pointers are used to reference BDD nodes. If a reference is to the inversion of the node, the lowest-order bit is set to 1. A related construct is used in the benchmark *254.gap* (there, though, a 1 in the low-order bit denotes a value that is not a pointer at all). A dereference of one of these fields, apart from the guard condition that checks the low-order bit and adjusts the pointer or control flow appropriately, will cause either an unaligned access or an address translation. This and other related modes constitute the "operational" wild load type, in which local operations can yield clues that a load speculation is potentially harmful.

[17] Negative offsets are commonly used in memory allocation routines. Information pertaining to the managed block of memory is stored at a negative offset from the pointer used by the rest of the program to access the stored data.

These modes occurred to varying degrees in *gcc*, *parser*, *perlbmk*, and *gap*, in which they caused between 5% and 25% of execution cycles to be spent in the kernel handling spurious page faults before they were mitigated. Although the speculation of these loads causes a severe problem within the general speculation schema, they may also degrade performance in a recovery code-based approach. Wild loads could also be a problem for compilers using the recovery code model, though not as serious a problem as for general speculation, due to misses in the L1TLB incurring L2TLB searches and, depending on the operating system configuration, walks of the VHPT.

The next section describes a means of preventing these injurious loads that, at least for these benchmarks, solves the problem. It should be pointed out, however, that these methods are only heuristics, and these events are not totally avoidable in a weakly typed language like C. It should not, however, be difficult to instruct programmers how to avoid conditions that are not detectable by the described techniques, and the author sees these constraints as placing no real and injurious limits on the programmer's freedom.

6.7.1 Mitigating the wild load problem

Careful study of the wild loads in these cases, together with the source code that generated them, led to a set of simple heuristic analyses that could mark these loads to prevent their highly unprofitable control speculation. The solution developed occurs in two phases (`PtoL`, the lowering phase between the high- and low-level representations, and `Lssaopti`, the SSA-enabled portion of the compiler back-end), due to practical constraints in the IMPACT compiler infrastructure, but it would not have to be implemented this way. The description here, therefore, abstracts away the details of this two-stage implementation.

Figure 6.7 shows a pseudocode representation of the algorithm for identifying loads that have the potential to be wild due to dereferences of a union field. The algorithm is conducted in two linear passes over a procedure's operations in an already-constructed SSA def/use graph [122]. In the first pass, SSA definitions resulting from those loads identified syntactically by the front end as accessing union fields are collected into a work list. Second, this property is propagated along use-arcs throughout the SSA graph. The

```
1: procedure MARK-UNION-FIELD-WILD-LOADS(S)  ▷ S: the procedure SSA graph
2:     U ← V ← {}
3:     for all o ∈ S do                          ▷ Search all operations in the SSA graph
4:         if IS-LOAD(o) and  LOAD-FROM-UNION-FIELD(o) then
5:             U ← U ∪ dest[o]
6:         end if
7:     end for                          ▷ U contains all SSAs defined by loads from union fields.
8:     for all u ∈ U do                          ▷ Forward-propagation on SSA use links
9:         for all v ∈ (use-ops[u] − V) do
10:            if IS-MOVE(v) or IS-PHI(v)  or IS-ARITHMETIC(v) then
11:                U = U ∪ dest[v]
12:            else if IS-LOAD(v) then
13:                PREVENT-SPECULATION(v)
14:            end if
15:            V = V ∪ v
16:        end for
17:        U = U − u
18:    end for
19: end procedure
```

Figure 6.7 Union-field wild load speculation avoidance algorithm.

propagation is only through move, phi (SSA merge), and arithmetic/logical operators, not, for example, through loads. When such a propagated value reaches a load, the load is marked to prevent its speculation.

Propagation is *to* but not *through* load (memory dereference) operations. The reasoning behind this is as follows: loads that immediately dereference the union field, or an offset thereof, are in danger of dereferencing an illicit pointer on an off-path execution instance. It is thus critical that these loads not be speculated. Since these loads will thus be "trapped" under their original execution condition, any subsequent dereferences of the values they produce are protected by flow dependence from being speculated dangerously (at least with respect to the indicated union field). To avoid an overbroad restriction of speculation (recall that *all* speculation is prevented by the antispeculation attribute, not just that relative to a particular condition), propagation stops at load operations.

Figure 6.8 shows the algorithm for marking those loads susceptible to the illicit-offset pattern of wild load. This algorithm is stated as three passes: one that prepares the lists

```
1: procedure MARK-ILLICIT-OFFSET-WILD-LOADS(S)        ▷ S: the procedure SSA
       graph
2:     W ← L ← N ← {}
3:     for all o ∈ S do                          ▷ Search all operations in the SSA graph
4:         if IS-BRANCH(o) and NULL-COMPARISON(o) then
5:             W ← W ∪ VARIABLE-OPERAND(o)
6:         else if IS-LOAD(o) then
7:             L ← L ∪ o
8:         end if
9:     end for                          ▷ L contains loads; W contains NULL-checked SSAs
10:    for all w ∈ W do                                ▷ Back-propagation on SSA def links
11:        d ← def-op[w]
12:        if IS-MOVE(d) or IS-PHI(d) then
13:            W ← W ∪ source-SSAs[d] − V              ▷ No node visited more than once
14:            V ← V ∪ source-SSAs[d]
15:        else if IS-LOAD(d) then
16:            N ← N ∪ dest-SSA[d]
17:        end if
18:        W ← W − w
19:    end for                                  ▷ N contains all NULL-checked, loaded SSAs
20:    for all l ∈ L do                          ▷ Back-trace address generation of all loads
21:        if ILLICIT-OFFSET-ADDRESS(l, N, S) then
22:            PREVENT-SPECULATION(l)
23:        end if
24:    end for
25: end procedure
```

Figure 6.8 Illicit-offset wild load avoidance algorithm.

of loads and `NULL`-check branches, one that propagates the "`NULL`-checked" property up the SSA graph, and, finally, one that back-traces the address calculation of each load and finds the possibility of an illicit adjustment of a `NULL`-checked value. This back-tracing is essentially similar to that commonly used in low-level memory disambiguation and dependence testing, so it is not detailed here. In the experimental configuration, it simply detects negative constant offsets of `NULL`-checked values. This ignores potential accesses beyond the 4 kilobyte `NaT` page in the positive direction, but these seem to be an infrequent problem. A more complete solution should address this.

The algorithm for detecting the third mode of wild load, the operational form, would be substantially similar to that shown in Figure 6.8, so it need not be included here.

6.7.2 Evaluating the wild load solution

A practical evaluation of the wild load solution described in the previous section indicates that it functions quite well. The excessive time spent managing spurious page faults, especially in *gcc*, *perlbmk*, and *gap*, is curtailed. Only *gcc* and *gap* indicate an increase in kernel time with speculation, and this increase contributes less than 2% to benchmark execution time in each case. The remaining cases in *gap* could be removed with an implementation of the operational wild load detection algorithm; those in *gcc* appear to require more information than is conveniently propagated to the phases of the compiler in which the current algorithms act. Nonetheless, this relatively simple solution solved the vast majority of the wild load-related performance problems without rendering control speculation ineffective.

A theoretical evaluation of the wild load mitigation heuristics, however, shows points for improvement both in the direction of safety and in the direction of relaxing conservatism. With respect to safety, these heuristics could be deceived in a number of ways. In the present *ad hoc* implementation, for example, union field references are detected in a purely syntactic manner. If the address of a union field were taken, and this pointer were dereferenced elsewhere, the heuristic would be oblivious to the fact that a "dangerous" location is being dereferenced. Furthermore, programmers often use "`void *`" fields, which are not explicitly unions, to hold mixed pointer/integer data. A more general solution, therefore, would possess the ability to detect union-like behavior in nonunion fields. Likewise, since the SSA graph is constructed on register-promoted communications only, propagation of any of the interesting properties through any address-taken or structure-field variables is ignored. Means exist for dealing with these issues in the construction of a more resilient SSA graph, but are inaccessible in the current IMPACT implementation.

The approach is potentially overrestrictive in two ways. First, it may detect innocent loads and mark them as "wild candidates." If the approach marked all the loads in

offending benchmarks, it would certainly curtail the wild load problem, but it would also render control speculation impotent to improve performance. Empirical results, however, show that the technique described marks only a small minority of dynamic load instances as potentially wild. Across the suite, only 1% of dynamic loads are marked to prevent speculation. The only benchmarks with significant numbers (greater than 0.1% marked) are: *gcc* (29%), *gzip* (4%), *parser* (2%), *vpr* (1%), and *perlbmk* (0.5%). The large number of loads marked in *gcc* are expected, due to the prevalence of the `rtx` data structure, which contains many involved unions.

Second, this approach is conservative in marking potentially wild loads for no control speculation whatsoever. It is likely that there is some specific control dependence, and not all control dependences on which the load depends, that enforces the salutariness of the address to be dereferenced. In the "operational" and "illicit offset" cases, it is likely possible to identify this specific control dependence. In the "union-field" case, one might make some good guesses about the meaning of control dependent on fields in the same or related data structures. This conservatism, however, was not addressed here, as it did not seem to cause serious performance limitations in practice.

It is the author's opinion that these wild load avoidance techniques, and simple adaptations of them, are fit for practical use and have demonstrated that means can be found of rendering general speculation safe to use without rendering it ineffectual. It is this a viable alternative for the Itanium platform that offers both compiler simplicity (aside from the problem of detecting potential wild loads) and a potential reduction in run-time speculation overhead.

6.7.3 Speculation of floating-point operations

Control speculation of floating-point operations such as multiply-accumulate (`fma`) operations is supported on the Itanium architecture. A detailed description of the provided mechanisms is given in [123], so this treatment is confined to a brief presentation. The support for floating-point speculation is by the provision of four floating-point control and status registers. Aside from the first, `sf0`, which is inherently nonspeculative, the other three can be set to disable various forms of exceptions. Each floating-point

operation specifies the status register with which it interacts. Those that report to registers with traps disabled may be control-speculated without fear of generating spurious *software-visible* floating-point exceptions. For programs that use such exceptions, control-speculative floating-point operations must be followed up with a floating point status check operation and re-execution code, not unlike the recovery blocks associated with complex load speculation patterns. For programs that do not concern themselves with floating-point exceptions, this check is unnecessary.

In the IMPACT environment, floating-point speculation is enabled by a flag in the microarchitectural machine description file (See Section B.11.2). Due to the nature of common usage, speculation of these operations most frequently occurs in scheduling. If floating-point speculation is enabled, IMPACT assumes that the program is unconcerned with floating-point exceptions and therefore generates no floating-point check-recovery sequences. For SPEC CINT2000 benchmarks, control speculation of floating-point operations was found to have little positive performance impact (only *vpr* and *vortex* appeared to benefit, and those by less than 2%). For *eon*, control speculation of floating-point had a disastrous performance effect, causing a 33% reduction in performance. This is a consequence of the speculation model for floating-point operations, which fundamentally adopts the general speculation approach. The various status registers allow for the deferral of *software-visible* floating-point exceptions, preventing undesired program termination or interruption from these events. They do not, however, permit the deferral of floating-point software-assist (FPSWA) events. These events, which require a trap to kernel code, supply complex corner cases in floating point computations which were adjudged too difficult and infrequently used to be put into the hardware units [57, 117]. The problems in *252.eon* occur when a loop containing a multiply-accumulate operation (`fma`) is modulo-scheduled. In the three-stage schedule, speculative loads (stage one) wind up feeding a speculative `fma` (stage two). As the last iteration of the loop is coming to an end, the `fma` tries to add values loaded from beyond the end of the array originally intended to be accessed; the result of this operation will never be used. Adding the values stored at these locations happens to require software assistance. Since the FPSWA is

slow (hundreds or thousands of cycles), even these relatively infrequent events carry a substantial performance penalty.

It should be pointed out that the particular problematic code sequences encountered in *eon* could have been handled without control speculation by using a counted-loop modulo scheduling schema with stage predicates. The IMPACT compiler does not support this mode at the present time; other compilers, such as Intel's, do. For true "`while`" (noncounted) loops, however, control speculation of floating point operations bears unresolved performance risk because of an inability to defer FPSWA events. For now, IMPACT's speculation of floating-point operations was disabled for *eon*. A dramatic increase in kernel time with control speculation, accompanied by an increase in SIR stalls in performance monitoring data [51], is a sure indicator of this problem.

The need for a "fast," "silent," or "flush-to-zero," mode has been noted at least since the implementation of speculation in the Multiflow system [56]. Colwell pointed out cases such as "`if (q != 0) r = d/q;`" that call for silencing of exceptions if speculation is to be allowed. This dissertation's experiments have simply shown that such a mode does not allow totally general speculation of floating point operations (without potentially substantial performance penalty) in a software-assisted floating-point architecture like Itanium's.

6.8 Performance Effects of Control Speculation *in situ*

Having explained the modes of control speculation applied in the IMPACT compiler, let us consider some empirical results. Control speculation is very effective in reducing the execution time of most of the SPEC CINT2000 applications, in both predicated and nonpredicated contexts, as shown in Figure 6.9. This is not a surprising result, as the primary motivation for creating Superblock and Hyperblock regions is to enable enhanced instruction scheduling. Such improvements in scheduling are much less prevalent without control speculation, since branches (on average, one out of seven instructions in **O-NS** code and one out of ten in **I-CS**) and potentially-excepting (not provably safe) loads abound in these applications.

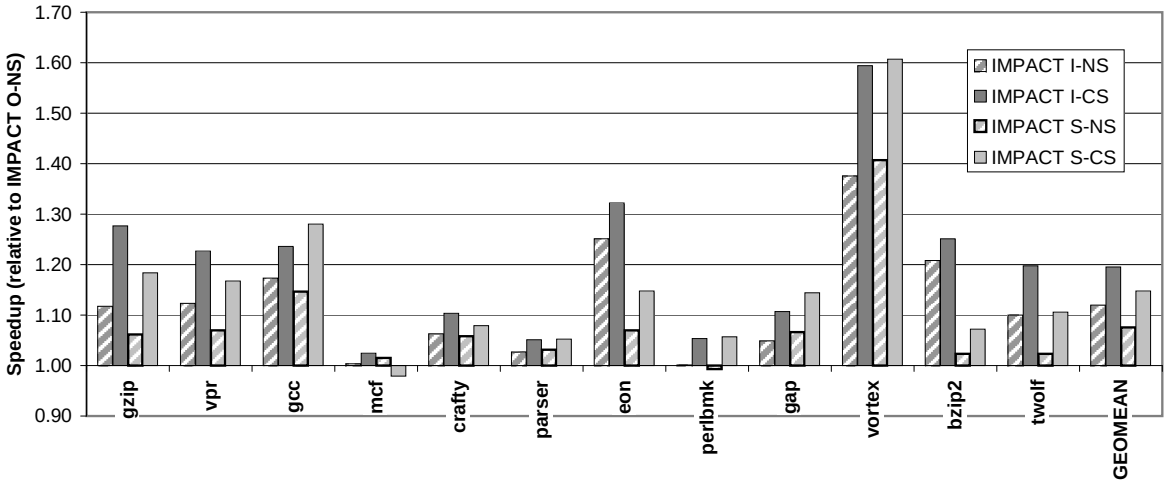


Figure 6.9 Effect of control speculation on performance.

S-NS and **I-NS** appear on average to receive approximately equivalent gains in performance when control speculation is added. Despite this, it is important to note that single-path and multipath specialization present somewhat different contexts within which control speculation can do its work. In single-path regions, generally only the path anticipated to be the most commonly executed is present, so speculative operations are generally very likely to be on-path. In multipath (Hyperblock) code, however, less-prevalent paths are also included. This poses the opportunity for increased issue of off-path speculative operation, and therefore greater potential side-effects, or penalties, from control speculation. Nonetheless, these effects are not materialized unless control speculation actually occurs, so this chapter on control speculation is the appropriate place to have this discussion.[18]

6.8.1 Effects on instruction issue

Figure 6.10 shows the effect of control speculation on the number of operations issued in the single-path (**S-NS**/**S-CS**) and multipath (**I-NS**/**I-CS**) contexts. While control speculation causes operations to execute more frequently than in the original program, these results show that IMPACT's speculation benefits reflect only a fairly selective

[18]Provably safe loads, as well as other nonexcepting operations, are control-speculated in the **I-NS** configuration. These could conceivably, but are unlikely to, have some performance effect.

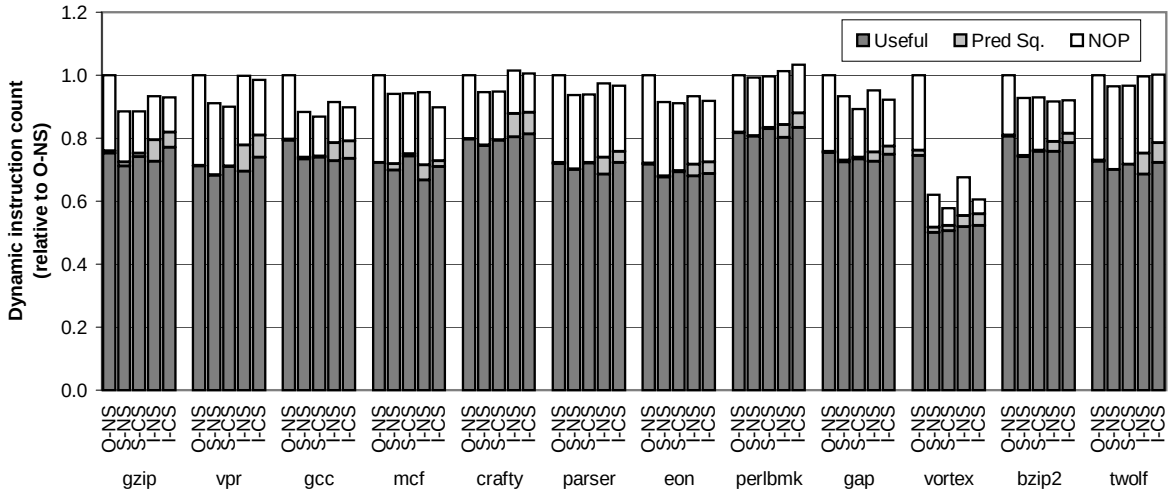


Figure 6.10 Effect of control speculation on dynamic instruction count.

speculation of operations (a pronounced rise in executed operations is not observed in the speculation-enabled, **S-CS** and **I-CS** versions). Likewise, relatively few operations wind up being predicated-off (p = 0), even though the Hyperblock formation being performed is much more proactive than typical commercial approaches. The number of "useful" instructions increases by an average of 2.7% (never more than 6.2%) between **S-NS** and **S-CS** and an average of 3.7% (never more than 6.4%) between **I-NS** and **I-CS** because of operation speculation, since in this context "useful" means "non-`nop`, p = 1" operations. This margin can be considered misspeculation, since on-path speculative operations would have executed in the nonspeculative version.[19] The speedups attained with this small degree of misspeculation reflect selective region formation and a careful use of execution bias.

6.8.2 Effects on data access

Some of the most interesting (and confounding) interactions of CFS transformation are with the data delivery infrastructure of the Itanium 2. Given the prominence of data cache and data access-related latency in cycle accounting results for this processor, the significance of this interaction is not a surprise. Interestingly, although the compiler

[19] Of course, this coarse measurement discounts those opportunities for optimization exposed by control speculation which may have somewhat reduced the number of operations in the control-speculative versions.

fails to make a concerted effort to use the slack it purchases to schedule loads to miss, free control speculation of loads and their dependents does not (at least as a first-order performance characteristic) increase the time spent in data-related stalls. This is in itself an interesting result, as previous work [38] predicted that a substantial proportion of cache misses would be due to off-path speculative operations. Chapter 8 describes these issues in detail.

6.8.3 Interaction with predication

As noted earlier, control speculation can interact positively and negatively with predication. It can reduce the dependence height of paths enclosed in a predicated region by relaxing control dependences, rendering paths more compatible than in a non-control-speculative version, but it can also expose commonly off-path instructions as sources of execution scoreboard stalls, spurious cache and address translation misses, or even spurious page faults. An evaluation of the data shows the positive effect to predominate, and strongly so. This is clear at the benchmark level, as the application of control speculation to predicated code (comparing **I-NS** to **I-CS** configurations) tends to improve, and not degrade, performance, and it generally delivers performance equal to or better than the nonpredicated **S-CS** configuration.

Sampling data collection allows the gathering of finer-grained, function-level data. This provides an expanded number of distinct samples, within which to compare the performance effects of, and evaluate the interactions of, the various techniques.[20] Within this set, comparing **S-NS** to **I-NS** execution time, 13 prominent examples appear in which the application of predication alone (without control speculation) degrades performance by 5-20%. With the application of control speculation (to both Superblock and Hyperblock code), seven of these deficits are eliminated, and one prominent one (accounting for 3% of *vortex*'s execution time) is cut in half. This demonstrates control speculation's ability to reduce the dependence height and improve the scheduling compatibility of predicated regions.

[20] Although radical procedure inlining reduces this number of samples, it still consists of 99 distinguishable samples, each representing more than 1% of program execution time, instead of 12.

There are also examples of procedures whose performance is degraded with the application of control speculation, including `Swap()` in *crafty*, the performance of which is 5% worse in **I-CS** than in **I-NS**. This results from two factors: first, the number of registers used is increased due to increasing ILP, and the extra allocation causes more RSE activity. Second, the scheduler, with increased freedom purchased by control speculation, delays important branches to improve the predicted main path. Situations like this are rare (only this function and one in *vpr* are retarded by control speculation, and the *vpr* example is only incidentally related to control speculation). Given the ability to avoid wild loads, general control speculation is a stable and predictably beneficial tool in enhancing ILP, especially in predicated code.

6.9 Data Speculation

The Itanium architecture [57] and EPIC systems in general [73, 124, 125] support a second type of explicit speculation, *data speculation*, the speculative placement of loads before potentially aliasing, logically previous stores. The IMPACT compiler does not currently make use of this feature, although its potential has been studied in the context of IMPACT's optimizer, both in initial EPIC research [38] and in limited experiments on Itanium. Its implications will therefore be considered only briefly here. Since, unlike control speculation, data speculation requires recovery code and since, like control speculation, there is a tangible cost to misspeculation in the application of data speculation, such use would require careful analysis of profitability and risk in the compiler.

Although IMPACT's aggressive pointer analysis reduces the benefit IMPACT-compiled code could derive from data speculation (perhaps in contrast to production compilers), the authors do observe many opportunities for data speculation. The benchmark *gap*, in which pointer analysis is unable to resolve critical spurious dependences in otherwise highly parallel loops, appears particularly promising. A limited initial application, currently in progress, is providing a 5% speedup; much more is attainable. Aside from mitigating deficiencies in alias analysis, data speculation can also allow the compiler to manage even "known-sometimes" dependences. Other researchers have shown opportunities to exist for profitable integration of data speculation into optimizations [64].

7 PROCEDURE INLINING AND EPIC PERFORMANCE

Procedure inlining is generally an important prerequisite for achieving useful levels of instruction-level parallelism in the SPEC CINT2000 benchmarks. Relative to a compiler that performs only basic-block instruction scheduling and classical optimization, an ILP compiler like IMPACT is far more reliant on procedure inlining to achieve high performance. This was documented using SPEC95 benchmarks in [74]. Procedure inlining provides three basic benefits: (1) elimination of call mechanism overhead; (2) specialization (optimization) of the inlined body for its calling context (the classical benefit); and, (3) the effective formation of instruction-level parallelism across call sites (a benefit particularly required for EPIC machines). This chapter documents the inlining performed in the described experiments, as well as the modifications made to the IMPACT Pcode profiler and inliner [46, 74] to make them capable of delivering satisfactory results across in SPEC CINT2000 suite.

IMPACT, in this experimental context, is used as a whole-program optimizer, with profile-guided optimization. This bears two important implications for the inliner: first, it is able to inline calls across file boundaries without distinction;[1] second, it is able to use control profile information to bias inlining in favor of more frequently executed call sites.

[1]Intel's compiler for Itanium has this capability if run with `-ipo`; GCC presently does not.

7.1 Controlling the Inliner

Since phase ordering considerations generally constrain inlining to be done early in the course of compilation, well before the needs and consequences of subsequent transformations can be evaluated, control of inlining can pose a challenge. Procedure inlining in the IMPACT compiler depends on profile information to expand call sites in priority order ($priority = \frac{execution\ weight}{\sqrt{callee\ size}}$), at most until the amount of touched code is doubled (the expansion ratio of 2.0 is an empirically determined value; because this transformation is performed at the high level, this is only approximate).

For a given application there is some degree of permitted inlining, assuming that it does not become fully inlined before such a point, at which further inlining causes more degradation (due to instruction cache effects relating to the application's increasing footprint) than benefit (due to additional, enabled CFS transformations or specialization-based optimizations). At this point, performance begins to suffer. This effect was just beginning in one application, *crafty*, at the inlining ratio selected. Beyond this ratio, only one application, namely *vortex*, appears to continue to benefit from inlining. This ratio, therefore, was selected because it tended to provide enough inlining to achieve good levels of ILP while not unduly impacting instruction cache performance across the suite. All benchmarks except for *vortex* and *perlbmk*, however, achieved approximately equivalent results with half this inlining code growth threshold. Reducing the ratio did not substantially aid those benchmarks, such as *crafty*, in which instruction cache miss is a significant performance component.

7.2 Indirect Call Site Transformation

In the past, only direct call sites (those calls to specific, named procedures, rather than through function pointers) were accessible to IMPACT's inliner, since callees were known only at direct call sites and since multiple callees might be invoked from a given, single direct call site. This limitation proved unsatisfactory for the C++ benchmark *eon*, for which indirect function invocations are frequent (but each call site usually has one or a few typical callees). This is typical of C++ and also of some object-oriented

```
1 fptr = farray[i];
2 (*fptr)(arg1,arg2);     // Call frequencies: foo():66% bar():34%
```

(a) Initial code with profile information

```
1 fptr = farray[i];
2 if (fptr == foo)        // Profile: 66% taken
3   foo(arg1,arg2);       // Direct call site ready for inlining
4 else
5   (*fptr)(arg1,arg2);   // Call frequencies: bar():100%
```

(b) Transformed code, ready for inlining

Figure 7.1 Conversion of indirect to direct call site for inlining.

programming styles in C. To remedy this deficiency, first, profiling of indirect function invocations was added to the Pcode control flow profiling stage. Then, once information was available about which were the frequent callees at a particular indirect function invocation, a transformation was made available to the inliner to extract direct call sites from indirect ones. This transformation is shown in Figure 7.1. It is highly effective in allowing inlining across indirect function invocations (removing 88% of dynamic indirect function calls) for *eon*, enabling one of the highest rates of call removal in the benchmark suite. The interpreter *perlbmk* also benefits substantially from this indirect call site inlining transformation, as does the compiler *gcc*.

This approach to indirect function call inlining is relatively general and easy to implement, but does come with the overhead of runtime function pointer comparisons, which may include accesses to the linkage table (the address of `foo` in Figure 7.1 may not be a link-time constant). It is a better solution, though applicable only in certain instances, to avoid the function pointer comparison by using other mechanisms to determine whether the specialized, inlined version is appropriate in a certain instance. In C++, for example, there are several well-developed algorithms for fast static analysis of virtual invocations [126]. These techniques can prove that a certain call site is limited to one or a handful of callees and identify the prerequisite values that dictate which callee should be invoked, eliminating the need to handle function pointers.

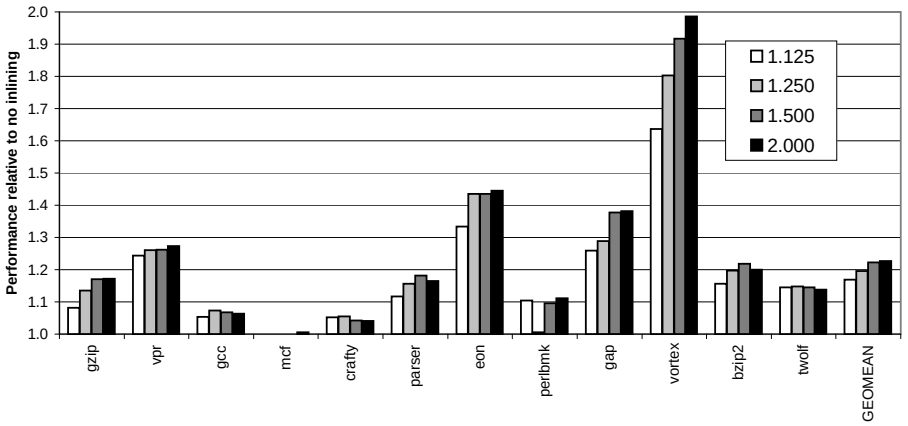


Figure 7.2 Variation of **I-CS** performance with inlining degree.

7.3 Effect of Inlining

Procedure inlining contributes handsomely to the performance of applications compiled with IMPACT. Figure 7.2 shows the speedup that results from **I-CS** compilation with various degrees of inlining ranging from none to 2.0, the ratio used in the experiments reported here. (Recall that the same degree of procedure inlining is applied in all configurations of the compiler, but that inlining may disproportionately benefit code with CFS transformation.) Increasing inlining generally continues to increase performance for a perhaps unexpectedly large range of code size increase. Only in *crafty* do instruction cache limitations start to detract meaningfully from performance at an expansion ratio of 2.0. The consistently unimpressive gains in *mcf* are due to its domination by data cache stall time; *perlbmk* has one anomalous result of unknown origin. Inlining is clearly of great importance to achieving substantial gains from CFS transformation.

The effect of inlining on the benchmarks is briefly summarized in Table 7.1. The (dynamic) number of instructions per subroutine call instruction, before and after inlining, is indicated in the first two columns of data. The third column indicates the proportion of dynamic subroutine calls removed by inlining, and the final columns indicate the degree of total and touched (instructions executed at least once) static code expansion associated with the inlining decisions applied.

One may note that the code expansion ratios indicated in Table 7.1 vary, but are significantly smaller than the 2.0× target ratio indicated above. This is for two reasons:

Table 7.1 Inlining statistics

benchmark	instr. per call		% calls	code expansion	
	pre-inlining	postinlining	inlined	all	touched
gzip	175	3567	95.1%	1.45	1.41
vpr	69	2708	97.5%	1.43	1.21
gcc	127	245	51.7%	1.56	1.36
mcf	34	4556	99.3%	1.76	1.48
crafty	82	384	78.7%	1.51	1.57
parser	38	232	83.5%	1.44	1.46
eon	53	755	93.0%	1.50	1.61
perlbmk	77	260	70.3%	1.49	1.22
gap	56	322	82.6%	1.49	1.49
vortex	35	502	93.8%	1.54	1.34
bzip2	95	4243	99.1%	1.57	1.41
twolf	160	11446	98.6%	1.34	1.37

First, the inliner, which operates at the Pcode level, only estimates code size expansion, while these numbers reflect actual instruction counts (in Lcode, after classical optimization). Second, the inliner does not account for a reduction in touched code size for procedure bodies when said bodies have no remaining (non-inlined) call sites. This is an intentional behavior, in deference to the possibility of deviation from the training profile that results in new call sites becoming active.

7.4 Potential for Improvements in Inlining Techniques

Despite its seeming good behavior here, inlining does present some interesting problems for the EPIC compiler. At the outset of this dissertation work, the interaction between inliner and ILP transformations seemed like a potentially fruitful, if also potentially challenging, field of study. As it turned out, however, practical applications of this style of technique are limited, at least within the body of SPEC CINT2000, and pressures on instruction cache resources are not sufficiently high or dependent on inlining to render interesting experimentation possible. As indicated in Section 3.5, the first-level instruction cache hit rate only once, in *crafty*, drops below 90% and only twice below

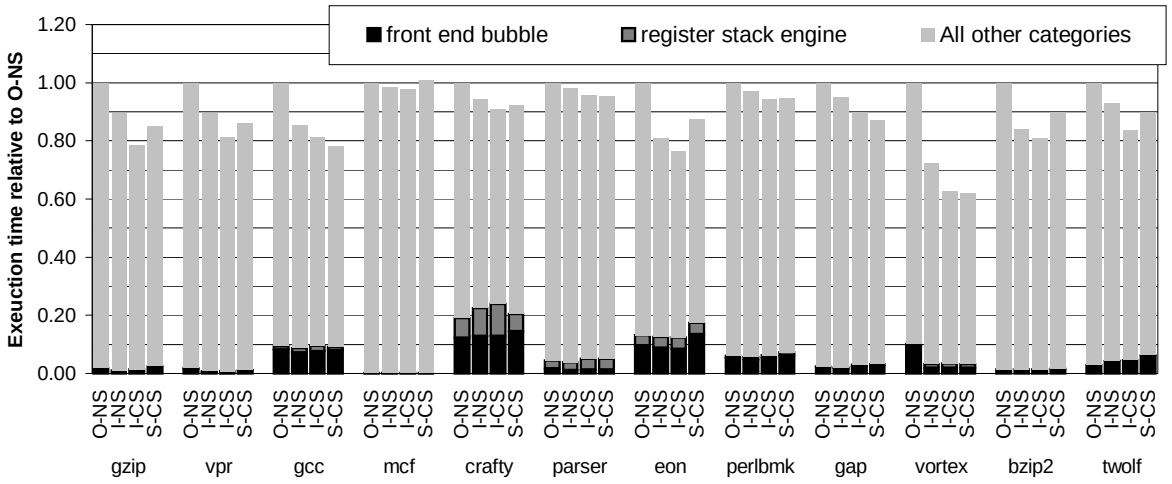


Figure 7.3 Execution cycles sensitive to inlining penalties.

95%, in *eon* and *twolf*, even with the aggressive and only lightly constrained approach applied here.

Figure 7.3 shows the contribution to performance of the two stall categories that can be attributed to the negative side-effects of inlining, front end bubbles (mainly instruction cache misses) and register stack engine activity. In few benchmarks are these cycles major components of the performance equation. Nonetheless, it is important to consider a few problems or opportunities in inlining that were exposed by this work.

7.4.1 Inlining and register stack engine activity

First, let us deal with the register stack engine, which plays a prominent role in *crafty* and *eon*.[2] Inlining has mixed effects on register stack engine (and register spill/fill) activity. By rendering procedure invocations less frequent, it can tend to reduce the activity of the register stack. It does, however, tend to increase the size of register stack activation records due to the inclusion of more code in each function, since IMPACT performs a single allocation for the entire function body. This can in some cases increase the amount of register stack activity. In *crafty*, for example, the procedure `Quiesce()`,

[2]Comparing with `icc` results, IMPACT usually spends less time in RSE activity, due to much more aggressive inlining. (`icc` also supports partial inlining, which enables it to amortize inlining costs differently.) IMPACT causes substantially more RSE activity than the production compiler in *crafty*, *parser*, and *eon*.

which has low register allocation needs, calls `Evaluate()`, a procedure with heavy register utilization, and then recurs on itself. When inlining heuristics cause `Evaluate()` to be inlined into `Quiesce()`, the size of the register stack allocation performed in the recursion is greatly increased, leading to increased activity. A similar case occurs in *parser*.

Since inlining is performed long before the register utilization needs of a function are understood, it is difficult to control this kind of behavior in inlining heuristics. Curtailing inlining in strongly connected components of the call graph (even by applying the global code expansion limit locally) sharply curtails this effect, cutting the number of cycles spent in register stack engine activity by half in *crafty* and by three-fifths in *parser*. In the case of *crafty* and *parser*, this improves performance slightly (2%). In *eon*, performance is degraded by 2% due to an *increase* in register stack engine activity (profitable opportunities for activation record combination are being squandered). Finally, the performance of *vortex* is severely degraded (loss of 16%), as the reduction in inlining performed decreases subsequent scheduling and optimization freedom. These difficulties suggest an opportunity for solutions, such as partial inlining, that do not require inlining decisions to be made only at the procedure granularity, and suggest that inlining should for best results be performed in the context of knowledge of subsequent, potential ILP transformations.

7.4.2 Control of inlining degree

Inlining takes account only of the desire to eliminate calls in its decisions about the desirability of inlining functions, subject to some total code size expansion limitations and constraints on function size to prevent optimizer overload. These are totally artificial guidelines, designed to prevent inlining disasters—not to carefully craft good decisions. Inlining does not consider whether the available instruction level parallelism will actually be improved by the inlining transformation or to what degree a function body may be specialized for a particular call site; thus, inlining may be expanding code size for no gain.

Inlining considers only *static* code expansion to be a detriment to performance—not "*dynamic footprint*" code expansion. The latter is actually the better metric. For example, the inlining of a callee function at several call sites in a loop will impact the loop's instruction cache footprint, potentially endangering processor front-end performance, while inlining a function at several call sites "far apart" in the course of program execution will have a lesser effect. Inlining should consider the interprocedural control flow graph of the program in performing its transformations to trade cost and benefit more effectively. McFarling noted this limitation, provided a useful summary of previous work on inlining, and proposed a new heuristic solution [127]. This solution takes into account the instruction-count cost of performing the call and return and the expected cache miss cost of the required code expansion. Today, in the context of ILP optimization, the problem is more complex, but the basic idea of this work, that making better inlining decisions requires knowledge of program context, remains valid. Work like McFarling's may become relevant for larger programs with less inherent instruction locality.

Finally, inlining considers a procedure as a unit rather than supporting the partial inlining of hot paths of a procedure only. Partial inlining could achieve the same benefits with reduced cost, both statically and in the "dynamic footprint" sense, if it produces code with less lukewarm code expansion. It could also reduce the potential negative impact of inlining on register stack utilization in procedures with inlined callees, if the partial nature of the inlining allows infrequently used regions of the callee that nonetheless use many registers to be excluded.

7.4.3 Programming and phase ordering problems in inlining

EPIC provides both unique motivations and unique accommodations for unusually aggressive control flow transformations. An example clarifies this point. Figure 7.4(a) shows a stylized representation of the procedure inlining transformation. On the left, a subroutine call is invoked from within a loop. The compiler decides to inline the subroutine body into the loop. Such inlining may, for example, allow the loop to be modulo-scheduled, extracting instruction-level parallelism by executing multiple loop iterations in a pipelined fashion. Figure 7.4(b) is similar, except that here the loop's control flow

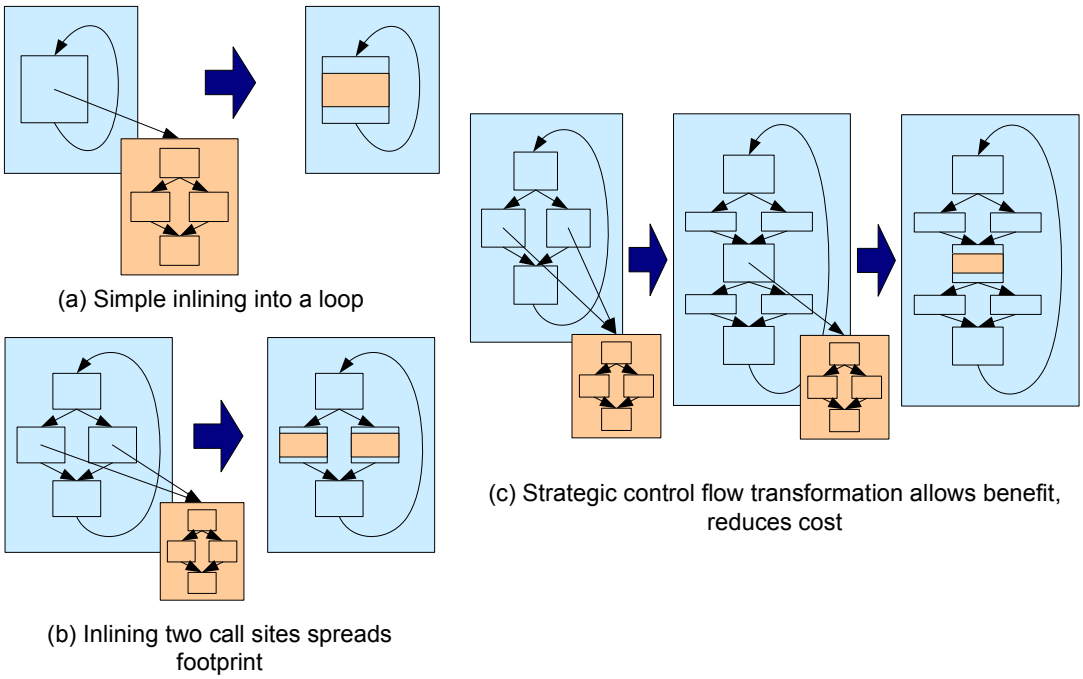


Figure 7.4 Inlining, control flow transformations, and concise program expression.

contains two calls to the same subroutine. If both paths are frequent, and if a sufficient inlining budget is available, IMPACT will inline them both, creating two copies of the callee code inside the loop. Inlining both, however, may be inferior to the solution in Figure 7.4(c), in which a control flow transformation is applied to inline a single version of the callee. This approach inserts new control flow to reduce code size, ordinarily a risky proposition. Since EPIC provides if-conversion as an option for subsequently removing the added control flow, however, this may be a profitable approach.

7.4.4 Program structure and procedure inlining

As the control flow structures selected by the programmer do not always lend themselves in the best way to ILP creation by function inlining, transformations to reduce the number of active call sites should take place prior to inlining. The example of Figure 7.5, taken from the benchmark *crafty*, demonstrates this point. Here, within the body of a single, simple loop, are 10 calls to the function `FirstOne()`, of which at most one will be invoked on any given traversal of the loop. Trying to accelerate the loop by inlining all, or even some, of these calls will result in massive code replication for arguably little

```
1   while (attacks) {
2     if (color) {
3       if (And(WhitePawns,attacks))
4         square=FirstOne(And(WhitePawns,attacks));
5       else if (And(WhiteKnights,attacks))
6         square=FirstOne(And(WhiteKnights,attacks));
7       else if (And(WhiteBishops,attacks))
8         square=FirstOne(And(WhiteBishops,attacks));
9       else if (And(WhiteRooks,attacks))
10        square=FirstOne(And(WhiteRooks,attacks));
11      else if (And(WhiteQueens,attacks))
12        square=FirstOne(And(WhiteQueens,attacks));
13      else if (And(WhiteKing,attacks))
14        square=WhiteKingSQ;
15      else break;
16    }
17    else {
18      /* Repetition of above, but for black pieces */
19    }
20    ...
21  }
```


Figure 7.5 Code example from *crafty* `swap.c`.

benefit. A simple control transformation could reduce this loop so that it contains a single call to `FirstOne()`, shared among the mutually exclusive cases. In theory, then, this single instance could be partially or wholly specialized for a subset of the different cases, if necessary. Attempts by the author to make such profitable transformations by hand have, however, been unsuccessful, for two reasons. First, performance relies on specialization for particular cases, and combination of call sites destroys context-specific execution bias. Second, the combination of paths to a single call site adds dependence height because of the need to select and arrange input operands. This dependence height is not successfully hidden because the loop is serially dependent on the value produced by `FirstOne()`. It appears that these optimization-hampering effects reverse gains achieved through a reduction in inlining overhead. These difficulties, though, do not detract from the point of the example, which is to show the awkwardness of CFS transformation when applied to some common programming idioms.

To summarize, inlining should take place later, when more knowledge is available about the context of call sites, and should be guided by new heuristics that take into

account the primary benefits and primary costs of the transformation. The importance of getting this right is indicated by the fact that variation of inlining parameters today results in wide and difficult to anticipate performance deflections. Problems with inlining have so confounded some compiler researchers that they have proposed radical alternative models for thinking about interprocedural optimization. One approach [128] advocates the elimination of procedural boundaries, effectively allowing the compiler to treat an entire program as one function containing some unorthodox control flow elements. The compiler then would operate on this representation, specializing and forming regions as if forming Hyperblock regions in a normal control flow graph. Though the same fundamental questions are still not answered in this work—namely, when specialization is truly advisable—it at least presents an optimization model in which the high and difficult-to-manage cost of inlining is not incurred speculatively, early in the compilation process.

7.4.5 Effect of inlining on profile accuracy

IMPACT currently performs two profiling runs; one in the Pcode intermediate representation, prior to inlining, to guide inlining decisions and classical optimization; and the second in the Lcode IR, after classical optimization. This second phase guides region formation, ILP optimizations, register allocation, and instruction scheduling. The Pcode profiler provides control flow, loop profile, and call graph (call edge weights, including indirect procedure calls) profile information; the latter provides the first two categories of profile information only (since interprocedural optimizations are performed prior to the Lcode phase, there is no need for improved call graph profiling). Profile information is context-insensitive; that is, profile weights resulting from different invocations of a particular procedure are not distinguished. This can have an effect both during inlining and subsequent to inlining.

Given context-insensitive profile information, inlining decisions require arbitrary redistribution of call weight across newly created call arcs. Figure 7.6 shows a contrived, but representative, example. Figure 7.6(a) shows a small call subgraph, with procedures

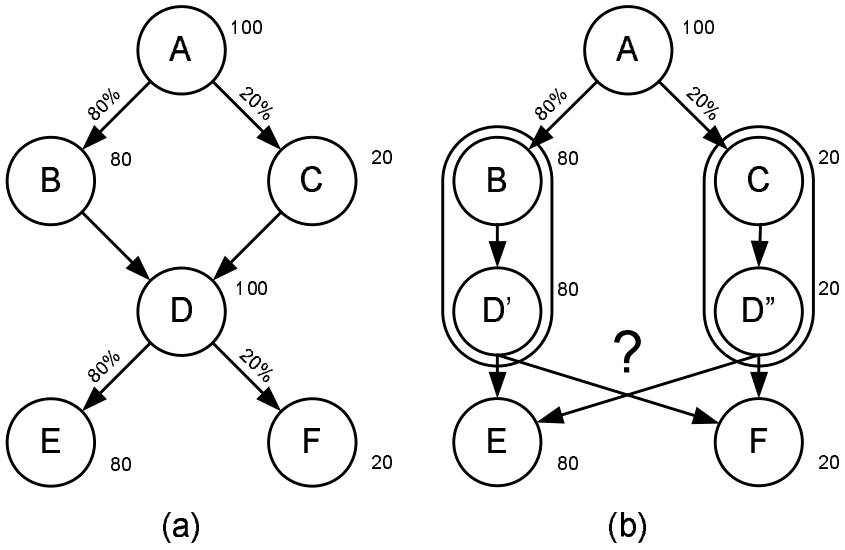


Figure 7.6 Specialization and inlining: (a) before inlining; (b) after inlining.

marked with execution weight. Procedures A and D each contain a single indirect function call which, 80% of the time calls B or E, and 20% of the time calls C or F, respectively. B and C each directly invoke D. Now suppose that D is inlined into both B and C. The effect of this transformation is shown in Figure 7.6(b). The compiler lacks contextual information about the callee successors of D, and so must arbitrarily assign weights to the arcs from D' and D" to E and F. With respect to the less-frequently traversed D", there is no guarantee that the compiler's uniform distribution of weight will even indicate the correct majority callee. This behavior is frequent in *gap*, an arithmetic interpreter that recursively evaluates input expressions. Inlined callees thus frequently execute in a different, higher "level" of input expression trees and have vastly different biases than their lower-level parents.[3] No context information is available to counter this effect, so later inlining decisions are often based on poor estimates of program bias.

The effect of using the profile estimated during inlining, rather than a correct one, for subsequent stages was measured. Table 7.2 shows the net (percentage) degradation in performance that results from skipping the second profiling stage. The data indicate that the code versioning inherent to procedure inlining (where multiple inlinable call sites

[3]Stating the problem in these terms causes one to wonder if such specialization is appropriate in a static compilation environment—interpreting programs like *perlbmk* and *gap* are truly specialized for specific input scripts.

Table 7.2 Performance loss due to omission of post-inlining profile, **I-CS** configuration

benchmark	Pct. perf. loss
gzip	5.2%
vpr	0.0%
gcc	0.0%
mcf	0.0%
crafty	8.6%
parser	1.6%

benchmark	Pct. perf. loss
eon	5.0%
perlbmk	18.5%
gap	2.8%
vortex	0.7%
bzip2	1.5%
twolf	0.0%

exist for a particular procedure) occasionally creates significantly disparate versions. In one or more of these versions, the estimated profile does not accurately reflect execution bias.

These issues suggest that more attention needs to be focused on context in profiling. Profile maintenance is a difficult problem, not only for the inliner, but also for other code-specializing techniques (such as the tail duplication inherent to Superblock and Hyperblock formation). The author has encountered situations similar to those indicated in Figure 7.6 in Superblock formation. Ideally, if path-sensitive[4] information were available to the region former, it could aid in region formation decisions as well as subsequent profile maintenance.

Some recent work in the just-in-time compilation community has addressed these issues [129], but static compilation as yet has no access to context-sensitive profile information. Efficient path profiling techniques [130–132] may offer some means of obtaining this type of information.

[4]Path profile information on the intraprocedural control flow graph is analogous to context information in the call graph.

8 THE DATA DELIVERY SUBSYSTEM AND EPIC COMPILATION

The "EPIC fundamentals" studied in this dissertation did not occur in a vacuum, but rather in the context of a complicated, modern microprocessor and a complete compilation environment. Experiments in which a better theoretical use of speculation or predication was made often ended in performance degradation. This chapter documents a few of the microarchitectural and compiler phenomena that interacted with the use of EPIC features in interesting ways. One is tempted to believe that these issues are relevant only to the Itanium 2 microarchitecture, and that this chapter is therefore superfluous. The point, however, is that the seemingly benign irregularities of a microarchitecture or compiler system can easily disrupt an apparently good plan of execution. This is a serious problem for EPIC performance in general, and EPIC compiler experimentation in particular. Without the benefit of a statistically significant experimental data set (results from many different code segments with genuinely different behavior), these effects can easily taint results and lead to false conclusions about approaches to fundamental issues. Some of these complexities (such as the exposure of latencies due to interaction of loads and stores in the memory subsystem) are probably unavoidable in a modern architecture. EPIC compiler writers will have to continue to wrestle with these issues, some of which are difficult to anticipate at compile time.

Schlansker et al., in their seminal pre-Itanium evaluation of the prospects for EPIC system performance, made the following hopeful pronouncement: "Our analysis indicates

that the behavior of individual load operations in integer as well as floating-point benchmarks is favorable to compiler-directed cache management." They note the bimodal behavior of loads, by which certain loads rarely miss, while others do with some regularity, and the typically small number of loads responsible for a large majority of stall time. This optimistic assessment was based in part on work, such as [53], that suggested that loads that were likely to miss could be identified, and that they could either be prefetched effectively or simply scheduled to accommodate miss latencies. They do admit "though our work indicates that miss-sensitive scheduling can be an effective compiler technology, there are still issues left to be resolved. First, missing loads need to be identified prior to or during compilation... Secondly, it is not clear whether adequate parallelism and scheduling flexibility is present to schedule missing loads with the cache-miss latency in integer applications" [4].

The work presented here confirms these concerns. The following brief sections document the general problem with the magnitude and predictability of data access latency for CFS transformation and the various means undertaken to mitigate it in the compiler. Casual, performance analysis-guided experiments with scheduling loads to miss generally met with very little success, but several other problems yielded to a combination of analysis and transformation. The most difficult problem, the management of varying data access latency, however, is likely to require a microarchitectural solution for nonnumeric applications. Examples of these approaches are briefly surveyed in Chapter 11.

8.1 Effects of CFS on Data Cache Performance

CFS transformation has the potential to increase pressure on the data cache and translation hierarchy. Control speculation results in more loads being issued. Speculation and region formation collaborate to bring this increase number of loads into a condensed number of cycles. Simply because of the increased temporal proximity and increased number of memory access events, one would expect to observe some degree of problems, as queues become more full and as accesses become more collision-prone. It is, in fact, possible to observe in the data evidence of these effects, although the magnitudes of change in load number and distance seem small.

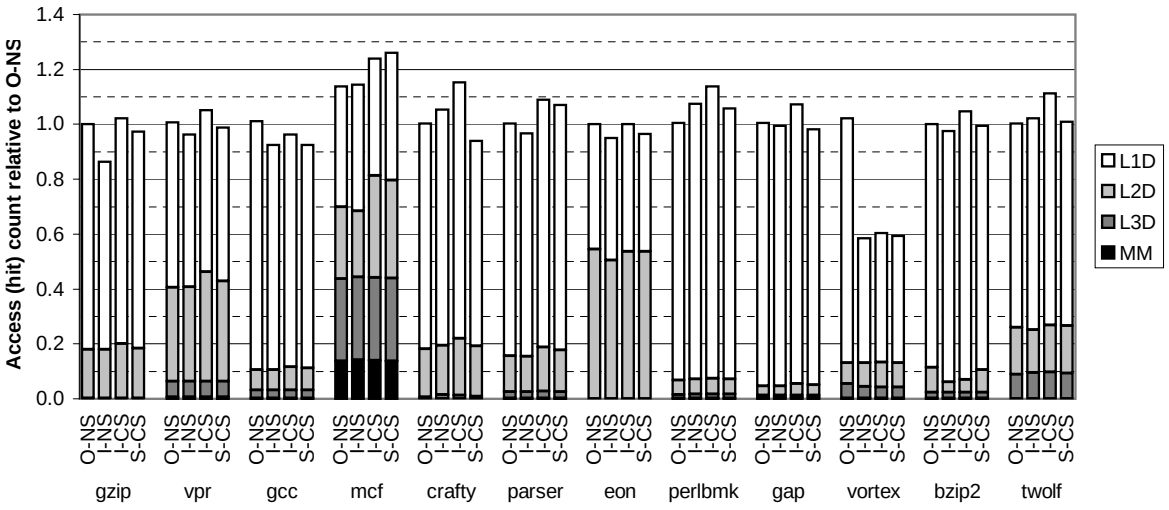


Figure 8.1 Effect of CFS transformation on data access.

In the Itanium 2, while the first-level data cache is fully dual-ported, the second-level cache is arranged into sixteen, 16-byte-wide banks. A second access to a bank in a given cycle incurs a penalty of up to several cycles, as the second load must be recirculated in the access queue. The fact that the first-level cache's four-bank-wide fill requests also cause access conflicts further complicates the problem. The Itanium 2 Processor Reference Manual documents these and other related memory subsystem interactions that can impact performance, many of which are beyond the compiler's purview [51].

The effect of increased load density due to CFS was found to be small and sometimes reducible. The juxtaposition of stores and loads turned out to be a much more serious matter, a multipartite solution to which was necessary to make the CFS techniques profitable for most benchmarks.

Figure 8.1 shows the number of loads executed in the **O-NS** and CFS configurations, normalized to the **O-NS** value. In addition, within each bar, the proportion of accesses satisfied in each level of the data cache hierarchy is indicated. Control speculation results in an average increase of 8% in the number of loads executed (comparing **I-NS** and **I-CS** results). Off-path control speculative loads are therefore relatively infrequent. When the number of accesses satisfied below the L1 cache is considered, the increase in these likely-to-degrade-performance events is 11%. Control speculation causes, however, almost no effect on accesses to the L3 level or to main memory.

When the number of cycles spent stalled on memory access is considered, however, only a 1% average increase can be detected in the difference of **I-NS** and **I-CS** results. The benchmarks *crafty*, *gap*, *vortex*, and *twolf* exhibit net reductions in integer load stall time in the speculative configuration, presumably due to schedule slack incorporated between loads and consumers, given the better scheduling freedom control speculation provides.

Other benchmarks suffer. *Gzip*, *mcf*, *parser*, *eon*, and *bzip2* suffer a few percent increase in integer load stall time each; *vpr* suffers a whopping 15%. Detailed results appear to show that increased conflicts in the memory subsystem (bank conflicts in the L2, cache fills conflicting with new requests, memory access queue interactions, etc.) are more often the cause of degradation than the access of data not in the cache, or the pollution of cache data, by speculative loads. More loads in fewer cycles implies a higher level of interaction among the loads, and this interaction comes with a performance penalty. *Vpr*, for instance, in the control-speculative version, has two loads to adjacent fields, both likely to be serviced in the L2, juxtaposed in the same cycle. This leads to a bank conflict and several cycles of penalty. *Mcf* suffers from the compaction of spatially nearby loads and stores in a modulo-scheduled loop. Although measures are taken to counteract the performance degradation due to spurious store forwarding, they are not completely effective in this case.

Some of these penalties are reflected as cache misses; others as L1D micropipeline stalls. These are conflict-related cycles detailed in Appendix A.6. These cycles are difficult to anticipate and manage in the compiler.

8.2 Effects of CFS on Data Address Translation

Data address translation latency is frequently a prominent contributor to performance loss in this dissertation's experiments. Figure 8.2 shows the rate of first-level data TLB misses (including only accesses that hit in the L2 TLB) for the various configurations,

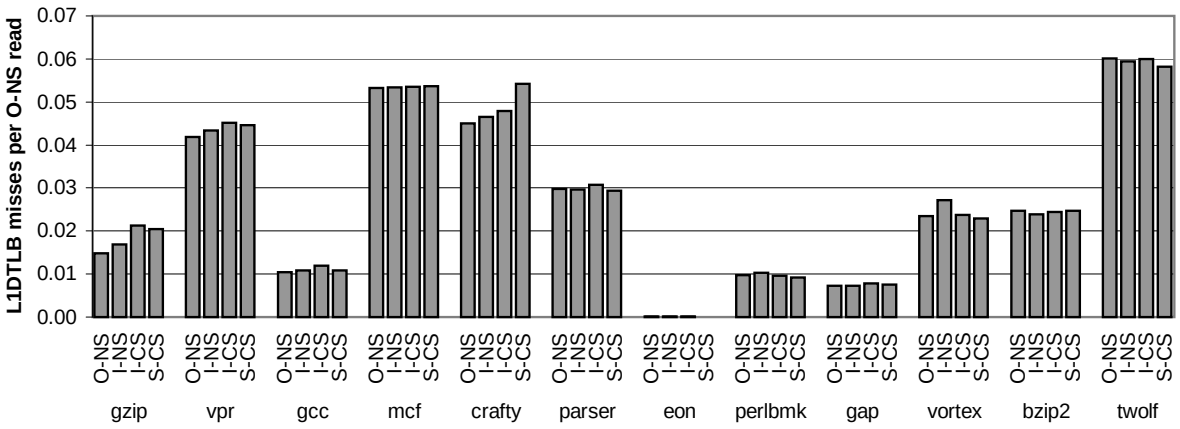


Figure 8.2 First-level data TLB misses (hitting in L2DTLB) per **O-NS** data access.

normalized to the number of data accesses in the **O-NS** configuration.[1] Since each first-level DTLB miss equates to at least a 9-cycle data access latency (4 cycles for the L2 TLB access and 5 cycles for the L2 cache data access), these rates sometimes substantially increase the expected latency of loads. Figure 8.3 shows the corresponding rate for those accesses that proceed to miss in the L2 TLB and require operation of the hardware page walker (HPW). These accesses are more costly, involving at least a 25 cycle penalty. Pages not found by the HPW are referred to the kernel. These *very expensive* operations are, fortunately, also very infrequent.[2] Details of Itanium 2's data address translation hierarchy are provided in Appendix A.1.

Control speculation causes freer execution of load operations. We can observe, in the data of Figure 8.2, small increases can be observed in the number of data TLB misses with speculation in *gzip*, *vpr*, *gcc*, *crafty*, and *parser*. There is generally much less change in the number of accesses required to the more expensive HPW mechanism. Figure 8.4 shows the estimated contribution of these events to benchmark execution time, as a fraction of **O-NS** execution cycles.[3] Control speculation indeed increases the occurrence of these

[1]With the exception of *crafty*, in which speculation disproportionately increases the number of "hit" accesses, and *vortex*, in which optimization disproportionately removes them, this is approximately the data TLB miss rate for all the configurations. In *crafty*, the **I-CS** rate is 0.04 and the **S-CS** rate approaches 0.06; in *vortex* the two control-speculative configurations have a rate of 0.04 data TLB misses per data access.

[2]Without wild load mitigation measures (Section 6.7), however, these effects are quite substantial, for example, increasing the execution time of *176.gcc* by 25%.

[3]In other cycle accounting graphs, such as Figure 2.6, these cycles are distributed among the L1D/FPU micropipeline stall and load bubble categories.

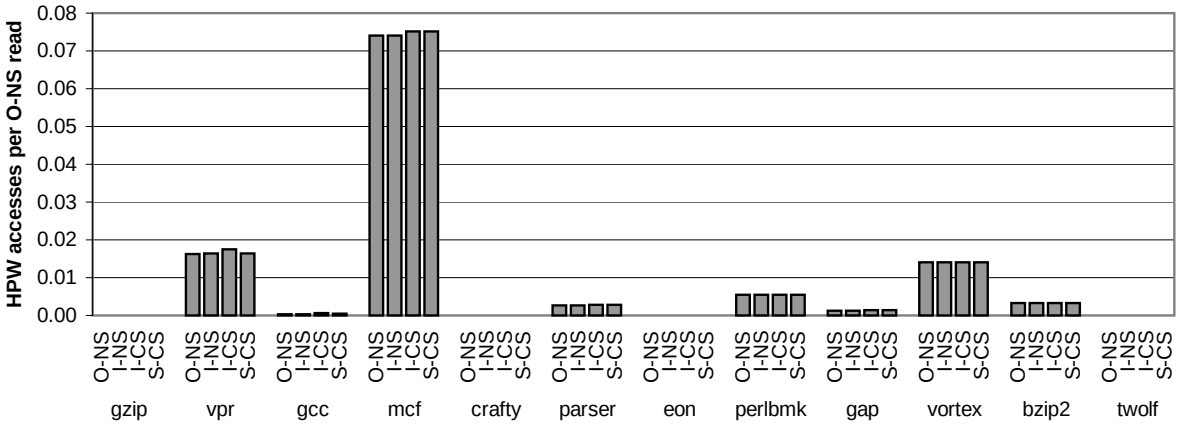


Figure 8.3 Second-level data TLB misses (HPW accesses) per **O-NS** data access.

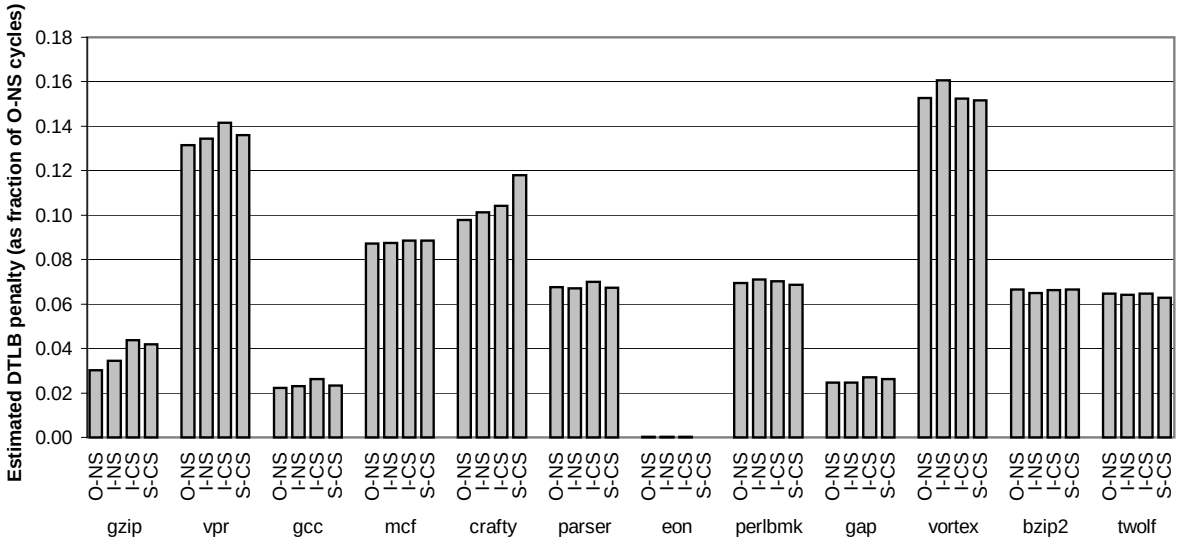


Figure 8.4 Estimated DTLB penalty as fraction of **O-NS** cycles.

events, but generally only by a small degree. The largest change (a 20% increase in DTLB penalties) is in *crafty*, in which random accesses to large (64 KB) constant tables (in `FirstOne()`, `LastOne()`, and `PopCnt()`) are frequently speculated. Assuming 4 KB pages, the 32 entries of the L1DTLB can address only 128 KB of data, so these large, randomly accessed tables pose a risk to virtual memory performance. Control speculation substantially increases the number of loads into these tables, increasing pressure on the already-strained virtual memory system. One would expect the impact for **I-CS**, with its multipath speculation, to be higher than for **S-CS**. This is the case everywhere but in *crafty*.

8.3 Store-to-Load Dependence Elimination

Like many other machines, Itanium 2 suffers a performance penalty when a store operation is followed within a few cycles by a load operation accessing a similar address.[4] Being an in-order machine, Itanium 2 exposes these penalties directly as stalls varying between 1 and 17 cycles, depending on the number of cycles separating the offending store/load pair [51].[5] Further exacerbating this problem is the fact that optimization for instruction-level parallelism tends to generate these conditions quite handily. Simple schedule compression accounts for part of the problem; more instructions in fewer cycles means a greater likelihood of higher penalties. Worse is the fact that load operations tend to move up in schedules (aided by control speculation and memory disambiguation) until they encounter a barrier to their further upward code motion—often a store with respect to which they cannot be disambiguated. This store is sometimes to the location of the load—creating a substantial penalty. Thus, while one might consider the store-load dependence problem to be entirely orthogonal to the core issues of this dissertation, it at the very least poses a vexing problem when one purposes to measure the effect of ILP transformations on real hardware. It is one more good example of a real hardware issue that militates against the ILP production means proposed in the literature.

There are four problematic, meaning liable to cause performance degradation, cases of store-load juxtaposition. The problems all occur when a store and a nearby, subsequent load both access the same 8-byte, aligned region of memory. The four subcases: First, the store may be provably (by the compiler, using local pointer value analysis) accessing the same location as to the load. This is a detectable problem, solvable with a memory copy propagation (local register promotion) technique. Second, the store may be provably (by the compiler, using local pointer value analysis and/or loop dependence information such as omega test) accessing a different location from that accessed by the load, meaning no true dependence exists, but the load and store are determinable to sometimes access the

[4] Due to its often surprising and devastating mode of attack, this event is appropriately dubbed a "whammo" in the EPIC compiler writer's vernacular.

[5] These stalls are accounted both to the L1D micropipeline and to load latency, making their total performance effect nontrivial to derive.

same 8-byte aligned segment of memory. The detection of "nearness" may be through examination of data structures or through induction variable analysis. This is also a detectable problem, but may be more difficult to solve. Third, the store and load are in a known potentially alias relationship, but the compiler cannot determine if the load and store are necessarily (always) to the same (or nearby) locations. Fourth, and finally, the load and store happen to access the same 8-byte aligned segment of memory, but the compiler has no information that this is liable to occur (perhaps the two accesses are to adjacent `char` variables). The first two of these cases were found both to be particularly problematic and to have relatively cost-effective solutions, so these two were dealt with in the work underlying this dissertation. The other two cases appear to occur only very infrequently in the studied suite of benchmarks (after other optimizations have occurred) and thus were not addressed.

The first case, that of a store writing a given location followed by a load from the same location, is the most straightforward. Ordinarily, one of a number of optimizations would eliminate this case by converting the load instruction into a copy (or subword extraction or sign extension, as appropriate) from the value source register of the store instruction. In the IMPACT compiler, this transformation is called *memory copy propagation*. A similar technique was employed in Hewlett-Packard's compiler for the PA-8000 superscalar RISC machine [133]. Traditionally, this transformation was only applied if the store's execution condition subsumed that of the load. Experimentation revealed that with aggressive use of predication, however, predicated stores of stronger or independent execution condition than subsequent aliased loads were frequent enough to cause substantial store-load forwarding penalties. This necessitated the extension of memory copy propagation to include a *partial copy pattern*. In this pattern, the load instruction is moved before the store (avoiding the forwarding penalty) and a predicated move operation is added after the store, to update the loaded value if the accessed location should have been updated by the store instruction. This enhancement was necessary to eliminate some "spuriously negative" results of increasing levels of predication.

8.4 Unroll-Under-Predicate Schema

In software pipelined loops, unrolling is sometimes desirable. It can allow better scheduling of loops with fractional minimum initiation intervals and sometimes admit improved optimization [134]. On Itanium 2, one factor demanded the unrolling of software pipelined loops, that being the problem of spurious store-to-load forwarding penalties. Modulo scheduling of loops that, for example, shift the contents of a character array by a byte, tends to move nearby (but always independent) loads and stores into very close temporal proximity, incurring heavy penalties. If the load and store can be disambiguated successfully from the store and can be control-speculated, loop unrolling can provide an opportunity to separate nearby loads and stores from each other (eight loads can be performed, followed by eight stores). Among SPEC CINT2000 benchmarks, in *bzip2* this achieves a dramatic increase in performance.

The `while` loop modulo scheduling schema employed in IMPACT, however, does not permit scheduling of loop side exit branches in any but the last loop stage.[6] For this reason, a new unrolling schema was introduced to allow the unrolling of loops under predicates (copies of the loop body are subjugated under a loop continuation predicate rather than being guarded by a side-exit branch). In addition, such unrolled loop bodies are converted simultaneously to fully resolved predication (FRP) form to allow all remaining side-exit branches to sink to the bottom of the schedule.[7] Appendix B.10.1 describes the IMPACT compiler's loop unrolling schemata (and alternatives to these approaches) in more detail.

8.5 Related Work and Concluding Remarks

Collard et al. partially addressed similar issues in [135]. Their technique relied on a more sophisticated cache interaction model, but applied only to scheduling and enforced a nonzero latency (spurious) flow dependence between stores and subsequent independent

[6] This simplifies the generation of epilogues for modulo-scheduled code generation.

[7] In FRP form, all instructions subsequent to a side-exit branch are guarded by a predicate that prevents them from executing if they are relocated above the branch. This technique is useful for purposes such as this, but too often incurs too much overhead for general use.

loads adjudged to be to the same cache line, whereas the technique described here allows safe reordering while avoiding expensive juxtapositions of stores and loads. Their technique, nonetheless, achieved a result similar to that presented here for *256.bzip2* (1.25× speedup in their framework). Other CINT2000 components were only barely affected.

The problem of temporal access conflicts is a good example of a secondary effect, generally observed only in real hardware and not in a research simulator, which can have a substantial deleterious effect on performance. Worse, this effect may increase with application of more extensive ILP transformations, complicating experimental evaluation.

9 PERFORMANCE ANALYSIS METHODOLOGY

This dissertation relies on real-machine measurements, performed using real benchmarks in a standard operating system environment. This setting presents unique limitations, challenges, and opportunities. This chapter briefly discusses these observations and describes the tools used to produce the data presented in this work.

9.1 Limitations and Challenges

The primary limitation is that the use of a real machine, while it allows evaluation of a broad range of different compilation strategies, does not admit modifications of the architecture or eliminations of particular microarchitectural effects. These microarchitectural effects often confound straightforward evaluation of transformation techniques as one would hope they might operate in the abstract. There are any number of examples of these limitations, some more obvious than others.

One of the most often problematic components is the data delivery subsystem. This subsystem is fixed by the microarchitecture (preventing experimentation with different cache sizes, etc.) and also very complicated, defying effective compiler modeling. As noted in the detailed experimental results, small changes in schedule often exposed L2 bank and line fill conflicts, causing larger-than-expected and even contrary-to-expected changes in results. Store-to-load conflicts, such as those exposed in *bzip*, proved an even more injurious problem. Instruction delivery, branch prediction, and various other microarchitectural considerations also caused problems in this work.

Such difficulties are not always solely microarchitectural. The interaction of register allocation with CFS transformations is another problem that could have been addressed after most of the more "fundamental" issues could have been worked out. Likewise, the nonorthogonality of predicate definitions had a disruptive effect on generation and optimization of predicate networks; this could easily have been abstracted away in a simulated environment.

In a simulator-based study, the interaction of these events with CFS techniques would not have been studied, or if it had been, it would have been one of the last things to investigate. On real hardware, problems such as this confront the experimenter from the first day of work, and the reality they present is often harsh and noisy.

The compiler is no less a source of perturbation in the measurement of the effects of particular transformations. The interaction of region formation with optimization leaves it very uncertain what exactly particular transformations are accomplishing. The IMPACT compiler was extensively retooled, involving generalization of many optimizations and analyses, to minimize these effects, but this is not in general a soluble problem. The results for *vortex* demonstrate this well. Massive optimization takes place after region formation which may or may not be possible in the baseline version (with improved techniques). Optimization choices then use many registers in a function where predication of a key loop requires a few more, causing spill code to be inserted. This spill code (and similar secondary costs) more than offset the beneficial effects of predication throughout the benchmark, making Hyperblock effectiveness look inferior to Superblock.

Given the complex interaction of effects in a real machine and real compiler, if reported results are to mean anything at all, they must be painstakingly investigated. Just as easily as the microarchitecture can take away from performance of an optimized version of a program, it can steal from the baseline. A final, benchmark-level performance number reporting on the effect of a compiler technique is *honest*, provided the known variables are suitably controlled, but it may not be *meaningful*.

For practical reasons, only a dozen benchmarks were experimented with, consisting of perhaps several hundred performance-relevant code regions. The "strong law of large numbers" does not apply here. A single deviant case in a given benchmark can often

sway results. This magnifies the effects mentioned here, which are more frequent in the real machine environment but are not entirely absent in simulation. This is a daunting consideration for anyone contemplating compiler-based performance work.

Achieving the degree of understanding of the effects of CFS transformation on Itanium 2 described in this dissertation has required a great deal of work to mitigate the effects of both compiler and microarchitectural "performance land mines." This work was much harder and produced much noisier results than simpler, restricted simulation experiments.

9.2 Opportunities

Despite all these challenges, real machine work is worthwhile and important. It is difficult to perform meaningful compiler research on significant benchmarks in a simulation environment.

Microarchitectural modifications can be tested in a simulation environment, sampling pertinent sections of program execution. In that context IPC is a useful measure, and total performance can be extrapolated from reasonable and statistically sound experiments. Not so with compiler work, which can through optimization and speculation change the number of instructions being executed and also have complex and far-reaching effects on the execution of other parts of the program. Whole-program execution is really necessary if results are to be reliable. For benchmarks like SPEC CINT2000, with 12 components each executing for on the order of 10^{11} cycles, only real-machine execution can reasonably provide such measurements.

The opportunity to compare to contemporary commercial compilers on the same platform is an invaluable benefit of this approach. This enables compiler experimentation in a "performance-validated" context. If a new technique produces a result better than the baseline number for a given compiler, but the relation of the net performance to competent compilers is unknown, the benefit might easily be from an unanticipated source (perhaps a commonly known optimization took effect as a side-effect of the technique under test). When, as is frequently the case in these experiments, CFS techniques are

shown to produce higher performance than both the IMPACT baseline and a very competent commercial platform compiler, one can be more confident that something "new and different" is taking place.

Finally, the performance surprises of running on real hardware often lead to interesting new understandings of proposed techniques. The extent and nature of the "wild load" problem were undetermined before this real-machine work provided a real opportunity for its study.

9.3 Performance Monitoring Infrastructure

This work relied exclusively on results from the Itanium 2 performance monitoring infrastructure, consisting of four registers capable of measuring hundreds of microarchitectural events, as well as interrupt hardware capable of reporting near-exact instruction pointer values when particular performance monitoring register counts are reached [51]. While they do not offer the visibility that a simulator would, and while being able to measure only four events at a time is limiting, these resources were sufficient to generate a large amount of useful data.

Performance monitoring hardware means little without an appropriate infrastructure for collecting and aggregating results. The *pfmon* driver and *perfmon* infrastructure developed for Linux-ia64 by Stephane Eranian of Hewlett-Packard Laboratories filled this bill exquisitely [63]. This software provided event counting and sampling capabilities for the full suite of performance monitoring data available on Itanium 2. The sampling facility was extended by this dissertation's author to include a low-overhead aggregation of events into an instruction pointer-based histogram. The sampling-based results presented here (such as in Figure 5.13) reflect the sampling of the instruction pointer every 65 536 execution cycles, with an accuracy of a few issue groups, for the entire SPEC CINT2000 reference input. These results are stable with respect to repetition and change in sampling period. This new feature enabled the low-overhead production of function-level results, as well as aiding in localization of particular performance problems. A similar feature has been incorporated into newer versions of the *pfmon* software, so it is unnecessary to detail it here.

10 RELATED WORK

The roots of EPIC research reach back to at least 20 years ago, with work on the machine that came to be called Itanium itself spanning at least a decade. There is in this period an immense amount of material in the literature on EPIC and EPIC-related topics, much of which has been introduced at points of salience in the other chapters of this dissertation, particularly in Chapter 1. This chapter provides a brief, unified survey of the previous related work as a resource for broad exploration.

10.1 General VLIW and EPIC Research

Schlansker et al. [4] of Hewlett-Packard Laboratories (HPL) surveyed previous work on EPIC features and techniques, at the time referred to collectively as HPL PlayDoh (now called the HPL-PD architecture), in a 1996 technical report. This report serves, together with with the work of August et al. [38], as an example of the culmination of EPIC research prior to the availability of real hardware implementations in the Itanium Processor Family in 2001. The latter paper provides experimental data, to which we may compare the results of this dissertation's experiments.

When we examine previous, simulation-derived EPIC results, such as the performance of the nine SPEC CINT92 and SPEC CINT95 benchmarks in [38], we find a speedup of 1.17 for predication and 1.68 for predication and speculation combined on the IMPACT EPIC simulator. The fact that this far exceeds our results on Itanium 2 is explained in three ways. First, past "clean" simulations did not model data and instruction cache stall

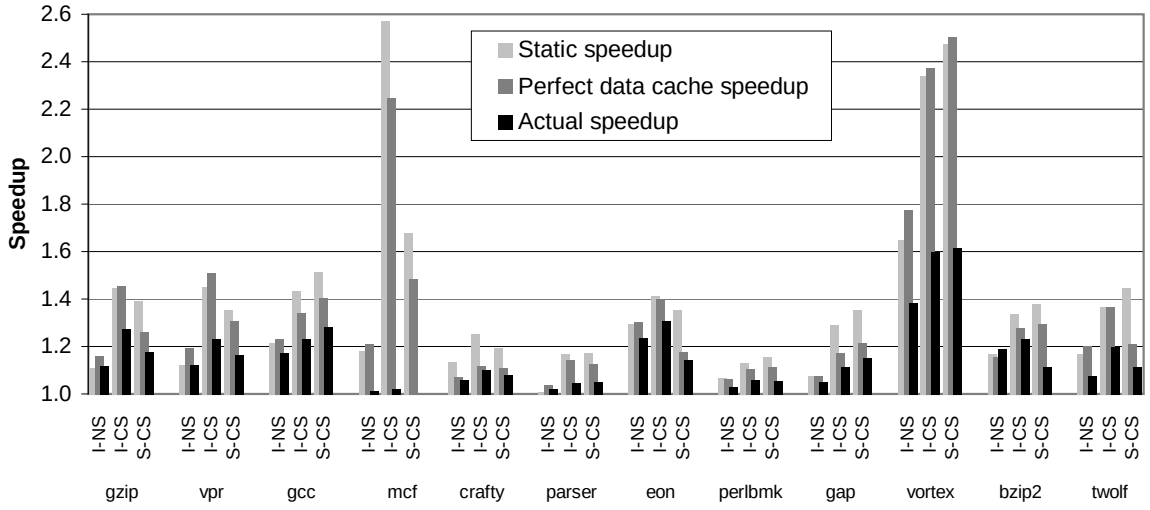


Figure 10.1 Actual and two idealized speedup measurements.

cycles or other dynamic events. One issue stands out far ahead of all others, and this is the often dominant role of data cache miss stall time in performance outcomes. When we correct for this using measurements of these effects from performance monitoring counters, we find much better agreement with these past results. Figure 10.1 shows the relation of the ILP-configured compilations to the IMPACT baseline (**O-NS**). The "exploited" speedup reflects the change in total execution cycles, as was indicated in Table 2.2. The "planned" speedup, on the other hand, measures the change in the number of execution cycles statically anticipable by the compiler.[1] Considering only these cycles, as in past simulations, IMPACT achieves an average speedup of 1.47, closer to the 1.68 speedup achieved in past work. (To emphasize the importance of data cache stall as a contributor, excluding only this runtime effect category, IMPACT achieves a speedup of 1.41. Clearly, this is an aspect that needs to be addressed to strengthen the compiler's ability to plan for high EPIC performance.) Second, the SPEC CINT2000 benchmarks are substantially harder to parallelize than the older benchmarks; as they have more frequent and more irregular control flow, they require more code-expanding transformations to expose the same degree of ILP, but their larger code size can cause such expansion to have undesired effects on instruction cache performance. Finally, the

[1]This measure includes the **unstalled** and the three **scoreboard** components of Figure 2.6; it subtracts out all "dynamic effects." The "planned" execution time assumes, for example, that all branches are predicted correctly and all loads complete with minimum latency.

benchmarks of [38] benefited from data speculation, which IMPACT does not currently exploit on IA-64.

This work, then, largely confirms the work of August et al. [38], given the differences between their hypothetical EPIC system and the Itanium 2 microarchitecture. The earlier work expounded on the constructive collaboration between if-conversion and speculation techniques. This dissertation has shown this same synergy in real hardware experimentation and adds an understanding of how instruction cache, data cache, branch prediction, exception handling, and operating system model color a modern realization of the technology outlined by the earlier paper.

A long heritage of other VLIW and superscalar compiler work contributed features that are now part of IA-64. This work included complete, hypothetical machines that supported extensive research projects [124]. As real EPIC hardware only recently became available, and as research compilation environments are still adapting to it, the validation and recalibration of these results to the modern situation is work just begun. Triantafyllis et al. [40] demonstrated using the Intel production compiler that controlling optimization for EPIC systems is a difficult problem, as large fluctuations in performance can be observed with changes only in how, when, and to what degree existing optimizations are applied. Their work pointed out interactions similar to those indicated here, within a very different compiler infrastructure. Choi et al. [39] performed a focused evaluation of if-conversion on the Itanium processor, in the context of a more conservative production compiler. The relation of this dissertation to their work was discussed in Chapter 5.

There has been extensive work in ILP compilation for VLIW and EPIC systems, as well as in a number of peripherally related topics. This section surveys this background, providing an understanding of the wide variety of individually promising techniques at hand and the difficulty of integrating selected ideas into a stable, productive framework.

10.2 Historical Development of CFS and its Variants

History provides a unique perspective on the control flow structural approach to instruction-level parallelism. The literature shows a complex and multifaceted pattern of development, starting with complicated and full-featured schemes. Simplified (and yet

highly profitable) models were then introduced, which encouraged significant development of ancillary techniques. These models relied to some degree on the characteristics of machines and benchmarks at the time for their success. Today, the simplified models are under substantial stress. The reasons for this include the accumulated complexity of the ancillary techniques and changes in both machine technology and software structure. Some of the more complicated models of the past are beginning to re-emerge, and the interaction of these models with the important ancillary techniques (which were developed in the context of simplified models such as the intraprocedural Superblock model) has placed compiler developers in a real quandary. An understanding of the historical contributors to the models will prove an indispensable aid in unraveling this situation, and is thus included here.

The ILP cultivation methodology described here began as primarily an instruction scheduling problem in the broad class of VLIW-styled machines. In the earliest work, instruction scheduling was known as "microcode compaction" (finding operations that were compatible to execute in the same machine cycle and compacting them into executable bundles). The earliest of this work focused on compacting single control blocks (i.e., control equivalent regions, including no branches) of microcode using classical data dependence tests and basic machine description technology [136].

Gradually compaction was extended to include "microtemplate transfer," the movement of instructions[2] across control block boundaries. This transformation involved notions of data flow analysis (liveness) and control dependence / equivalence. Tokoro's work showed examples of this type of motion (this type of work would later be referred to as "menu compaction" because of its selection of motion alternatives) and laid out the basic rules permitting it, but stopped short of presenting a unifying framework to control the most beneficial exchange of micro-operations[3] [138].

[2]The author has modernized the terminology used in these early papers, which had various names such as "operation," "micro-operation" and "microtemplate" for what are today called instructions.

[3]One interesting facet of Tokoro's work is that it considered control dependence and data flow interprocedurally, a feature that has only occasionally re-emerged in later systems [128, 137].

10.2.1 Trace scheduling

Compilation (and limited, continuing hand-coding) of more control-intensive codes led to the development of global instruction scheduling techniques. The work of Fisher extended compaction to traces [44], applying dependence-height prioritized, list-based instruction scheduling across splits and joins in the control flow graph. This general work forms the basis for all subsequent approaches to global (i.e., inter-basic-block) scheduling techniques. The key innovation relative to the menu compaction method, which compacted individual basic blocks and then sought specific opportunities for interblock instruction transfer, is the fact that an entire trace (a contiguous, straight-line sequence of blocks tending to be traversed for a given input) is scheduled at once. This allows freer scheduling and achieves better schedules, being less prone to the "local minima" on which the menu-based approach could founder.

Trace scheduling operates by greedily selecting traces from the control flow graph (in decreasing order of their frequency, as indicated by the control flow profile) and scheduling them, until no instructions are left unscheduled. In the course of scheduling, when an important trace is compacted, compensation code may be pushed out into off-trace blocks to achieve a better schedule within the trace. In practice, in the Multiflow compiler, only stores were permitted to be extruded in this manner, limiting code expansion [25].[4] Both program structure (disruptive presence of loops and joins) and profile variation (difference between the inputs of which the compiler has knowledge and those used in actual practice) impact the effectiveness of trace scheduling. Performance of program paths that are not maximally covered by traces may be degraded, and a nonnegligible code replication cost is incurred.

With respect to the handling of complex control flow, Fisher's trace scheduler had a notion of control equivalent motion, even across loops, based on a hierarchical decomposition of structured control flow graphs [44]. A second-generation trace scheduler addressed the problem that trace scheduling, since it worked with a single trace at a time, often missed potentially profitable motions of instructions between different traces

[4]Many details of the Multiflow compiler and Bulldog, the research compiler on which is was based, are available in [25] and [139], respectively

(what was called "nonlinear compaction"). This acknowledges that not all branches are highly biased, and it is necessary to exploit ILP across branches that have no statically determinable, highly-biased direction. This new scheduler operated on maximal acyclic control flow subgraphs, or *clusters*, rather than single traces, using a *speculative yield* function that estimated, based on profile information and a number of other factors, the likely benefit of speculating an operation upward within the cluster [140].

The ability to schedule potentially-excepting operations upward, above conditional branches, granted significant freedom to the trace scheduling compiler. The Multiflow hardware, on which the trace scheduling approach was applied, provided support for suppression of exceptions on speculated, potentially-excepting operations [25, 56]. This "silent" or "general speculation" model formed the basis for early experiments with EPIC machines, and is still applied in the IMPACT compiler today (see Section 6).

Another respect in which the trace scheduling work is fundamental to later VLIW/EPIC techniques is that of profile dependence. Much of the work suggesting that control flow profiling was a usable means of guiding compiler decision making was performed in this context [32, 140]. Given the radical transformations that are now necessary to render instruction-level parallel execution of irregular programs, new approaches, or at least new studies, may be called for (as suggested by the difficulties encountered in maintaining useful profiles across procedure inlining and tail duplication, as described in Chapter 7).

The trace scheduling work, while fundamental to all subsequent approaches to enhancement of ILP through instruction scheduling (in the presence of control flow) lacked some features that would allow the technology to advance dramatically in the handling of complex and irregular codes. Most notably, it did not support wholesale code replication (e.g., tail duplication) for the purpose of specialization. This idea was the primary contribution of a simplified technique yet to come, that being Superblock formation. A trace also could not include the back-edge of a loop, prohibiting software pipelining.[5]

[5]The Multiflow architects considered software pipelining to be of limited value because they favored heavy unrolling (up to the 96th degree) in the compilation of vector loops [25].

10.2.2 Superblock-based approaches

The trace scheduling model allowed for the biased motion of operations along common paths, to compress the schedule of frequent cases. It did not, however, substantially simplify the control flow graph. Traces could potentially contain side-entrances, which posed complications for instruction scheduling (requiring complex bookkeeping) and often prevented simplifying code optimizations. Researchers saw an opportunity for simplified scheduling and enhanced optimization within regions that were single-entry rather than potentially multiple-entry like the traces of earlier work. Thus, the Superblock, the next step in the development of ILP techniques, was a simplification of the earlier model [22]. The Superblock is a *single-entry*, potentially multiple-exit trace.[6] A Superblock is formed by selecting an important trace, which may contain side-entrances, and then by performing tail duplication to render the region single-entry. The cost of this simplified model is substantial, up-front code expansion. This creates the fundamental tension of the Superblock model between code expansion and optimization and scheduling scope. The desire to perform additional specialization-based optimizations on Superblocks began to muddy the waters of ILP transformation. No longer was it just about scheduling.

10.2.3 Hyperblock-based approaches

The Superblock retains one crucial shortcoming of the earlier trace framework—it optimizes single traces in isolation. Gradually, program complexity, the cost of branch misprediction, and the need for higher levels of parallelism became more prominent. Predication, the ability to turn individual instructions on and off with a Boolean predicate, became a means of incorporating more than a single control flow path into an optimization and scheduling region [36, 76, 83]. Where multiple Superblock candidate traces can be combined by predicating individual instructions with Boolean predicates, a Hyperblock [36] results. This presents new opportunities, as well as new difficulties. Hyperblock formation potentially involves less code expansion than Superblock, since each block incorporates multiple paths, but complicates both the region selection process

[6] The Superblock is structurally identical to the extended basic block of [49, p. 714]; the distinct name reflects the manner of construction, not a different structure.

and subsequent optimization, as was described in Chapter 5. The complexity and risk of the Hyperblock formation process led to work in partial reverse if-conversion [48], and the predicate definition networks of aggressively formed Hyperblocks were shown to be susceptible to Boolean network optimization [84].

Until this dissertation work, however, research-grade Hyperblock formation techniques had not been demonstrated in a real-machine context. These experiments indicated the cost of very aggressive speculation usually assumed to be performed in predicated regions. The necessary means undertaken to correct these problems may reduce the applicability of very aggressive formation and optimization techniques.

10.2.4 Optimization of Hyperblocks

Various mechanisms have been proposed for optimization of already-formed Hyperblock regions, where optimization means a reduction in dependence height. There are important contributions here, but this dissertation argues that these techniques do not of themselves provide primary means of making ILP compilation outcomes more predictable or more generalizable. Simply adding them to the wobbly tower of ILP transformations is not likely to improve performance dramatically for a broad spectrum of nonnumeric applications like SPEC CINT2000. Many hold a very kernel-focused view of ILP (important, to be sure, but not effective for every program), possibly contributing to the expansion of lukewarm code for little tangible gain. Some also need to be applied in the context of more strategic optimization. The work in Boolean optimization of predicate networks, for example, met with reasonable success in its goal of reducing predicate computation height, but often failed to achieve the best possible results because it did not know how to use instruction ordering, register naming, promotion, and knowledge of machine resources together to best advantage. Finally, these techniques are subject to the typical compiler phase-ordering constraints. Those described here, for example, assume the formation of Super- or Hyperblock regions before they begin their work; their heroic efforts may not be able to make up for bad formation decisions.

August demonstrated the inefficiency of Hyperblocks formed too aggressively, but suggested that overaggressive formation was unavoidable because such "overreaching" often

enabled subsequent, profitable transformations. Subsequent research has underscored this observation. Partial reverse if-conversion was introduced to reduce the overhead of decisions that, in the final analysis, resulted in a possible degradation [48, 92]. This work also introduced the idea of optimization of the predicate defining instructions as a Boolean network [84].

Schlansker et al. laid out a number of strategies for height reduction in Superblocks. They describe techniques by which control recurrences in important loops having data-dependent side exits can be compressed [141] and later introduce the idea of "blocked back-substitution," a means for controlling the expansion of terms in height-reduced computation trees [142]. They later describe a technique whereby infrequently executed side exits can be combined and moved off-trace using fully resolved predication [68]. This technique, dubbed Control CPR (critical path reduction), can also be applied profitably to Hyperblock regions. All these techniques have the potential to change dramatically the characteristics of control and data dependences within a Hyperblock region, but all are applied after the regions have been formed.

Carter et al. proposed Predicated Static Single Assignment as a means of removing false dependences in optimization and scheduling of Hyperblocks [85]. This work used a new scheme, block predication, for predicate computation, which was more amenable to efficient logical operations on existing predicates (as was done in [84] to a much less radical degree). Within this framework, operations were promoted (released from guard predicates, reducing predicate computation dependences) and split as necessary at data flow merges to relieve constraints. Results showed substantial improvements in instruction-level parallelism for wide (16-issue) and infinite-issue Trimaran machines. Interesting is the dynamic code expansion (total number of operations executed) required to deliver between 1.4× and 2.3× speedups on a 16-issue simulator: between 1.3× (for the best result) and 3.3× (one of the lesser results). The technique seems to be highly variable in the amount of dynamic code expansion it causes (also in the number of register live ranges overlapped), without an obvious way of maximizing benefit while minimizing this expansion—making this a technique in need of stabilization.

Eichenberger et al. [86] developed a similar scheme for control and data height reduction in Hyperblocks. Their scheme selectively replicates operations (splits nodes implementing a data flow merge) within a Hyperblock as needed to enable path-specific optimizations. The node splitting enabled the optimization of some additional predicate expressions after the manner of [84]. This work was demonstrated experimentally within the Trimaran framework for small benchmarks (*wc*, *129.compress*, and *130.li*) on very wide machines (issue width 16, 32, and 64). As we found in experimentation for [84], these types of techniques could well be expected to be unstable in resource-, code-size-, and register-constrained environments (as would be the case with more complex benchmarks and more conservatively sized machines). Expanding code arbitrarily, even if infinite-issue dependence height is reduced, can often degrade performance unexpectedly.

In more peripherally related work, Ebcioğlu et al. [143] described a number of techniques similar in character to some of the proposed Hyperblock optimizations in their attempts to increase oracle parallelism in a dynamic compilation environment. In particular, to reduce the exponential growth of instruction trees (an alternative to predication preferred in IBM's VLIW research), they employed a unification technique that found and combined common instructions on distinct paths.

10.2.5 Ancillary transformations

A perusal of even early VLIW compilation work reveals a variety of ancillary transformations that existed to assist the trace- or region-based optimizer in exposing instruction-level parallelism. In the Multiflow culmination of the trace-based work, which performed little bulk code specialization, although it did schedule traces according to execution profile bias, loop unrolling was employed to expose more parallelism within loops [25].

Early Superblock work identified branch target expansion, loop peeling, and loop unrolling as useful region-extending transformations; register renaming, induction variable expansion, and accumulator expansion as means of breaking loop anti-, output-, and

flow-dependences; and sinking-based partial dead code removal[7] as a path compression technique [22].

10.2.6 Trace scheduling revisited

Since the introduction of the Itanium architecture, trace scheduling has been revisited in a new, multipath form. Wavefront scheduling [34, 50] performs global scheduling on acyclic control flow graphs using a path-based representation for data dependence. This approach is used in the Intel Electron Code Generator (part of the `icc` compiler). Wavefront scheduling performs many motions in a fashion similar to the trace scheduling techniques (including, for example, the movement of code across loops via hierarchical control flow reductions). While it works on maximal acyclic regions,[8] as does Fisher's latter scheduler [140], it adds the notion of a *wavefront* that represents the scheduling frontier that moves through the region. This concept enables natural schemes for the calculation of speculative yield and the placement of compensation code. Finally, Wavefront scheduling makes use of both control speculation (using the recovery code Sentinel model) and predication in moving instructions across conditional branches. Where a branch predicate (and/or its opposite) is available before the branch is to be executed, instructions from one subsequent path or the other may be moved above the branch without speculation, if guarded by the appropriate predicate. This reduces the overhead of "speculation." Likewise, operations may be moved down across control flow merges without speculation by guarding them under a predicate.

Moon and Ebcioğlu proposed the *selective scheduling* framework to maximize instruction level parallelism within a global scheduling framework. This is similar in spirit to the Wavefront scheduling scheme. Both these frameworks, like the much earlier and less general compensation-based trace scheduling approaches [44], attempt to accommodate some limited degree of instruction replication, speculation, and renaming into the

[7] VLIW work generally referred to this transformation as "operation migration." See Appendix B for a description of these techniques.

[8] For these purposes, an acyclic region may contain loop(s) that have been reduced to single node(s).

scheduling process to improve parallel issue. These frameworks succeed in packing instructions effectively into existing control flow structures, sometimes opportunistically forming Superblock or Hyperblock regions, without the explicit cost of tail duplication. They do not, however, make dramatic modifications to the control flow graph, like peeling, that tend to expose more parallelism in more control-intensive programs. The work of Moon and Ebcioğlu contains some interesting comments on the scalability of their technique, stated in terms of the ratio of usefully executed instructions to total executed instructions (including all speculation). They note that, for some benchmarks, increased aggressiveness only serves to decimate this ratio, suggesting more powerful techniques or a different way of thinking about ILP are in order.

While it is not within the scope of this dissertation to compare the control flow structural approach to a Wavefront-based approach, it is important to note that Wavefront has the potential to deliver quite respectable performance on a broad range of programs. Wavefront scheduling was reported to deliver a 30% performance improvement relative to basic block scheduling on Itanium [50]. It is the opinion of the author, however, that the control flow structural approach, by virtue of its code specialization, offers the potential for greater gains (at the cost of potentially more extreme profile dependence). That this is likely the case is borne out in the experimental results. It is important to note that Wavefront and the approach described in this work are not entirely exclusive of each other in implementation—selective Wavefront motions may have a great deal to offer a CFS compiler, and vice-versa.

10.2.7 Bringing CFS transformation into the present

Recent challenges that would have been peripheral or even foreign to the techniques of the past include profile variation, the very frequent control flow characteristic of highly structured or object-oriented programs, the relative constrictiveness of modern EPIC machines, and the highly variable latency of memory operations. Various chapters of the dissertation have substantiated these issues, and the reader is well advised when considering these past techniques to consider the implications of these problems. Many of the promising ideas of previous work have been rendered much more difficult to achieve

in a real hardware context and in benchmarks with less dependable and less regular patterns of control flow.

10.3 Inlining and Code-Expanding Transformations

Procedure inlining and path specialization through code-expanding transformations (really different cases of the same basic transformation) are frequently employed as the basis for ILP transformation but are rarely examined critically. Much of this technology was developed when applications were much smaller and there was less pressure on the instruction cache (cf. [22], which made only passing notice of 1.5× to 4.5× code expansion factors in Superblock formation—levels clearly harmful in many contexts today).

Hyperblock-based work [36, 48], likewise, rarely concerned itself with the code expansion performed in enabling formation of large regions in the presence of long, cold paths. Today, since more aggressive tail duplication and peeling must be performed to enable effective region formation and since the instruction cache is placed under much more stress by larger application footprints, these are now primary concerns.

In a work not primarily concerned with EPIC machines, McFarling proposed a technique for deciding when to perform procedure inlining that took into account instruction cache limitations and program structure [127]. This work reflects some of the concerns expressed here, but does not take into account the unique features of the problem in EPIC systems, namely the definite need to produce ILP, the degree of specialization of the inlinee possible in a powerful transforming compiler, and the possibility of transforming the program prior to inlining to consolidate multiple calls to the same or similar procedures. There is also, of course, an extensive body of work in optimizing procedure layout for cache performance [144, 145]. This work would ease but not solve the cache capacity vs. code customization conflict.

11 CONCLUSION

This dissertation has demonstrated how IMPACT's "control flow structural (CFS)" approach to compilation for Itanium 2 provides a 1.20 average speedup (including single-benchmark speedups of up to 1.59) relative to traditional optimization of code at the same inlining and interprocedural analysis levels, and a 1.70 average speedup relative to GCC, in the SPEC CINT2000 suite of nonnumeric applications [5]. These benefits come by specialization of single-and multipath regions, selected according to execution bias, for more instruction-parallel execution. Region specialization prepares common paths for specialization and, with the aid of predication, removes branches, reducing the penalties of branch misprediction. Instruction scheduling, empowered by region specialization and control speculation mechanisms, compacts these regions for efficient execution. CFS transformation is effective in the real-world context, and generally compares favorably with commercial compilers taking more conservative approaches.

CFS transformation has always been assumed to have two primary, negative side-effects. The first is that, since these techniques rely on such extensive code replication, instruction cache performance losses are bound to rapidly outstrip the benefits of transformation. The second is that, since speculation causes more memory accesses in a machine already often stymied by memory latency, the gain of speculation would also be spent rapidly. Experimental results presented here show neither of these effects to have substantially dampened the gains of the control flow structural (CFS) approach. In fact, significant overall gains and, most curiously, some improvements in instruction

fetch efficiency have been observed as the result of specializing transformations, suggesting that simply curtailing static code size is not the best means of improving Itanium 2's front-end performance.

These positive results are, however, colored by a new quantitative understanding of potential risks: in a few cases, potentially excessive replication led to small deleterious instruction cache effects; elsewhere, control speculation of "wild loads" exacted a heavy toll. The latter problem was resolved satisfactorily with new approaches to control speculation in the compiler, and new heuristic analysis mechanisms designed to prevent the control speculation of likely-to-fault loads. These techniques made the general speculation model a practical and effective approach to control speculation across SPEC CINT2000. Less could be done about the former problem within the practical constraints of the IMPACT framework and the limited observation points provided by SPEC CINT2000.

This effort thus shows the potential for CFS-style transformation to provide higher performance than now-accepted EPIC compilation approaches, while pointing toward a need for a new style of interprocedural optimization framework for more controllable CFS optimization. This work provides an important milestone in the development of EPIC, showing that further gains will rely on more sophisticated, as well as more aggressive, compiler techniques, capable both of exposing more ILP and of managing traditionally secondary elements such as instruction cache and the performance-stability of speculation. Finally, the increasing significance of run-time effects, such as data cache and DTLB stalls, in determining end application performance has been demonstrated. This calls for better microarchitectural management of these events or new research that puts them within the compiler's understanding.

This dissertation aimed to increase the aggressiveness, and hence the potential for final code performance, of CFS techniques in an EPIC compiler, while at the same time seeking to improve the performance stability of these techniques. While at a practical level this goal was achieved within the experimental benchmark set, much more systematic work could be done now that the central problems have been exposed and characterized. It has become apparent that, for programs structured in such a way as not to admit extensive procedure inlining as an appropriate means of CFS enablement (for example,

crafty and *parser*) alternative means must be employed. The classical structure of the ILP compiler as an early, interprocedural phase performing procedure inlining, followed by intraprocedural processing, is no longer tenable if the instruction cache is to be managed effectively. This change would require extensive infrastructure modification and the author was, regrettably, thus not able to perform experiments with such a framework in this dissertation.

Having sounded at best an uncertain clarion call for the success of EPIC techniques in producing acceptable levels of performance on the Itanium 2 platform, the author would like to conclude briefly with his opinions on the future directions of ILP research. One of the most pronounced differences between the projections of the previous generation of research and this dissertation is the anticipated future scaling of the EPIC architectural approach. Only seven years ago, the author was among a group of researchers who predicted "by the year 2000, hardware technology will be capable of producing microprocessors that execute up to sixteen instructions per clock cycle [38]" (presumably under the "simplifying" EPIC paradigm). This has not come to pass, and no one is making the same predictions today, even by the year 2005, for two reasons. First, a more complete understanding of the complexity of pulling ILP out of modern, nonnumeric applications renders the building of such a processor an almost laughable idea. Second, several years of meteoric technology scaling have made the building of such a processor, with its intensely complicated centralized design, highly impractical. Architecture has turned in more decentralized directions.

As Rau and Schlansker pointed out very insightfully, early in EPIC's development, though, this shift does not mean the end of EPIC's usefulness [27]. This dissertation has demonstrated substantial simplification of instruction flow and improved instruction fetch efficiency that could make threads cooperatively executing a program more compatible with each other than otherwise possible.

One thing is clear, though, and that is that compiler changes alone will not deliver dramatically higher performance on today's platforms. At the same time, a hardware solution cannot afford to sacrifice the basic simplicity of the EPIC microarchitectural model. The author sees two directions of development that are likely to be profitable:

the first attempts to ameliorate the negative effects of memory latency with a limited amount of dynamic execution; the second adds the ability to exploit fine-grained threads while maintaining the features of EPIC within individual instruction streams.

Although experiments generally showed that CFS transformations, including control speculation, did not materially increase the incidence of performance-degrading memory latency events, these events did affect outcomes. The impact of ILP-enhancing techniques employed in this dissertation was substantially diluted by the dominant effect of memory latency stalls. It appears that, at least for these nonnumeric benchmarks, this problem must be addressed microarchitecturally. Some means of doing this have been explored, including full out-of-order implementations of the Itanium architecture [29, 146] and limited out-of-order adaptations more in keeping with the EPIC theme of hardware simplicity [28, 147–149]. While the evaluation of these approaches is beyond the scope of this work, these appear to be promising possibilities. With the adoption of one of these models, much more of the potential gain from ILP optimization, as was shown in Figure 10.1, could be exposed.

EPIC techniques can dramatically simplify instruction streams [101], potentially making them more compatible components of a multithreaded execution environment. Several examples were encountered in this work which support the idea of generating microthreads to execute parts of nonnumeric programs. These are especially appropriate where the typical, statically-scheduled ILP expression of parallel constructs would be prohibitively expensive or where runtime delays are likely to occur in one thread or the other.

In conclusion, this dissertation found that the historically proposed control flow structural techniques, when appropriately controlled, can deliver competitive to exceptional levels of performance on the Itanium 2 microarchitecture, a modern instantiation of the core concepts of EPIC design. It also identified the significant limiting factors in further improvement of this model, including profile stability, management of instruction cache and registers in an interprocedural framework, diminishing returns of path expansion, execution time becoming decreasingly anticipable at compile-time, and suggested potential future directions for EPIC research. This timely, real-world evaluation of a generation

of experimental techniques will serve as a practical foundation for another generation of research into the effective placement and management of the software-hardware interface.

APPENDIX A. THE INTEL ITANIUM 2 MICROPROCESSOR

The Itanium 2 microprocessor provides a concrete reference implementation of a contemporary, general-purpose EPIC system. This appendix provides additional details about important processor subsystems, including performance measurement apparatus, that was not included in the main presentation of the dissertation. For further information, the interested reader is referred to the detailed description of the microarchitecture presented in [51].

The microarchitecture of the Itanium 2 processor confesses the faith of its designers in the compiler's ability to deliver instruction-level parallelism using its many registers, wide issue, control and data speculation, and predication. Figure A.1 shows an overview of the Itanium 2 pipeline, which is, briefly, an 8-stage, nominally 6-issue, in-order design. The execution units are presented with up to six compiler-selected, independent operations in each cycle; the processor attempts to execute these as given, without reordering or recombination. Instructions are marked for parallel issue by grouping them into specific bundle templates (which, together with resource limitations, specify the types of instructions that may issue together) and demarcating issue group boundaries with explicit stop bits (indicated as "`;;`" in assembly code).

The pipeline view of the architecture consists of two "rigid" pipe sections separated by a small instruction buffer. This allows the front end to fetch instructions at a rate greater than the execution rate of the back end, and to continue to fetch instructions during back-end stalls. The entire core pipeline is only eight stages in length, in comparison to

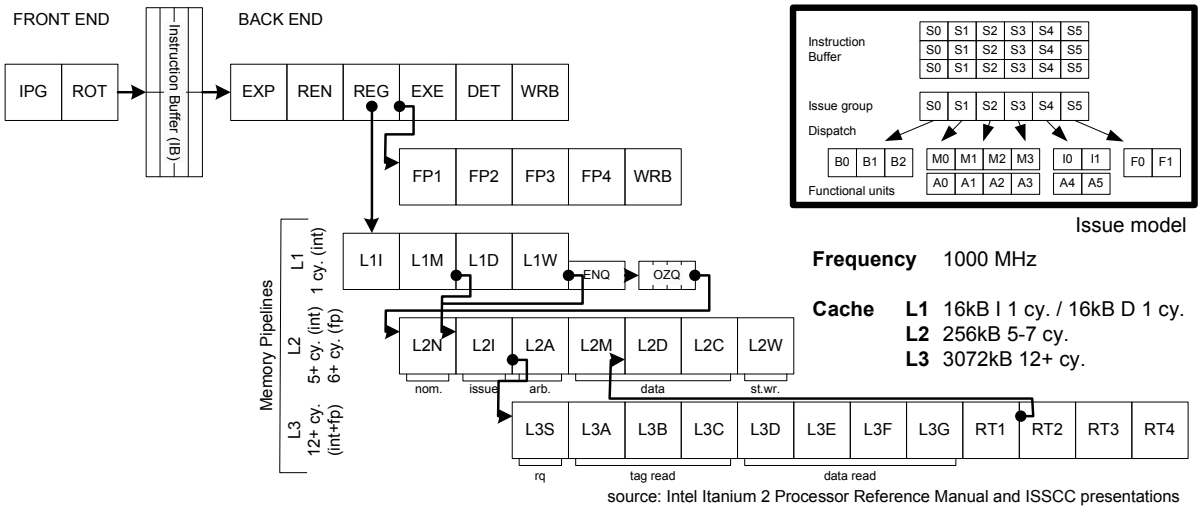


Figure A.1 Intel Itanium 2 pipeline.

more than 30 stages for other contemporary designs. This implies a relatively low clock rate, less than half the clock rate of the superpipelined approach [14], and thus requires the compiler to identify substantial instruction-level parallelism to achieve competitive performance.

A.1 Instruction and Data Delivery

A sub-1-cycle L1D and L1I (both only 16 KB) allow 1-cycle L1D access and a 0-bubble taken branch. A branch direction misprediction results in a 6-cycle penalty. The design was calibrated specifically to provide a 0-bubble taken branch (supported by a 3/4 cycle L1I) and a 1-cycle data cache load time (3/4 cycle L1D) [41]. This decision can be interpreted as a hedge against the odds of the compiler being unable to transform the program to remove taken branches and to provide enough ILP to hide multicycle L1 accesses. Allowing longer latencies could have allowed increased capacity in these important caches, but this decision would only have been wise if the compiler could be expected to have consistent effectiveness in transforming away dynamic branches and in optimizing code for ILP.

The Itanium 2 has a 16 kilobyte first-level instruction cache, a 16 kilobyte first-level data cache, a 256 kilobyte shared second-level cache (at approximately a 5-cycle latency),

and a 3 megabyte tertiary cache,[1] also shared (with at least a 12-cycle latency). Floating-point accesses bypass the first-level data cache, and the latencies of accesses to lower levels of cache vary due to queuing behavior and conflicts in the memory subsystem.

The effectiveness of address translation caching is sometimes an issue for data accesses (less so for instruction accesses in these experiments). Itanium 2 has a 32-entry, fully associative L1 DTLB, whose entries are for *fixed-size* 4 kilobyte pages. When an entry is expelled, all L1D lines corresponding to the expelled page are invalidated, increasing the costliness of thrashing in the L1 DTLB. An integer load that misses in the L1DTLB but hits in the L2DTLB (a 128-entry, full associative table that, unlike the L1DTLB, is capable of representing larger pages in single entries) is penalized 4 cycles (in addition to the L2 data cache access latency). The hardware page walk that occurs if the address does not hit in the L2 takes at least 25 cycles. The effects of this problem on results were discussed in Chapter 8.

Increasing the system page size will move events from the "VHPT" column to the L2TLB column, but will not decrease the total number of misses in the L1TLB, which has fixed-size 4 KB pages. The handling of a L1TLB miss in L2TLB is at least twice as fast as handling it in a walk of the VHPT, so inceased page sizes could substantially benefit *mcf* and, to some extent, *vpr* and *vortex*. Increased page sizes, because they cannot increase the L1TLB hit rate, cannot mitigate the effect of general control speculation where it increases virtual memory penalties.

For further details on the complex and performance-critical data and instruction delivery mechanisms of the Itanium 2, see [51].

A.2 Branch Prediction

The Itanium 2 processor implements a two-level Yeh-Patt branch predictor [97] in a manner tightly integrated into the first-level instruction cache. A secondary history table stores prediction data for lines evicted from the first-level cache, extending the accuracy of the predictor across large spans of code. This prediction scheme achieves a 95% rate

[1] Newer models of the Itanium 2 have larger third-level caches.

of correct branch prediction across the SPEC CINT2000 suite in these experiments. The IMPACT implementation uses static branch prediction hints for those branches control flow profiling shows to be highly taken ($\geq$ 97% taken) or highly-fall-through ($\leq$ 3% taken), except in the context of `bbb` bundles, in which the use of static prediction hints often has unanticipated performance consequences. This in theory reduces the pressure on prediction hardware, but in practice seems to have little effect on net performance, at least within SPEC CINT2000.

A.3 Control Speculation

Control speculation was the most uniformly productive, EPIC-specific architectural feature studied here. Since the architectural mechanisms supporting it were discussed already in Chapter 6, it will not be belabored further here.

A.4 Predication

The predication mechanisms offered in the Itanium 2 were generally in keeping with those described in the earlier HPL-PD architecture [124], and so support the common methods for if-conversion and other uses of predication. One feature, though, bears mentioning: The set of predicate defines provided in the Itanium architecture are not fully orthogonal; that is, for some of the predicate definition types, not all the comparison types are available. Generally speaking, only equality, inequality, and comparison relative to zero are available in predicate defining (`cmp`) instructions with parallel (`or`- or `and`-type) predicate destinations. Where unsupported comparison/destination combinations are needed, extra instructions (with additional dependence height) must be inserted. This complicates optimization of the predicate network [84] and poses a difficulty for general use of these types, which are important to height-reducing techniques such as branch combining [79] and critical path reduction [68]. This was a key factor in the fact that branch combining was often detrimental to performance in the described experiments with IMPACT, and so was disabled.

A.5 Register Resources

The Itanium architecture provides extensive register resources to sustain high rates of instruction issue in the presence of many live value ranges. Most of these registers are organized into the *register stack*, a windowing structure composed of 96 hardware registers. At a function entry point, a compiler-specified number of these register are allocated as function temporary registers. These registers are automatically allocated, saved, restored, and deallocated by the Register Stack Engine as required by the pushing and popping of invocation records on the execution stack. Some of the register can be specified to rotate in the context of modulo scheduled loops, allowing kernel-only modulo scheduling schema to be applied.

IMPACT successfully uses these registers for promotion of memory-bound variables across large regions of program execution (but only intraprocedurally). Additionally, the cultivation of ILP tends to increase the number of live ranges overlapped, as one might expect. Section 7.4.1 dealt with the implications of inlining and other code replication and optimization on register stack activity, and Section 5.7.5 reflected on the impact of predication on this resource.

The register stack engine is noticeably active (contributing more than 1% to execution time) in *gcc*, *crafty*, *parser*, *eon*, and *vortex*. It should be noted that the Itanium 2 implementation implements *lazy spilling*, the spilling and filling of the register stack only when allocations or deallocations require it. The architecture supports the idea of asynchronous maintenance of the register stack, which would allow the machine to preemptively spill and fill backup frames of the register stack during cycles when the cache interface is idle, in an attempt to hide the latency of these adjustments [57]. This might effectively hide most of the register stack engine activity encountered in these experiments.

Table A.1 Cycle accounting categories in terms of hardware counters

Category	**Event(s)**
unstalled execution	CPU_CYCLES(**user**) − BACK_END_BUBBLE_ALL(**user**) User cycles in which at least one instruction retired.
floating-point scoreboard	BE_EXE_BUBBLE_FRFR(**user**) Cycles stalled on pending floating-point operations.
integer load bubble	BE_EXE_BUBBLE_GRALL(**user**) − BE_EXE_BUBBLE_GRGR(**user**) Cycles stalled on pending integer loads.
L1D/FPU micropipeline stall	BE_L1D_FPU_BUBBLE_ALL(**user**) Cycles stalled on the L1D/FPU micropipelines.
front end bubble	BACK_END_BUBBLE_FE(**user**) Cycles the back end is stalled due to the front end not supplying instructions to execute.
branch misprediction flush	BE_FLUSH_BUBBLE_BRU(**user**) Cycles stalled during flush after branch misprediction.
register stack engine	BE_RSE_BUBBLE_ALL(**user**) Cycles in which execution is stalled due to register stack engine operation.
miscellaneous user	BE_FLUSH_BUBBLE_XPN(**user**) Cycles flushing after exception detection. + BE_EXE_BUBBLE_GRGR(**user**) Cycles stalled on scoreboarded integer unit results. + BE_EXE_BUBBLE_ARCR_PR_CANCEL_BANK(**user**) Miscellaneous stall cycles, including AR, CR, or PR dependences, load cancellation, or bank switching.
kernel	CPU_CYCLES(**kernel**) Cycles in kernel code.

A.6 Performance Monitoring

The experimental results presented in this dissertation are entirely based on real-machine measurements. This would not have been possible without the extensive performance monitoring support built into the Itanium 2 design or without the Perfmon features developed for the Linux kernel by Stephane Eranian of Hewlett-Packard Laboratories [63]. A few words are in order regarding these facilities.

Table A.1 indicates the performance monitoring measurements comprising the various categories shown in the cycle accounting figures throughout the dissertation (Figure 2.6, for example) [51]. The categories (and underlying hardware counters) are set up in such a way that they are mutually exclusive, so the counters taken collectively are an accurate summary of the state of the back end throughout execution. The Itanium 2 front end is

decoupled from the back end by means of a small instruction buffer, so while it is possible to determine in which cycles the back end is stalled due to an inability of the front end to deliver instructions, it is not always possible to definitively determine the reason for the front-end stall.

- **unstalled execution** This category counts all user (nonkernel) cycles in which at least one instruction retired. Assuming that no latencies were exposed and that no run-time anomalies (such as branch mispredictions or exceptions) occurred, this would be the total number of execution cycles for the application. Early experiments in EPIC architecture/compiler design, in their simulation models, effectively assumed this to be the case [38].

- **floating-point scoreboard** Here are accounted cycles of back-end stall due to dependences on scoreboarded floating-point register values, including products of floating-point arithmetic and conversion operations and floating-point loads. Unfortunately, there is no obvious means of distinguishing between load-related and non-load-related floating point stalls.

- **integer load bubble** This category includes cycles during which the main pipeline is stalled on the outcome of a pending integer (non-floating-point) load. On average, 25% of execution cycles are expended in this manner. One subtlety is worth pointing out with respect to the measurement of this category: store-load dependence cycles mentioned in Section 8.3 are largely accounted here. Addressing spurious store-load dependence sites reduced this category of cycles by half in *256.bzip2*, without much effect on the L1D micropipeline stalls. This was contrary to what had been expected, based on the relevant literature [51]. This category does not reflect the total number of cycles stalled due to data loading, due to the inclusion of floating point load stall cycles in the previous category and the accounting of certain exceptional cases to the next category, L1D/FPU micropipeline stall.

- **L1D/FPU micropipeline stall** Here are found cycles accounted to main pipe stalls due to the L1D[2] cache access unit or, less frequently (always less than 1% of total execution cycles, and generally an insignificant amount), the floating-point subpipeline. L1D events account for the vast majority of this significant category of cycles, which accounts for 9%, on average, and up to 22% (in *vortex*), of execution cycles. FPU micropipeline contributions are insignificant, except in *vpr*, where they contribute 0.6% of execution cycles.

  L1D stalls fall into 12 categories, which are not mutually exclusive (more than one of them may be occurring in any given cycle). These are as follows, listed roughly in decreasing order of importance. The percentage of **O-NS** and **I-CS** cycles in which each activity is operating are given at the start of each description:

  - **Data cache unit recirculation (`L1D_DCURECIR`)** (8.5-9.3%) Stall due to the data cache unit recirculating. These cycles, the most numerous category, are secondary effects of other events, or of collisions of other events. While their magnitude is important, they offer little in the way of diagnostic guidance. Those not accounted for by production from other L1D micropipeline events may be related to collision of access requests in the memory system (two simultaneous loads to the same L2 cache line, for example).

  - **Hardware page walker (`L1D_HPW`)** (2.1-2.2%) Stall due to activity of the hardware page walker after a TLB miss. A single access to the HPW takes at least 25 cycles.

  - **Level 2 back-pressure (`L1D_L2BPRESS`)** (1.0-1.5%) Stall due to too many outstanding requests) in the L2 OzQ (out-of-order request queue). When the 32-entry OzQ is full, the L2 substructure applies back pressure to the L1D unit, stalling issue until an OzQ entry becomes available.

[2]This name appears to be something of a misnomer. The L1D is the primary cache. The L1D units manage the execution of all loads and stores, whether they hit or miss in the first-level data cache.

– **Store buffer cancellation (`L1D_STBUFRECIR`)** (0.6%) Stall due to a store buffer cancellation requiring recirculation. These occur when a store and load accessing the same cache line occur within three cycles of each other, in that order. These become significant in *bzip2*, where they account for 4.8% of execution cycles; in other cases, they account for less than 1.0%. These are an indication of the behavior described in Section 8.3.

– **Translation lookaside buffer transfer (`L1D_TLB`)** (0.4-0.5%) Stall due to transfer of line from second to first level TLB. A cycle accounted here will also reflect 3 cycles of recirculation (above) and 5 cycles of load stall.

– **Fill conflict (`L1D_FILLCONF`)** (0.4%) Stall due to a store conflicting with an ongoing fill. These exceeded 1% of execution time only in *gzip*, and then only barely.

– **Full store buffer (`L1D_FULLSTBUF`)** (0.0%) Stall due to the store buffer being unable to accept another store. These never account for more than 0.5% of execution cycles in the experiments.

– **Not-a-thing (NaT) generation (`L1D_NAT`)** (0.0%) Stall due to a need for recirculation to perform `NaT` generation. These occur only in code that uses control speculation, but still do not account for only insignificant numbers of cycles (less than 0.5% in all cases).

– **NaT spill/fill conflict (`L1D_NATCONF`)** (0.0%) Stall due to a conflict between spill and fill operations over the `unat` register. These were not significant in the experiments.

– **Load ordering conflict (`L1D_LDCONF`)** (0.0%) Stall due to architectural load ordering conflict. These did not occur in measurable numbers, since they related to load and store operations with specialized acquire and release semantics, which are only infrequently used in the studied benchmarks.

  - **Load check ordering conflict (L1D_LDCHK)** (0.0%) Stall due to a load check ordering conflict. These did not occur in measurable numbers, since data speculation was not applied in the experiments.

  - **DCS (L1D_DCS)** (0.0%) Stall due to access of system registers, never significant in this dissertation's experiments. (DCS is not defined in the relevant literature.)

- **branch misprediction flush** These are cycles during which the back end is stalled due to the pipeline flush that occurs after detection of a branch misprediction, approximately 6 cycles for a typical "whether" misprediction. The average CINT2000 benchmark exhibits a 29% reduction in branch misprediction flush cycles in I-CS mode, relative to O-NS. Branch misprediction flush, however, typically contributes only about 6% of total execution time, so this effect is not a primary factor in performance improvement due to prediction and region formation, as the conventional wisdom might suggest [39].

- **register stack engine** This counter tallies cycles of back-end stall due to operation of the register stack engine (RSE). The RSE spills and fills general registers as required by allocation and deallocation requests in the program call sequence. For most benchmarks, this category is relatively insignificant, as function inlining has eliminated most heavyweight callsites. ILP optimization, including inlining, though, tends to increase register consumption. If frequently traversed call structure remains after ILP optimization, then it is likely this category will be impacted. RSE activity is significant in *crafty*, in particular, in which 6% of cycles are spent in it prior to ILP optimization, and this number of cycles increases by half after optimization. *gcc*, *parser*, and *vortex* show similar trends, although the initial contribution is smaller. *eon* spends a constant 4% of its time in the RSE.

- **miscellaneous user** This category includes a handful of infrequent events, including exception flush, integer unit stalls, and various system register and bank switching penalties, as indicated in Table A.1.

- **kernel** Here are accounted all cycles spent executing kernel code.

A.7 Conclusions

Compiler work—particularly that which determines what degree of instruction-level parallelism the compiler can extract at a reasonable efficiency from general purpose programs—is highly relevant to the scaling of the Itanium Processor Family to future generations. Averaging across SPEC CINT000, for the nominally 6-issue Itanium 2 microprocessor, IMPACT and commercial compilers today produce plans of execution with, on average, 3.03 useful operations per cycle. These plans achieve an actual issue rate of about 1.29 useful operations per cycle. (Here a useful operation is defined as a non-no-op operation whose predicate evaluates to 1.) Slightly more time is spent in (largely unscheduled) stalls than is spent in cycles retiring instructions. This carries interesting implications for the furtherance of instruction-level parallelism in EPIC machines.

APPENDIX B. THE IMPACT COMPILER

The IMPACT compiler provides a full-featured environment for exploitation of ILP in C and C++ programs. Today IMPACT generates code either for simulation on a generic, experimental VLIW machine or for execution on the Intel Itanium 2 microprocessor [51]; the latter path is the one exercised in this dissertation, but the information contained in this appendix is generally also applicable to the generic path. This appendix serves three purposes: First, it provides details necessary for other compiler developers wishing to evaluate this work. Second, it serves as an instructional resource for future users of the compiler infrastructure developed in the course of this dissertation. Third, it provides a concise, critical review of IMPACT as it stands today, as a guide to future developers considering either an extension of IMPACT or the development of a new ILP compiler infrastructure.

B.1 Overview

Compilation of a C program through IMPACT, shown at a very high level in Figure B.1 can be divided into eleven general stages. When IMPACT is run using the OpenIMPACT command-line driver "oicc," the first stage appears to be the "compilation" or ".c → .o" stage; the second appears as a "profile linking" stage; the third as a profiling stage; and the fourth through eleventh as a single interprocedural optimization stage, culminating in production of the linked executable.

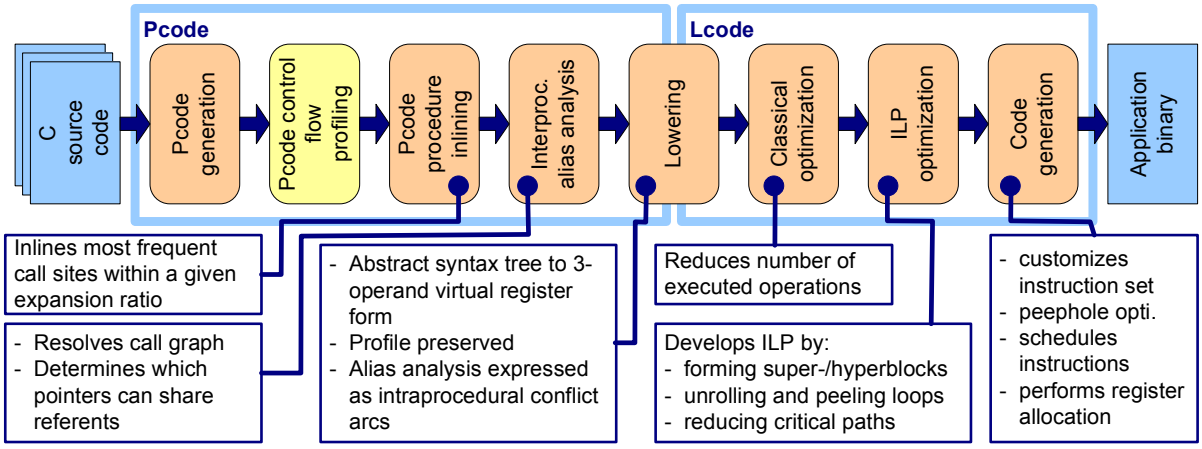


Figure B.1 The IMPACT compiler: high-level phase ordering.

Of particular note in this regard are the constraints imposed by phase ordering, which can limit the effectiveness and the controllability of ILP transformations, and the generally local nature of most of the important later phases, which can allow accumulation of surprisingly negative results at the whole-program level. Some of these issues were addressed in general in the dissertation.

B.2 Pcode Generation

This is the first detailed publication to use a totally rewritten IMPACT Pcode front end (mostly written by Robert E. Kidd under the auspices of the Gelato Initiative's OpenIMPACT project). This portion of the compiler will therefore be described in considerable detail.

C/C++ source code is preprocessed using another host compiler (*gcc*, for portability and convenience), parsed using the Edison Design Group C++ Front End (EDGCPFE), and translated into Pcode, the IMPACT high-level intermediate representation (IR), a relatively conventional abstract syntax tree representation. One Pcode file is generated per source code (.c/.cpp) file. Symbols are reconciled across these files in the next step of processing.

In a feature added for this dissertation work, C++ source code is handled in the normal fashion, except that EDGCPFE, in addition to parsing the preprocessed C++ into its internal representation for re-expression into Pcode, also lowers its C++ internal

representation into the C-style form that Pcode is used to receiving.[1] While this provides rudimentary C++ support in the manner of the earliest C++ compilers (i.e., the *cfront* approach), it is far from an optimal solution. Historically, this has not proven a very effective means for dealing with C++ code, since important semantics of the C++ language (such as those necessary to analyze and devirtualize performance-penalizing virtual function invocations [126]) are lost in the lowering process. This accounts for some portion of IMPACT's poor performance in the *eon* benchmark. To handle C++ properly, IMPACT's Pcode representation should be extended to support C++ abstractions and to provide basic C++-targeting optimizations.

B.3 Pcode Linking

The gathering together of the entire application's (library's) Pcode starts the IMPACT compilation process in earnest. Conventional linkage rules are applied to match symbol uses to definitions. The remainder of IMPACT operates within the interprocedural context created by this step. The Pcode linker (`Plink`) has to go somewhat beyond the duties of the ordinary linker (`ld`) in linking the application's Pcode files together, as Pcode is a typed representation. Since subsequent optimizations will perform cross-file inlining and optimization, the files need to share a consistent view of all the types defined across the application. This is not a trivial problem, as types are typically defined in header files, included in different inclusion paths and in various contexts in different source code files. Since the typical linker does not check types, programs often contain inconsistencies among the different source files. Different structures share common names; the same structure may have different names in different files, but its instances may be used interchangeably in the interprocedural context. The same is true of procedure invocations. The Pcode linker often finds (generally indirect) procedure invocations whose arguments do not match those of the function being called.

The Pcode linker generates a new Pcode symbol table file named "`a.out`" and a unique key for each symbol in the application. This key is used, rather than a symbol

[1] EDG also provides exception-handling and C++ runtime libraries that are linked with the application.

name, throughout Pcode optimization to prevent symbol conflicts. The key consists of a (file, id) pair. Symbols that rise to the global level are indicated by file number "1," indicating that their records are located in the top-level symbol table. Other symbols remain in their individual Pcode file symbol tables for scalability reasons. The top-level symbol table provides a structure facilitating random access throughout the entire application. This has revolutionized the design of most subsequent Pcode stages.

Most of IMPACT today still operates at the single-function level, but latter phases will still benefit from the summarized results of interprocedural analysis. Because of a strong need for improved whole-program optimization, we are in the process of improving the quality, accessibility, and persistence of the interprocedural view of the program created here.

The Pcode is also lowered and flattened in this phase (by the module `Pflatten`), meaning that some complex expressions are broken down into simpler elements to ease subsequent transformations.

B.4 Pcode Profiling

A probed version of the application is produced (via `PtoC` and the host compiler) and is run with training input(s). Probes gather control flow arc traversal counts and loop iteration counts, which are then annotated into the Pcode IR (by `Pannotate`). Procedure calls through function pointers (indirect procedure invocations) are also profiled in this stage, to give guidance to the inliner. Both the high-level (Pcode) and low-level (Lcode) representations can be reprofiled at nearly any stage of compilation, although this would not be very practical in a production environment. Subsequent transformations generally try to preserve and extrapolate these initial profiling results, but can do so with only limited success. The substantial value of a second profiling stage performed after classical optimization (any time after inlining would actually do; this is just a convenient location for a second profile in the IMPACT framework) was discussed in Chapter 7. The IMPACT compiler used in the described experiments used such a second profiling pass. It is possible that, if a certain degree of *context-sensitive* profiling information could be gained in the Pcode profiling stage, this step could be eliminated without danger to performance.

B.5 Pcode Optimization (Inlining)

Procedure inlining [46, 74] is the only significant component of the Pcode-level optimizer today. Procedure inlining provides three basic benefits: (1) elimination of call mechanism overhead; (2) specialization (optimization) of the inlined body for its calling context (the classical benefit); and, (3) the effective formation of instruction-level parallelism across call sites (a benefit particularly required for EPIC machines). Apart from inlining, independent code from the caller and callee cannot commingle in an EPIC machine. Inlining is performed after profiling, so that it uses profile data to find what appear to be the most beneficial sites to inline (as well as to identify the likely targets of indirect procedure calls). Inlining heuristics basically view reducing the number of procedure invocations as the optimization objective, and will inline the "hottest" calls[2] until some fixed maximum inlining ratio (typically a 2× increase in touched expression count) is reached. Indirect procedure calls are profiled and inlined using the procedures outlined in Section 7.2. Limited inlining of recursive cycles is supported. Further details of the inlining routines and their effects in these experiments were described in Chapter 7 of the dissertation and will not be repeated here.

B.6 Pcode (Interprocedural) Analysis

The Pcode interprocedural analysis (IPA) phase provides later stages of the compiler with precomputed memory access disambiguation information. Because the accuracy of this analysis exceeds that available in production compilers, IMPACT has a unique opportunity to perform memory optimizations. In addition to providing immediate performance benefits, these optimizations in turn expose other important problems as the next-limiting critical path features, creating opportunities for region selection and other ILP transformations to reap large gains. This module was substantially altered during this dissertation work.

[2]The actual priority function is calculated as the call weight divided by the square root of the callee size, to give some preference to smaller callees.

In the past, memory disambiguation information was computed using Cheng's interprocedural, flow insensitive, context sensitive alias analysis algorithms [150, 151]. An Omega test [71] was added to provide refined loop array dependence information, which is of only occasional importance for SPEC CINT2000 applications. The information derived from alias analysis was materialized in only one way: access conflict arcs (*sync arcs*) which represent a pairwise, memory-carried, may-alias dependence between two given low-level (Lcode) operations.

Four deficiencies were encountered in this work, with respect to the past IPA approach. First, the analysis could not be relied upon to deliver conservative results in certain cases, leading to incorrect optimizations. Second, the analysis exceeded time or machine resource limits for the more complex benchmarks, including *eon*, *perlbmk*, and, especially *gcc*. Third, the representation of dependence information as arcs often became dauntingly large, substantially slowing subsequent optimization phases. Fourth, while the arcs represent the result of interprocedural optimization, they are inherently intraprocedural, as they are drawn only between two operations in a single function. This makes them a practical mechanism for scheduling and some optimizations but does not, for example, permit substantial interprocedural optimizations, such as inlining, after the IPA results have been materialized. Furthermore, alias analysis cannot be rerun on the low-level representation, so any interprocedural transformations performed at the low level would be forced to behave conservatively with respect to most memory accesses. This last objection would prevent some meaningful areas of future research. The combination of these factors required a new approach.

Erik Nystrom's FULCRA Interprocedural Pointer Analysis System [59] was therefore incorporated into the IMPACT Research Compiler to support this dissertation work. (The IMPACT module name is `Pipa`.) FULCRA, unlike Cheng's analysis, supports several dimensions of configuration (see [59] for details):

- **Assignment directionality.** One may select either Andersen's formulation ("subtyping"), which has directional assignments, or Steensgaard's formulation ("unification"), which combines any two nodes that point to the same object. This dissertation uses Andersen's, which is more accurate.

- **Field sensitivity.** An analysis is field-sensitive if it distinguishes among fields of structures. This dissertation used the field-sensitive formulation.

- **Context sensitivity.** Context sensitivity means that effects of different invocations of the same function are distinguished. This, too, was enabled.

- **Heap cloning.** Anonymous, dynamically allocated structures are identified by the memory allocation site at which they were generated (`malloc`, `calloc`, etc.). A n-level heap cloning approach considers these sites to be distinguishable by a certain level of context, so that two sites with invocation records differing within n levels are distinguished. This is useful for programs that encapsulate calls to the basic allocation routines within allocator or constructor functions. Here, three-level heap cloning was performed. Three levels generally capture all available benefit without unduly increasing analysis time.

- **Pointer arithmetic safety.** Since C is not a type safe language, pointer arithmetic poses special challenges for IPA, especially in a field-sensitive configuration. FULCRA implements a new approach to field sensitivity that models these interactions in a low-level, offset-based fashion, based on the analysis of pointer arithmetic expressions. This feature was required by SPEC CINT2000, particularly *gcc*, which overlaps arrays and aggregates in complex ways.

With these configurations, the FULCRA analysis ran to completion in reasonable time (hours in a few cases) and provided correct[3] results for the studied benchmarks. Like Cheng's analysis before it, FULCRA requires a whole-program view. This entailed writing pointer behavior summaries for all library procedures invoked across SPEC CINT2000 (This is a particular challenge for a C++ application like *eon*.) The problem of missing code needs to be addressed for FULCRA to be more practically usable outside of fixed benchmark suites.

Aside from the problems inherent to the analysis itself, which FULCRA addressed handily, it was determined that a new representation for memory dependence information

[3]Correct means "without indicating a defect resulting in compilation failure or detected program corruption."

was required. The sync arc-only representation for dependence was replaced with a new, two-part representation, consisting of both sync arcs and *access specifiers* (also known in some circles as *sync variables* or *store sets*). In general, a load, store, or subroutine call is annotated with a set of access specifiers, each of which specifies a node number, heap cloning subscript, access offset (or 0 if unknown), and access size (-1 if the offset or size is unknown). A specifier-by-specifier comparison of the access specifier lists of two accesses reveals whether the accesses potentially conflict. This approach avoids the costly generation of sync arcs, while avoiding the need to keep the points-to graph throughout the compilation process.[4] For arc-specific information, however, like dependence distance, access specifiers are insufficient. For this reason, if dependence distance is known as a result of the Omega test, sync arcs are added specifically to represent this information. This compromise representation proved satisfactory for the purposes of this dissertation's experiments.

B.6.1 Auxiliary low-level disambiguator

In addition to the interprocedural analysis, IMPACT also implements a variety of simple disambiguation techniques in the Lcode back end. These rely on operation markings (attributes) generated by `PtoL` in the lowering from the Pcode high-level representation and on simple, local relations. The `Lssaopti` module improves these annotations, increasing the effectiveness of these simple mechanisms. It is important to note that there is a difference in character of the results from these analyses and the results of IPA—the local relations can deliver "must-alias" information in addition to the "may-alias" information provided by IPA.

B.6.2 Empirical evaluation

The effect of IPA on the performance of the SPEC CINT2000 benchmark suite was measured as part of this dissertation's experiments. On average, IPA yielded a speedup of

[4]Some compilers keep the points-to-graph, and perform queries directly on it, throughout the compilation process; see [152] for an example.

1.13× relative to a baseline using only the auxiliary low-level disambiguator[5] The suite's sole C++ benchmark, *eon*, stood out as exhibiting a 1.44× speedup with interprocedural analysis. *vpr* and *vortex* also stood out as benefiting particularly from IPA, exhibiting speedups of 1.28× and 1.18×, respectively. A simple, field-insensitive, context-insensitive analysis achieved the maximum speedup for *vortex*, but context sensitivity and heap cloning were necessary to maximally exploit *vpr*.

Ghiya, Lavery, and Sehr evaluated the importance of points-to analysis in the Intel production compiler using performance experiments on the Itanium (not Itanium 2) processor [152]. Their results are not directly comparable, since they use a different interprocedural technique and stronger simple disambiguation techniques (for example, disambiguation of indirect references and global, non-address-taken variables—IMPACT lacks this feature, which looks worth adding), but the comparison is still of interest. Interestingly, the benefits seen in IMPACT with respect to *vortex* and *vpr* are not indicated in their results. With respect to *vortex*, this is likely due to the lack of extremely aggressive Superblock formation in the Intel compiler, which is likely necessary to see the benefit of reordering enabled by pointer analysis. In *vpr*, a substantial gain is reported by [152], but it is with respect to global, non-address-taken variables—something that should be unrelated to the heap cloning benefits IMPACT identifies. There is clearly an opportunity for further investigation here, but space and time constraints prevent its resolution here.

B.6.3 Indicated future work

In the future it may be necessary to derive additional Itanium-specific analysis products from this phase. There is a microarchitectural need to find stores and loads to the same cache line, not just to overlapping addresses, to avoid spurious forwarding penalties [135]. This could make a difference in only a few benchmarks today (*bzip2* in particular) but could become more important with increasingly aggressive optimization. More

[5]Since the IMPACT implementation of Omega test [71] in `Pomega` and the supplemental arithmetic dependence analyzer in `Lssaopti` both require information from `Pipa`, these analyses are not available in the baseline version, even where they could trivially apply.

significant is the need to identify "wild speculative loads," those speculative loads that access nonexistent or forbidden pages other than page zero. These can have a devastating performance effect in the general speculation model. Chapter 6 provides more detail on these events and some early schemes for their avoidance. Finally, Intel's compiler benefits extensively from loop transformation and prefetching based on high-level analysis and transformations in *mcf*, *parser*, *eon*, *perlbmk*, *gap*, and *twolf* (these differences are visible between `icc` compilations with `-O2` and `-O3`). IMPACT lacks these features and so lags in performance in most of these applications. Continued study of ILP benefit in these specific applications should be in the context of high-level-optimized code.

B.7 Lcode Generation

The abstract-syntax representation of Pcode is translated in a conventional manner [49, pp. 463–508] by the module `PtoL` into the three-address form Lcode to be used for all subsequent compilation phases. Most nonaliased, scalar automatic variables are promoted to virtual registers in this stage. Other automatic variables are "materialized" into their locations in the activation record. Unfortunately, aggregates (structures and unions) are not dealt with very aggressively in this stage. These are simply materialized to the stack. Their fields have no opportunity, in this stage, to be register promoted. Any aggregate assignments, even those not explicitly written by the programmer but generated due to flattening, are therefore generated as sequences of the largest possible loads and stores allowed under the default structure alignment. Subsequent optimizations are relied upon to clean up this mess. In some applications, most notably *crafty* and *eon*, they are only partially successful in doing so, and performance suffers. Pcode-to Lcode-lowering should be enhanced to do a better job of dealing with these aggregates. Less importantly, objects materialized to the stack, but subsequently register-promoted by optimization, still consume stack space. This is largely just a minor annoyance, but could become significant in programs with great call depth.

Pcode is a typed representation; Lcode integer registers have no inherent type (i.e., an Lcode virtual register is not a `char`, `short`, or `int`; it is simply a register of the default machine register size). This means that the Lcode generation is the last place to exploit

type information in selecting implementation of operations. `PtoL` was extended to deal with this in limited cases. Divide operations in the Itanium architecture are not single instructions, but successive approximation sequences. A divide of 8-bit quantities is much shorter and faster than a full 64-bit divide. For this reason, divide and modulus operations are marked in `PtoL` with the size of the operation. These markings are used in the code generator, `Ltahoe`, to generate the appropriately tailored Itanium code sequences. In Lcode, such complex operations as divide are left as machine independent macro-operations to allow for more effective classical optimization. This is necessary to a certain degree to enable optimizations (such as strength reduction) which would otherwise have to be rewritten to recognize and contend with complex operation sequences. It is also, however, sometimes an obstacle to effective optimization as the Lcode operations do not necessarily reflect the number or cost of operations that will result in the lowered, machine level IR. As part of the same type lowering operation, sign- and zero-extension instructions are inserted liberally in `PtoL` to ensure language standard compliance. These are removed by subsequent optimizations in `Lopti`, `Lssaopti`, and `Ltahoe`.

The access specifiers and sync arcs derived in `Pipa` and `Pomega` are applied from Pcode expressions to the appropriate, lowered Lcode instructions to provide memory dependence information for the compiler's low-level optimization phases.

B.8 Lcode Optimization I: Classical Optimization

Low-level optimization begins with classical "Dragon Book" optimizations [49, pp. 585-680] of the Lcode, focused on classical optimization objectives—reduced number of operations executed, local simplification of control flow, minimization of memory access, etc. This optimization phase does not perform the voluminous code replication associated with later, CFS-focused stages. The module `Lopti` provides such optimizations as:

- **Branch and jump optimizations.** Indirect branch expansion (pulling common cases out of branch tables to enhance predictability and encourage subsequent optimization); decidable branch elimination, block coalescing, limited branch target expansion (no code growth).

- **Dead, unreachable, and inconsequential code removal.**

- **Local and global optimizations.** Constant propagation; constant folding; forward and reverse copy propagation; common subexpression elimination; redundant load and store elimination; memory copy propagation; strength reduction; constant combining; operation folding; sign extension elimination; logic and arithmetic reduction; lightweight predicate define network optimization.

- **Additional global optimizations.** Upward and downward instruction unification; disjoint virtual register renaming and coalescing.

- **Loop optimizations.** Loop invariant code motion; loop global variable migration (register promotion in the presence of aliasing); loop branch simplification; induction variable strength reduction; inductor combination/elimination.

- **Partial code elimination.** Partial redundancy elimination (with or without speculation), partial dead code elimination, dead store elimination, all with critical edge splitting [88].

These optimizations are performed on virtual register-based computation, not on variable accesses as might take place in Pcode. To expose as much code as possible to optimization, IMPACT adds transformations which "register-promote" as many variables as possible, using memory dependence information to detect aliasing and compensation code as required to safely contain aliased accesses. This simplifies the optimizations, since they need not deal with the complexities of potentially aliased memory operations, but can create situations which oversubscribe machine register resources, causing insertion of spill and fill code much later in the compilation process. This effect is not obvious to most optimizations, and so is difficult to avoid, but in practice does not seem to occur with alarming frequency.

Recently IMPACT's classical optimization suite has been modernized with more advanced tools, including partial redundancy elimination (PRE), partial dead code elimination (PDE), and single-static assignment (SSA) environments [61, pp. 252–258]. The

PDE concept has been enhanced to enable a greater degree of store-sinking by the addition of predicate guards. This technique enables loop-carried partially dead store elimination in the presence of potentially aliased uses on infrequent loop paths. This is a very sophisticated technique and makes elegant, if unorthodox, use of hardware support for predication.

The author has developed an SSA phase, embodied in the module `Lssaopti`, used mainly at this point for performing sparse analyses. These analyses are currently used to remove sign and zero extensions, an important problem in 64-bit architectures, and to improve the performance of load/store optimizations by examination of access expressions and annotation of useful properties.

These new, more systematic optimization engines are displacing the older, less general, less controllable, and less maintainable tools. They have strengthened IMPACT's global optimization capacity, which has traditionally been relatively weak. IMPACT has historically relied heavily on all important-to-optimize code being collected into Superblock or Hyperblock regions, where it would be subject to simpler but more powerful local optimizations. This posed a problem for a couple of reasons: First, it assumed that effective region formation can take place on poorly optimized code. Practical experience, however, indicates that such post-formation optimization can change good formation decisions into bad ones and vice-versa [48], so it is beneficial to produce the best possible code before applying region formation techniques. Second, for large, complex, modern applications, meeting the expectation that all important code is neatly enclosed in large Hyperblock regions may require an unacceptable level of code growth. Hence, code outside and across these regions may play an important role in performance. As IMPACT develops mechanisms to further reduce unnecessary code expansion, an effective base of global optimizations will be essential in reducing the performance impact of doing less "whimsical" region formation.

B.9 Lcode Region Formation: Parallelism Cultivation

One of the most critical elements in generating high-performance code for EPIC systems is the transformation of programmatic control flow into a form suitable for efficient

execution on the target machine. The basic principles applied here were described in the seminal Superblock [22] and Hyperblock [36] papers. Paths through an acyclic code region are enumerated, and (for Hyperblock) the dependence height and instruction issue requirements of each are determined. Heuristics are applied to select the region's most important and most compatible paths on the basis of these measurements. Hyperblocks may additionally pull a finite number of iterations of enclosed or adjacent loops into the region using loop peeling transformations (described in Section 5.4.4). Relative to the approach taken in previous publications, much has been added to improve the inclusion/exclusion decisions performed and to expose a greater portion of total executed code for inclusion into efficient Hyperblock regions. Much of this work was material to this dissertation, so it is described in Chapters 4 and 5, rather than here.

B.10 Lcode Optimization II: Parallelism Enhancement

After region formation, the IMPACT compiler performs a host of transformations intended to increase the degree of parallelism within the specialized code versions. These include induction variable transformations (expansion, reassociation, and elimination), register renaming, critical path reduction techniques (operation reassociation), and a variety of other parallelism-enhancers. These transformations occur in the module `Lsuperscalar`, in combination with a reapplication of the conventional optimization techniques employed already in `Lopti`. The latter are performed in the hope that specialization has happened to render new opportunities for these classical optimizations.

Today (in the SPEC CINT2000 benchmarks), much less useful optimization occurs after region formation than was once the case (in other benchmarks). The benchmark *vortex* is the one prominent exception, in which `Lsuperscalar` provides a 11% speedup from optimization alone (excluding the benefits of region formation). The benchmarks *parser*, *eon*, and*bzip2* each garner about a 2% benefit. For other applications, the non-region-formation aspects of `Lsuperscalar` cause no measurable gains.

B.10.1 Loop unrolling

The IMPACT compiler employs two loop unrolling schemata on loops consisting of single extended basic blocks (Superblocks or Hyperblocks). The first is the traditional technique; the second was developed for this dissertation work. In the primary schema, the loop body is replicated sequentially to the desired degree, and the loop continuation branches in all but the last loop stage are reversed, so that they become side-exit branches. Subsequent control speculation is relied upon to hoist instructions across these branches, allowing them to sink to the bottom of the loop.[6] Nonetheless, the accumulation of these side-exit branches complicates modulo scheduling and generally produces results that are not as good as for loops that have not been unrolled.

It is, however, in some cases desirable to unroll loops that will be modulo-scheduled. In the second schema, the loop is unrolled using unrolling stage predicates rather than side-exit branches. See Section 8.4 for details of this approach.

The loop unroller is controlled by regrettably simple heuristics, and operates relatively infrequently in the Itanium configuration (modulo scheduling often achieves better results than unrolling, probably due to infelicities in the IMPACT scheduling and register allocation infrastructure; see Sections B.11.2 and B.11.4). Unrolling is applied only to single-extended-basic-block loops.[7] Unrolling is limited by a maximum unrolling degree (8) and a maximum number of Lcode operations in the loop body after the unrolling operation (32). If the loop being examined is not a candidate for modulo scheduling (See Section B.11.3), the primary (side-exit) schema is applied. If the loop is a modulo scheduling candidate and meets the requirements indicated in Section 8.4, the second schema is applied. The benefit of unrolling depends almost entirely on the applicability of subsequent transformations. A more sophisticated implementation would evaluate the

[6] IMPACT's somewhat limited implementation of modulo scheduling only allows branches in the last stage of loops, making it critical that these branches be compressed to the end of the loop body, lest they unacceptably increase the height of the last stage, increasing the resulting initiation interval

[7] This means an extended basic block with a self-edge, not a loop with all paths contained in a single extended basic block. In many cases, only the "hot" path comprises the extended basic block, and only this block is unrolled.

consequences of transformations such as unrolling on subsequent optimizations on a loop-by-loop basis, rather than applying blanket limits. Recent work in optimization-space exploration pointed out the importance of such a diverse approach, albeit in a different compiler setting [153].

Other compilers apply more sophisticated unrolling techniques that take advantage of counted loops to eliminate side exit branches. The Multiflow compiler [25] performed preconditioning and postconditioning schema in its loop unroller. In pre-conditioning, a counted loop unrolled by degree N is preceded by $N-1$ bodies, each guarded by a conditional branch that initiates the unrolled loop if the number of iterations remaining is divisible by the unrolling degree. The postconditioned schema is a symmetric case (see [25] for details). A degenerate case of either pre- or postconditioning occurs when the loop iteration count is known at compile time and is an even multiple of the unrolling degree. In this case, side exit branches may simply be eliminated in the generation of the unrolled loop body, as they will never be taken.

B.10.2 Antidependence elimination

Achievable parallelism is enhanced by a variety of renaming and restructuring techniques that share the goal of removing antidependences. These optimizations are particularly effective in the case of unrolled loops, in which, for example, a single induction variable having an update in each unrolled stage can be converted into several parallel induction variables, each having only a single update. This sometimes results in the extrusion of compensation code at side-exits, which can occasionally prove troublesome for subsequent optimizations. The modulo scheduler deals with some additional cases of anti-dependence removal through its use of the rotating register facilities provided in the Itanium architecture (see Section B.11.3).

B.11 Machine Code Generation

Machine-independent Lcode is converted to machine-specific Mcode. Though Mcode is essentially similar to Lcode, this involves the assignment of machine-specific opcodes

and expansion of certain macro operations (including, on ia64, division). This creates the opportunity for additional optimizations (for example, those related to subexpressions of the expanded macros and ia64-specific predicate define sequences). Predicate promotion is applied to increase the availability of instruction-level parallelism. Local code transformations are applied to integrate inefficient small blocks of code into larger, more cache- and encoding-efficient regions. Code is scheduled once before and once after register allocation. Appropriate loops are modulo scheduled using a rotating-register, explicit prologue and epilogue schema.

B.11.1 Fine-tuning optimizations

Small transformations are made to the code, some replicating instructions, to enhance cache performance. These optimizations are similar in character and outcome to those described in [154, 155].

B.11.2 Instruction scheduling

The IMPACT compiler performs two acyclic and one cyclic scheduling phases using the general scheduling [156] and machine description [157] facilities developed by Gyllenhaal. Both were extended by Ueng [158] to manage the bundling problem and other special features of Itanium.

The scheduler is frequently a source of seemingly arbitrary changes in code performance, the place where chains of unrelated events culminate in a surprise performance degradation. This problem underscores the need to examine actual code, and not just performance numbers, to determine the degree of success being had by transformations. One good example comes in the calculation of *dependence height and speculative yield (DHASY)* heuristics that govern the priority order in which operations are scheduled. IMPACT's scheduler uses the computation detailed in [65] (originally of [109]), in which the difference between an operation's late time and the latest late time in a control block is a multiplicative weight in the priority function. This has the effect that low-probability block tails can affect the priority of important operations early in the block, sometimes creating suboptimal schedules. When the value produced by an instruction is not live

along all exit paths, the length of the tail can affect the priority of the instruction relative to other instructions, when the length of the tail in reality has nothing to do with their relative priority. This is a weakness in the DHASY model.[8]

In acyclic regions (everywhere but in software pipelined loops), the instruction scheduler schedules greedily (top-down, minimum-latency, earliest-ready-time scheduling of each operation), without regard for the effects of such scheduling on register pressure. When a substantial degree of slack is available on certain operations, or when the available parallelism vastly exceeds the execution resources available, the schedule generated may use more registers than necessary to secure the resulting total execution time. In extreme cases, spill and fill code may be inserted, with devastating performance effects. In less extreme cases, the result is more register stack activity than necessary at function invocation sites. The incorporation of a register pressure sensitivity technique such as that proposed by Schlansker and Kathail [142] and a rematerializing register allocator [61, pp. 488 ff.] would help solve this serious problem.

Evidence has been noted in this Itanium research of the potential for benefit from schedule-time code transformations, confirming some past IMPACT work in the area [156]. Often dependences can be restructured, allowing better scheduling, by simple transformations of the instructions being scheduled. The advisability of these transformations becomes apparent only during scheduling, when critical dependences are exposed and open scheduling slots are readily identifiable. These transformations have not been implemented in the IMPACT scheduler, however, due to the complexity of scheduling into the architectural bundles of the Itanium architecture.

B.11.3 Cyclic instruction scheduling

IMPACT's modulo scheduler is based on the algorithms of [159]. These have been extended to support both predication and the rotating-register schema supported on Itanium.

[8] An example of this problem occurred in *300.twolf*, but subsequent region formation changes eliminated it, so it cannot currently be reproduced.

IMPACT uses a rotating-register, explicit-prologue, explicit-epilogue model for modulo scheduling [160], despite the slight code expansion this incurs, on the theory that it provides better overlap of the wind-up and wind-down phases with surrounding code, enhancing ILP. A general attempt is made to cast loop bodies to be modulo-scheduled into fully-resolved predicate (FRP) form [92] to allow the branches to be compacted at the end of the loop, but this is not always possible after the loop formation has occurred.

Software pipelining is performed only on loops that have been rendered into single PEBBs with single self-loop-back edges. This simplifies the procedure greatly, and is successful because many loops can be so transformed, especially when predication is employed. Control handling in modulo-scheduling regions is, however, suboptimal. IMPACT does not currently use the counted loop support provided in Itanium [57]. Since loop-back branches are predicted-taken, this causes the misprediction of one branch per loop invocation and requires the use of control speculation (rather than stage predicates) for speculative stages. Finally, the scheduler disallows scheduling of branches in stages other than the final stage to simplify (and reduce the size of) epilogue tails.

Despite all these restrictions, modulo scheduling is often a profitable transformation, even in the nonnumeric codes of SPEC CINT2000.

B.11.4 Register allocation

The register allocator is a hybrid of Chaitin's [161] and Chow and Hennessy's [162] algorithms, extended to support predicate-sensitive allocation in the spirit of [94]. Several potentially profitable extensions could be proposed: (1) The register allocator does not currently support rematerialization [163], although it appears this would be a profitable transformation for IMPACT, particularly on Itanium. Because access to the data store is through a relocation pointer on Itanium, and because Itanium uses single-address-operand load and store operations, many of the values with the most scheduling and hoisting freedom are labels. These labels are more cheaply regenerated than spilled and filled. (2) The register allocator also does not yet support the efficient use of multiple UNAT registers, as is necessary for complex functions optimized for recovery code mode, or the efficient spilling and filling of predicate registers, as is rarely required in

practice [57]. Finally, (3) the register allocator would benefit from more interprocedural information to support its choice of callee- and caller-saved registers, particular in the context of the register stack.

B.11.5 Postpass

After register allocation, acyclic instruction scheduling is run again to allow absorption of spill code. Static branch prediction and instruction prefetching hints are inserted, and minor instruction cache alignment adjustments are performed. The resulting code is output in either Intel-format or gas-format assembly.

B.12 Machine Code Linking

IMPACT-optimized assembly code files are assembled and linked using conventional tools. If general control speculation is used, the binary is marked with `chatr` to disable deferral of speculative exceptions.

APPENDIX C. DETAILED BENCHMARK PERFORMANCE RESULTS

This appendix provides detailed EPIC performance insights specific to the various C and C++ benchmarks comprising SPEC CINT2000 [5, 62]. These benchmarks can be classified as *compute-intensive*, tending to emphasize CPU performance (including, to a minor extent, the performance of nearby cache structures) rather than other system components; *control-intensive*, characterized by frequent, relatively difficult-to-predict branches; *integer*, not dominated by either long-latency floating-point operations or by easy-to-parallelize, large, regular numerical computations. Within these general parameters, each benchmark offers a unique sample point for measurement of optimization effectiveness that is hopefully representative of a class of typical programs. The intention here is not to provide simply a tuning guide for SPEC CINT2000 on Itanium 2 (though this certainly is embedded here); rather, the SPEC CINT2000 suite is used to show concrete examples of certain general problems in EPIC compilation. A reading of this appendix should give the interested reader a relatively comprehensive view of the problems faced by an EPIC compiler in this program domain. The information contained herein serves three chief purposes:

- It is impossible to characterize a compiler's effectiveness for all programs in general and difficult to understand even concrete compiler-derived speedups without

detailed knowledge of the techniques applied. This detailed evaluation of the IMPACT compiler's effectiveness in optimizing well-known, readily available benchmarks on an easily characterizable, readily available system is intended to allow the reader to assess concretely the results of this dissertation.

- The only effective means for comparing compilers with the intention of transferring or improving optimizations is the analysis of specific code examples. This appendix is intended to speed the assessment of techniques developed in this research vehicle for incorporation into production compilers.

- In some instances, demonstrable opportunities exist for improvement beyond what was achieved in this dissertation. Their documentation here is intended to facilitate follow-on work in the research domain.

C.1 Summary of Performance Results

Table C.1 shows reportable and estimated[1] SPEC ratios for the benchmarks of the CINT2000 benchmark suite [5]. Each score is computed as one hundred times the ratio of a reference run time to the measured run time; a higher ratio thus indicates higher performance. For comparison, the graph includes results for contemporary versions of the GNU C compiler (GCC version 3.2) [164] and Intel's platform compiler (ICC versions 8.1.021) [35, 165]. GCC, the most commonly used compiler in the Linux-ia64 community, is unfortunately not a serious performance contender, so it warrants little further mention. These compilers are configured according to the descriptions provided in Section 2.3.1. The IMPACT compiler configurations are as described in Table 2.1, except for the **I-CS/R** column, which indicates the result of running the **I-CS** configuration with the benchmark reference input(s) used instead of the usual training inputs during profiling

[1] These results are denoted as *estimated* in accordance with SPEC rules [5]. For reasons of convenience, the IMPACT compiler was not run within SPEC scripts, although SPEC training inputs were used in the profiling stage and the compiler was run with consistent settings across the benchmarks suite. Measurement runs of the resulting binaries were run within the standard SPEC environment. These results accurately reflect those that would be obtained in a certifiable reporting run. GNU GCC and Intel ICC results were gathered on our systems in accordance with SPEC rules, and are reportable.

Table C.1 Estimated SPEC CINT2000 ratios for GNU GCC, Intel ICC, and IMPACT

compiler	GCC 3.2	ICC 8.1.021		IMPACT Compiler (20050409)					
config.	-O3	-O2	-O3	**O-NS**	**S-NS**	**S-CS**	**I-NS**	**I-CS**	**I-CS/R**
gzip	374	629	645	588	624	696	657	751	730
vpr	497	666	665	616	659	719	692	756	754
gcc	521	988	954	833	955	1066	977	1030	1045
mcf	333	335	692	330	335	323	331	338	336
crafty	489	801	806	657	695	709	175	745	777
parser	410	567	617	532	549	560	546	559	559
eon	273	895	1194	462	494	530	578	611	610
perlbmk	472	676	739	733	728	775	738	772	889
gap	375	601	661	569	607	651	597	630	649
vortex	550	1080	1081	867	1220	1393	1193	1382	1391
bzip2	414	662	746	610	624	654	737	763	781
twolf	557	798	923	738	755	816	812	884	894
GEOMEAN	430	697	792	609	656	699	683	730	743

stages. The difference between **I-CS** and **I-CS/R** columns gives an indication of the degree of meaningful profile variation between the training and reference inputs.

With respect to Intel's ICC compiler, IMPACT's peak performance (reflected in the **I-CS** column) makes a favorable showing, exceeding Electron's performance at `-O2` in nine benchmarks out of twelve, sometimes by more than 20%. With ICC configured with `-O3`, which enables data prefetching and high-level loop transformation optimizations not available to the IMPACT compiler (but presumably orthogonal to IMPACT's CFS techniques), IMPACT still delivers leading performance in half the benchmarks.

The IMPACT configurations are briefly described as follows: To increase the comparability of results, all configurations reflect the same degree of procedure inlining (maximum code growth ratio of 2.0× touched code size) and the same level of interprocedural pointer analysis (FULCRA analysis, configured to be context and field sensitive, using Andersen's formulation and heap cloning with a 3-invocation-level specialization limit [59]). The **O-NS** configuration serves as a baseline, including only classical code optimizations and none specifically designed to enhance instruction-level parallelism. It does not make use of control speculation, although it does make occasional use of predication in some optimizations (see Appendix B for details). This model is intended to reflect the best performance achievable with a traditional (i.e., non-ILP) compiler framework on the target processor. The **I-NS** configuration adds ILP region formation (including predicated

regions, or Hyperblocks) and ILP-enhancing ancillary transformations (height reduction, loop unrolling, loop peeling, etc.). The **I-CS** configuration, finally, adds control speculation. The **I-CS** configuration is the one put forth as the epitome in this work; the others are configured to illustrate the contribution of various optimization components. **S-NS** and **S-CS** configurations reflect the performance of Superblock-based approaches that do not take advantage of the predicated execution facilities of Itanium 2.

The column for **I-CS/R** reflects a run of the benchmarks in which the usual training inputs were replaced with the SPEC CINT2000 reference inputs. This configuration, while not legal under the SPEC run rules, gives the "ideal" case of oracular control flow profile information being available to the compiler. As indicated, only for the benchmarks *crafty*, *perlbmk*, and *gap* does this make a significant difference.

C.2 Cycle Accounting

Figure C.1 shows an accounting of the execution cycles expended in executing the SPEC CINT2000 benchmark codes as compiled with **O-NS**, **I-NS**, **I-CS**, and **S-CS** configurations of IMPACT. The total height of each bar is the execution time of the application as compiled, relative to the execution time of the O-NS configuration. For example, *gzip* executes in 22% less time when compiled with the I-CS configuration. Each bar segment represents the portion of execution time attributable to one of nine categories, as determined through use of the performance monitoring layer in the Linux kernel and Pfmon software [63].

Table A.1, in the previous appendix, indicates the performance monitoring measurements comprising the various categories shown in the figure [51]. Before delving into performance results for individual benchmarks, it is instructive to understand these accounting categories. The categories (and underlying hardware counters) are set up in such a way that they are mutually exclusive, so the counters taken collectively are an accurate summary of the state of the back end throughout execution. One caveat is appropriate here: instruction scheduling can change the allocation of events (i.e., scheduling instructions into what would otherwise be stall cycles moves these cycles into "unstalled," even

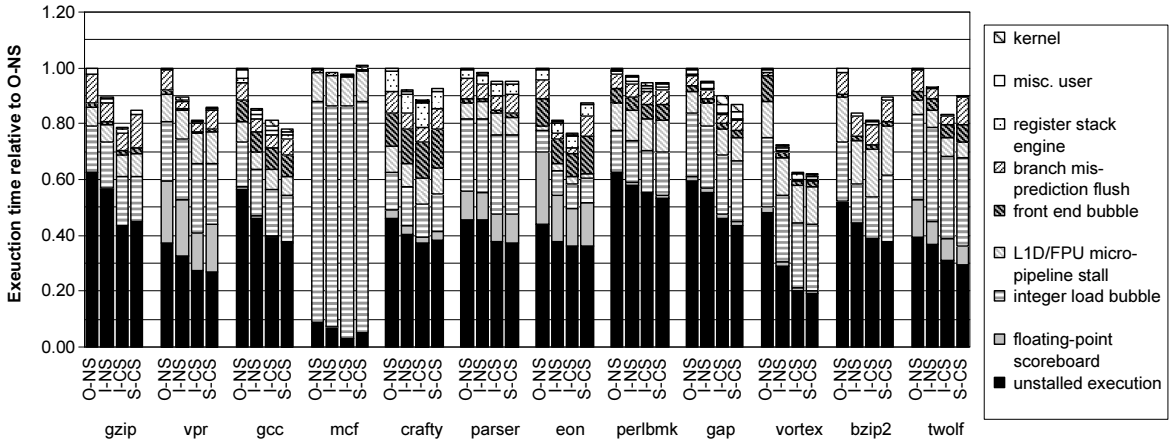


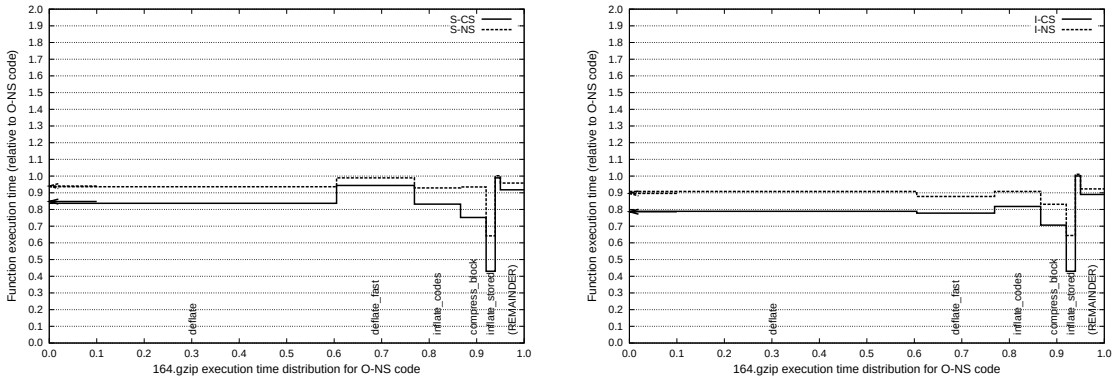
Figure C.1 Cycle accounting results: IMPACT **O-NS**, **I-NS**, **I-CS**, and **S-CS** configurations.

though net ILP is note increased). Since the compiler generally tries to generate schedules that are as compact as possible and does not generally attempt to schedule loads and other variable-latency operations at their miss latency, this is likely not a significant factor in interpreting these results, as long as they are taken at face value.

In Figure C.1, various general effects of ILP optimization can be observed. In general, code compiled for ILP using IMPACT shows a decrease in two categories: "Unstalled back-end cycles," cycles in which the machine retired one or more program operations, and "Branch misprediction flush," cycles in which the machine was recovering from a branch misprediction. This reflects successful ILP optimization, which attempts to mitigate the effects of control flow on execution and to compress more useful instructions into each cycle. This figure will be referred to occasionally in the following benchmark-specific results.

C.3 Analysis of Individual Benchmarks

Hereafter follow the program-specific analyses. Each program is characterized with respect to the basic algorithms it implements, thanks to the descriptions provided in [5], and with respect to its amenability to the ILP optimizations described in this dissertation. As a general indicator of the benefit of ILP optimization, two graphs such as those of Figure C.2 are provided for each component program. A brief word of explanation is



(a) (b)

Figure C.2 Execution profile for *164.gzip*. Execution times are for code with (a) nonpredicated and (b) predicated CFS transformation applied, relative to classically optimized code.

in order regarding the reading of these graphs. Figure C.2(a) compares the execution time of code optimized for ILP but without the use of explicit control speculation to the execution time of code optimized using only classical techniques. Each segment of the graph corresponds to one function (after function inlining). The proportion of the horizontal axis allotted to a function indicates its contribution to the total execution time of the program in the classically optimized version. The height of the segment for a function indicates its relative execution time in the code optimized for ILP. A segment below 1.0 on the vertical axis indicates a speedup for ILP techniques. The total relative execution time of the ILP-optimized application is indicated by the placement of the arrow on the left vertical axis. Figure C.2(b) shows similar results for ILP-optimized code using explicit control speculation. For example, the function `deflate()` accounts for 59% of execution time when compiled in the O-NS configuration and runs in 0.79× this amount of time in the I-CS configuration. As the arrow on the vertical axis indicates, this is also approximately the average execution time proportion for the benchmark.

C.3.1 *164.gzip*

Gzip is an implementation of the well-known Lempel-Ziv (LZ77) compression algorithm. As such, it has loop regions exhibiting a high degree of locality but containing nontrivial control flow. A prominent example was shown in Figure 3.1.

Table C.2 Cycle accounting data for *164.gzip*

Accounting category	**O-NS**	**S-NS**	**S-CS**	**I-NS**	**I-CS**
unstalled	1.492E+11	1.316E+11	1.069E+11	1.346E+11	1.033E+11
flush br	2.414E+10	2.564E+10	2.810E+10	1.547E+10	1.524E+10
flush xpn	2.441E+06	2.323E+06	2.153E+06	2.246E+06	2.031E+06
micropipe	1.615E+10	1.608E+10	1.891E+10	1.442E+10	1.854E+10
scorebd gr/gr	4.298E+09	4.684E+09	3.443E+09	4.183E+09	2.968E+09
scorebd gr/ld	3.849E+10	4.003E+10	3.804E+10	4.036E+10	4.211E+10
scorebd fr	1.382E+08	1.427E+08	4.621E+07	7.976E+07	4.414E+07
scorebd misc	2.961E+07	2.947E+07	2.958E+07	2.916E+07	2.944E+07
reg stack	5.927E+06	5.877E+06	6.742E+06	7.005E+06	6.788E+06
front end	4.375E+09	4.768E+09	5.619E+09	2.627E+09	3.249E+09
KERNEL	9.908E+08	1.126E+09	1.018E+09	9.893E+08	9.629E+08
TOTAL	2.378E+11	2.241E+11	2.021E+11	2.128E+11	1.865E+11

Figure C.2 shows the execution time for the program's functions, relative to the baseline (**O-NS**) version, in nonpredicated (**S-NS** and **S-CS**) and predicated (**I-NS** and **I-CS**) configurations. Tables C.2 and C.3 indicate cycle accounting and instruction counting data, respectively. IMPACT achieves a significant benefit by successfully using Superblock and Hyperblock formation, predication, and control speculation to transform key `while` loops for profitable modulo scheduling, handily outstripping production compiler results (with a SPEC score of 751 vs. 629 for `icc`). The most important loop (`deflate.c:397`) is optimized in the form of three versions, as selected based on control flow graph profile information. All versions are trace-specialization based; one version is developed using a peel of an inner loop. Only one (the outermost one, with the peel) includes multiple paths. Peeling buys about a 2% performance gain. (This example was presented in Section 3.3.1.) All these transformations are highly profile-dependent, but the application profile seems relatively stable.

I-CS achieves a similar reduction in unstalled execution cycles to that achieved by **S-CS**, but with concomitant reductions in branch misprediction flush and front-end bubble cycles. (A well-tuned **S-CS** approach sees increases in both these categories, fully accounting for the benefit of **I-CS**.) IMPACT's aggressive use of predication eliminates a quarter of branch misprediction stall cycles, relative to the **O-NS** version. Relative to a

Table C.3 Instruction accounting data for *164.gzip*

	O-NS	**S-NS**	**S-CS**	**I-NS**	**I-CS**
Useful instr.	3.632E+11	3.432E+11	3.577E+11	3.503E+11	3.721E+11
Pred. squashed instr.	3.468E+09	6.134E+09	5.233E+09	3.374E+10	2.310E+10
`nop` instr.	1.162E+11	7.849E+10	6.469E+10	6.622E+10	5.401E+10
Total instr.	4.829E+11	4.278E+11	4.276E+11	4.503E+11	4.492E+11

Superblock form, the Hyperblock version has only three-fifths as many branch misprediction stall cycles and one half the number of front end stall cycles. Predication-enhanced region formation showed substantial benefit in *gzip*, achieving a 1.09× speedup relative to **S-CS**, and **I-CS** benefited every function with significant execution time, compared to **S-CS**. Most applications of predication, however, tend to be the inclusion of simple hammock or diamond structures that may enable the modulo-scheduling of (or increase the coverage of) particular, important loop kernels. As shown in the example of 3.3.1, more aggressive use of predication could further increase the coverage of key kernel loops without measurably increasing their execution time. This yields an additional 2% benchmark speedup over **I-CS**, not reflected in the results given here. Given the current means of controlling Hyperblock aggressiveness, however, this level of if-conversion reduces the average performance of the suite.

Control speculation increases the number of cycles spent resolving data address translation by 43%, consuming a total of 5.6% of execution time in the **I-CS** configuration.

Interprocedural pointer analysis, including Omega test, yielded a 1.11× speedup in the **I-CS** configuration. Procedure inlining also increases performance by 1.11×, disproportionately benefiting CFS-optimized code (note that the peeling-based outer loop specialization example of 3.3.1 required inlining).

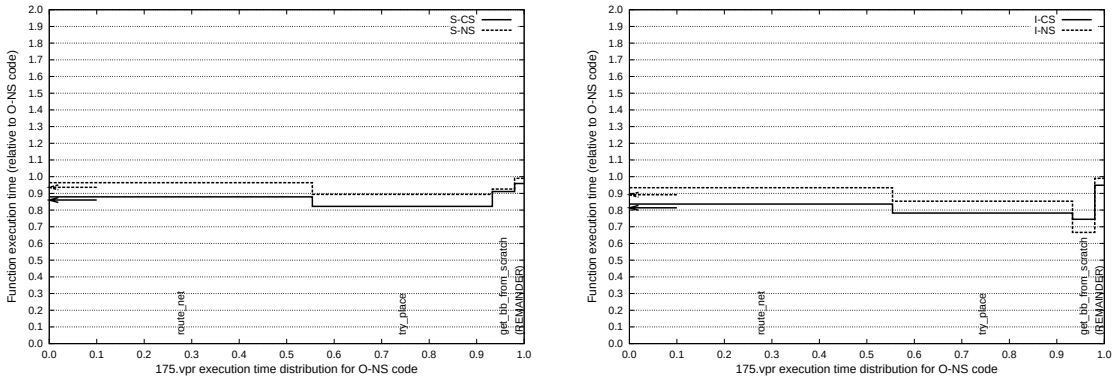
Modulo scheduling achieves a 9% benefit (in the **I-CS** configuration) relative to compilation without modulo scheduling in a mildly unrolled version of the code and a 7% benefit relative to a relatively heavily unrolled version.

C.3.2 *175.vpr*

Vpr consists of two phases, "place" and "route," of the procedure for mapping logic circuits onto field-programmable gate arrays (FPGA). Execution time is spent largely in maintaining a heap organized using floating-point keys.

Figure C.3 shows the execution time for the program's functions, relative to the baseline (**O-NS**) version, in nonpredicated (**S-NS** and **S-CS**) and predicated (**I-NS** and **I-CS**) configurations. Tables C.4 and C.5 indicate cycle accounting and instruction counting data, respectively. **I-CS** achieves a 23% improvement over **O-NS**. Predication-enhanced region formation (**I-CS**) shows substantial benefit in *vpr*, achieving a 1.06× speedup relative to the the **S-CS** configuration, and this improvement was relatively immune to reasonable changes in if-conversion heuristic parameters. This difference derives from `route_net()`, with a 10% reduction in execution time, and `get_bb_from_scratch()`, with a 40% reduction; **S-CS** is roughly as effective as **I-CS** in the important `try_place()`, in which control structures lend themselves more to relatively short, but stable, individual traces. The success in `get_bb_from_scratch()` is due to the example shown in Figure 5.1. The vast majority of the benefit of **I-CS** relative to **S-CS** is due to reduction in branch misprediction flush.

The function `get_bb_from_scratch()` contains an interesting loop, as used in the example of Figure 5.1. The use of predication in this loop (**I-CS**) achieves a speedup of 1.30 relative to a Superblock-only configuration (**S-CS**). This speedup would be higher (1.58) except for the fact that control speculation and the software pipelining it permits allow two unrelated loads to move into the same cycle in the schedule. These loads are frequently serviced in the L2 data cache and access the same cache bank. When this happens, one of the two fails to make the L2 bypass and recirculates for four cycles, penalizing loop performance by nearly 20%. This loop dominates `get_bb_from_scratch()`, and this loss of performance between **I-NS** and **I-CS** is clearly visible in Figure C.3, but this loss has nothing whatsoever to do with control speculation! (The increase in load-related scoreboard cycles is also clearly evident in Table C.4.) This effect was observed in



(a) (b)

Figure C.3 Execution profile for *175.vpr*. Execution times are for code with (a) nonpredicated and (b) predicated CFS transformation applied, relative to classically optimized code.

Table C.4 Cycle accounting data for *175.vpr*

Accounting category	**O-NS**	**S-NS**	**S-CS**	**I-NS**	**I-CS**
unstalled	8.441E+10	6.898E+10	6.047E+10	7.447E+10	6.272E+10
flush br	1.607E+10	1.541E+10	1.513E+10	6.930E+09	6.829E+09
flush xpn	9.028E+06	9.234E+06	1.257E+07	8.136E+06	1.719E+07
micropipe	2.255E+10	2.207E+10	2.541E+10	2.314E+10	2.478E+10
scorebd gr/gr	1.334E+06	1.350E+06	1.340E+06	1.355E+06	1.363E+06
scorebd gr/ld	4.840E+10	4.971E+10	5.035E+10	4.917E+10	5.643E+10
scorebd fr	5.008E+10	5.119E+10	3.831E+10	4.550E+10	2.995E+10
scorebd misc	3.487E+07	1.526E+08	1.240E+08	1.067E+08	1.684E+08
reg stack	7.465E+08	7.541E+08	7.531E+08	1.146E+09	1.227E+09
front end	3.226E+09	2.112E+09	2.227E+09	7.803E+08	6.512E+08
KERNEL	1.356E+09	1.352E+09	1.646E+09	1.433E+09	1.759E+09
TOTAL	2.269E+11	2.117E+11	1.944E+11	2.027E+11	1.845E+11

Table C.5 Instruction accounting data for *175.vpr*

	O-NS	**S-NS**	**S-CS**	**I-NS**	**I-CS**
Useful instr.	1.795E+11	1.721E+11	1.790E+11	1.755E+11	1.866E+11
Pred. squashed instr.	6.776E+08	5.653E+08	6.447E+08	2.117E+10	1.761E+10
`nop` instr.	7.235E+10	5.728E+10	4.774E+10	5.561E+10	4.437E+10
Total instr.	2.525E+11	2.299E+11	2.274E+11	2.523E+11	2.486E+11

the data and confirmed with hand-rescheduling of the affected loop to remove the bank conflict.

One of *vpr*'s key loops, a heap extraction routine (in `try_place()`), exhibits an opportunity for redundant load elimination that is only accessible to partial redundancy elimination (PRE) or to path-sensitive optimization. IMPACT applies PRE here, removing a redundant floating-point load (floating-point loads take at least six cycles on Itanium 2) from this key loop, achieving approximately a 3% benefit to overall execution time. This loop also exhibits additional opportunities for node splitting and slicing transformations and advance scheduling of loads to predictable addresses, but exploitation of these opportunities will require careful manipulation of loop-carried control dependence.

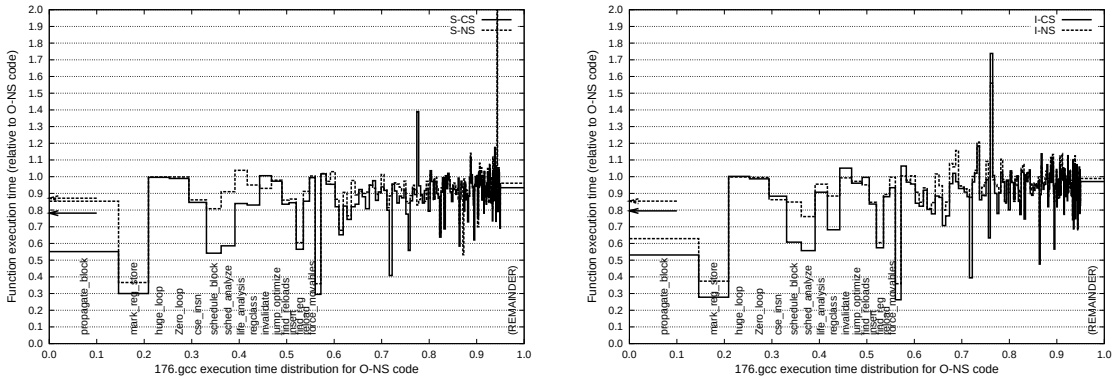
Interprocedural pointer analysis delivers a 1.28× speedup, but only when a field-sensitive, context-sensitive, heap-cloning approach is used. This reflects the most complex use of pointer analysis in the benchmark suite. Procedure inlining is responsible for an impressive 1.32× increase in performance. *Vpr* garners a 1% gain from `Lsuperscalar` optimizations (aside from region formation and unrolling).

Seventeen percent of execution time in the **I-CS** configuration is spent resolving data address translations, but this represents less than a 10% increase in such cycles due to control speculation. L2 bank conflicts increase with control speculation, especially with predication (doubling), but at their peak still account for only approximately 2% of execution time.

C.3.3 *176.gcc*

Gcc is the GNU C Compiler. Much of its execution time is spent in bit-vector data flow routines and instruction scheduling, both of which provide excellent opportunities for instruction-level parallelism.

Figure C.4 shows the execution time for the program's functions, relative to the baseline (**O-NS**) version, in nonpredicated (**S-NS** and **S-CS**) and predicated (**I-NS** and **I-CS**) configurations. Tables C.6 and C.7 indicate cycle accounting and instruction counting data, respectively.



(a) (b)

Figure C.4 Execution profile for *176.gcc*. Execution times are for code with (a) nonpredicated and (b) predicated CFS transformation applied, relative to classically optimized code.

Table C.6 Cycle accounting data for *176.gcc*

Accounting category	**O-NS**	**S-NS**	**S-CS**	**I-NS**	**I-CS**
unstalled	7.444E+10	6.061E+10	4.915E+10	6.107E+10	5.188E+10
flush br	8.358E+09	7.973E+09	7.588E+09	5.762E+09	5.906E+09
flush xpn	2.631E+06	2.508E+06	3.411E+06	2.458E+06	4.629E+07
micropipe	9.234E+09	9.184E+09	8.785E+09	8.656E+09	9.552E+09
scorebd gr/gr	3.701E+09	1.146E+09	1.149E+09	1.174E+09	8.019E+08
scorebd gr/ld	2.146E+10	2.160E+10	2.190E+10	2.178E+10	2.206E+10
scorebd fr	1.089E+09	1.110E+09	8.620E+08	1.002E+09	7.211E+08
scorebd misc	5.240E+08	6.170E+08	6.464E+08	8.361E+08	7.698E+08
reg stack	1.653E+09	1.613E+09	1.717E+09	2.121E+09	2.524E+09
front end	1.060E+10	9.985E+09	1.043E+10	9.399E+09	9.947E+09
KERNEL	9.505E+08	8.995E+08	9.781E+08	8.779E+08	2.890E+09
TOTAL	1.320E+11	1.147E+11	1.032E+11	1.127E+11	1.071E+11

Table C.7 Instruction accounting data for *176.gcc*

	O-NS	**S-NS**	**S-CS**	**I-NS**	**I-CS**
Useful instr.	1.786E+11	1.652E+11	1.663E+11	1.639E+11	1.656E+11
Pred. squashed instr.	7.859E+08	1.187E+09	1.176E+09	1.314E+10	1.230E+10
`nop` instr.	4.582E+10	3.235E+10	2.804E+10	2.888E+10	2.430E+10
Total instr.	2.252E+11	1.988E+11	1.955E+11	2.059E+11	2.022E+11

Gcc benefits substantially from IMPACT's Super- and Hyperblock loop transformations and modulo scheduling, cross-file inlining, load/store optimizations, and specializations such as loop peeling. Data flow routines, providing a great deal of available parallelism, are optimized with great success. Examining the cycle accounting data, substantial reductions in branch misprediction penalty and unstalled execution cycles (reflecting increases in both "dynamic" and "static" ILP formation) are evident. A Superblock-only region formation strategy is approximately equally effective as the Hyperblock-enabled region former in *gcc*.

As indicated by the nearly 10% of execution cycles spent in front-end bubbles, *gcc* has some nontrivial instruction cache behavior. At the same time, code-expanding ILP transformations are achieving substantial benefits. *Gcc* should therefore be one of the most interesting benchmarks in which to observe the effects of more systematic transformation techniques.

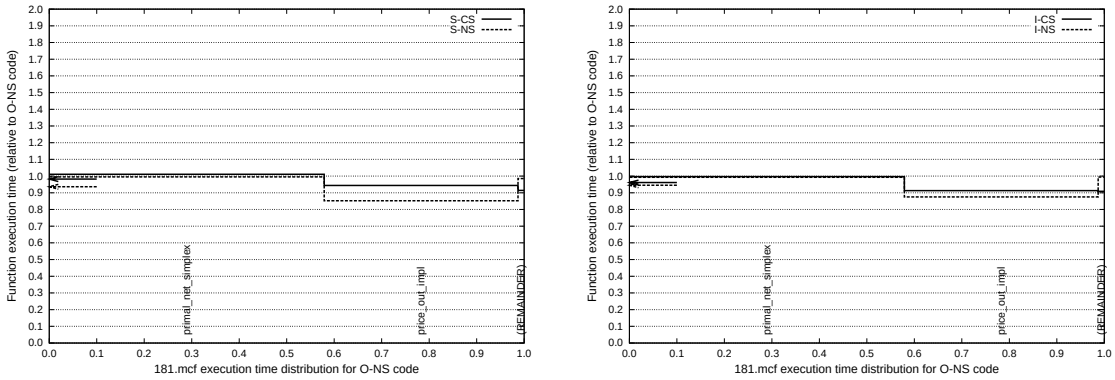
Due to its heavy use of "C polymorphism" in union-based data structures (chiefly `rtx`), *gcc* exhibited a very high rate of wild loads (20% of execution time was spent in the kernel handling spurious page faults) before the measures described in Section 6.7 were put in place to avoid these dangerous speculations.

Interprocedural pointer analysis, including Omega test, yielded a 1.11× speedup. Procedure inlining delivered a speedup of 1.11. Modulo scheduling achieves a 13% performance improvement in *gcc*, capitalizing on the compiler's many data flow loops. This is the best showing for modulo scheduling in these experiments with SPEC CINT2000. In Figure C.4, `huge_loop()` and `Zero_loop()`, which show no improvement, are library code.

C.3.4 *181.mcf*

Mcf is a bus (metropolitan transit route, not electronic interface) scheduling application. It involves the iterative refinement of a large linked data structure. The topology of the structure is updated throughout program execution.

Figure C.5 shows the execution time for the program's functions, relative to the baseline (**O-NS**) version, in nonpredicated (**S-NS** and **S-CS**) and predicated (**I-NS**



(a) (b)

Figure C.5 Execution profile for *181.mcf*. Execution times are for code with (a) nonpredicated and (b) predicated CFS transformation applied, relative to classically optimized code.

and **I-CS**) configurations. Tables C.8 and C.9 indicate cycle accounting and instruction counting data, respectively.

The linked data structure forming the core of this benchmark suffers from devastatingly bad data cacheability—95% of benchmark execution cycles are spent waiting for memory operations. As a linked data structure it is not susceptible to standard compiler-based strided prefetching strategies. A stride-detecting software prefetching strategy described by Wu, used in the Intel compiler, approximately doubles *mcf*'s performance [54]. The technique instruments the code to perform a run-time-determined strided prefetch. Periodic run-time re-evaluations allow the stride to be adjusted or the prefetch to be canceled if judged ineffective. Since these techniques are available today in the commercial environment, and since even with these techniques the contribution of CFS transformation to *mcf*'s performance would be limited, this technique was not pursued in the IMPACT work.

Due to the dominance of memory-related stalls in the performance of *mcf*, procedure inlining, region formation, predication, and speculation have only barely measurable effects on the benchmark's run time. Data address translation overhead accounts for approximately 10% of benchmark execution time.

Table C.8 Cycle accounting data for *181.mcf*

Accounting category	**O-NS**	**S-NS**	**S-CS**	**I-NS**	**I-CS**
unstalled	4.739E+10	4.368E+10	2.727E+10	4.015E+10	1.839E+10
flush br	5.586E+09	6.231E+09	6.296E+09	3.344E+09	3.558E+09
flush xpn	5.600E+07	5.308E+07	5.421E+07	5.506E+07	5.392E+07
micropipe	5.713E+10	5.673E+10	6.027E+10	5.919E+10	5.722E+10
scorebd gr/gr	1.541E+03	1.518E+03	1.518E+03	1.479E+03	1.545E+03
scorebd gr/ld	4.394E+11	4.561E+11	4.596E+11	4.391E+11	4.589E+11
scorebd fr	8.845E+07	6.706E+07	6.697E+07	6.963E+07	6.707E+07
scorebd misc	4.129E+07	3.474E+06	4.168E+07	2.810E+06	4.130E+07
reg stack	4.665E+07	5.696E+07	1.129E+08	5.983E+07	6.165E+07
front end	1.236E+09	9.359E+08	1.240E+09	7.958E+08	7.017E+08
KERNEL	2.501E+09	2.120E+09	2.705E+09	2.496E+09	2.480E+09
TOTAL	5.535E+11	5.660E+11	5.577E+11	5.452E+11	5.414E+11

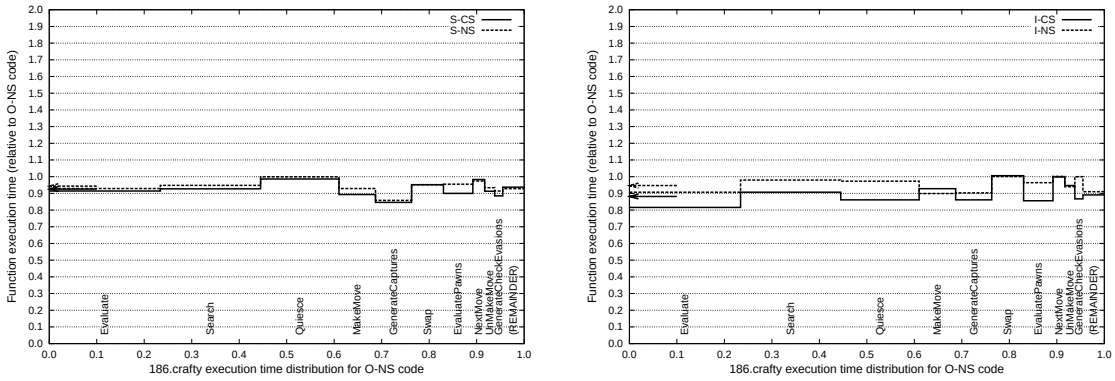
Table C.9 Instruction accounting data for *181.mcf*

	O-NS	**S-NS**	**S-CS**	**I-NS**	**I-CS**
Useful instr.	7.140E+10	6.908E+10	7.336E+10	6.600E+10	7.005E+10
Pred. squashed instr.	8.643E+07	2.036E+09	7.595E+08	4.616E+09	1.915E+09
`nop` instr.	2.739E+10	2.183E+10	1.914E+10	2.294E+10	1.679E+10
Total instr.	9.887E+10	9.295E+10	9.327E+10	9.355E+10	8.876E+10

C.3.5 *186.crafty*

Crafty is a chess game configured to play against itself. The position of the pieces on the board is maintained in many bit vectors. The algorithm searches a given number of moves ahead and scores possible outcomes.

Figure C.6 shows the execution time for the program's functions, relative to the baseline (**O-NS**) version, in nonpredicated (**S-NS** and **S-CS**) and predicated (**I-NS** and **I-CS**) configurations. Tables C.10 and C.11 indicate cycle accounting and instruction counting data, respectively. *Crafty* proves a problematic benchmark for the IMPACT compiler for several reasons. IMPACT's performance is competitive, although IMPACT suffers from an increase in instruction cache footprint due to excessive specialization. Any reduction in active code size will result in performance improvement. IMPACT is very successful at forming parallel execution regions in *crafty*, and one cost of this success



(a) (b)

Figure C.6 Execution profile for *186.crafty*. Execution times are for code with (a) non-predicated and (b) predicated CFS transformation applied, relative to classically optimized code.

Table C.10 Cycle accounting data for *186.crafty*

Accounting category	**O-NS**	**S-NS**	**S-CS**	**I-NS**	**I-CS**
unstalled	6.988E+10	6.273E+10	5.865E+10	6.147E+10	5.643E+10
flush br	1.187E+10	1.074E+10	1.080E+10	8.078E+09	7.751E+09
flush xpn	1.511E+06	1.470E+06	1.493E+06	1.493E+06	1.448E+06
micropipe	1.357E+10	1.353E+10	1.415E+10	1.222E+10	1.357E+10
scorebd gr/gr	1.137E+09	9.236E+08	1.077E+09	1.257E+09	9.831E+08
scorebd gr/ld	2.064E+10	2.104E+10	2.030E+10	2.126E+10	1.810E+10
scorebd fr	5.055E+09	4.618E+09	4.249E+09	4.623E+09	3.345E+09
scorebd misc	4.916E+08	2.690E+07	1.659E+08	1.387E+08	1.654E+08
reg stack	1.048E+10	8.261E+09	9.385E+09	1.480E+10	1.366E+10
front end	1.862E+10	2.155E+10	2.180E+10	1.943E+10	1.957E+10
KERNEL	3.738E+08	4.330E+08	3.566E+08	2.967E+08	3.139E+08
TOTAL	1.521E+11	1.439E+11	1.409E+11	1.436E+11	1.339E+11

Table C.11 Instruction accounting data for *186.crafty*

	O-NS	**S-NS**	**S-CS**	**I-NS**	**I-CS**
Useful instr.	1.841E+11	1.794E+11	1.833E+11	1.858E+11	1.880E+11
Pred. squashed instr.	7.182E+08	5.156E+08	6.509E+08	1.720E+10	1.582E+10
`nop` instr.	4.656E+10	3.913E+10	3.527E+10	3.194E+10	2.858E+10
Total instr.	2.314E+11	2.191E+11	2.193E+11	2.350E+11	2.324E+11

is high register stack utilization in many functions. Over 8% of cycles, in fact, are spent simply manipulating the register stack. Performance would benefit from more strategic inlining or from an interprocedural register allocation approach. Major recursive functions (e.g., `Search()` and `Quiesce()`) require inlining of off-recursion callees (e.g., `Evaluate()`) to achieve successful levels of ILP, but this inlining increases the number of registers used in the recursive invocation, noticeably impacting register stack engine activity. The procedure granularity does not provide useful boundaries for control of inlining. Likewise, management of instruction cache resources is poor. Better management of instruction footprint and register stack during code specialization would substantially improve results.

Current peeling heuristics gain only approximately 1%, although particular procedures benefit up to 6%. The net loss may be due to simple code expansion. Profiling with reference, instead of training, input yields about a 5% improvement, revealing that profile mismatch is a substantial factor in the benchmark's performance.

Crafty is a highly structured application, but not one written with inlining and ILP development in mind. In the development of this dissertation, it was thought that the compiler could make very long-range and very wise decisions about combining similar execution regions, allowing most of the critical ILP transformations to be performed while actually *decreasing* the application's instruction cache footprint. Crafty indeed contains some very profound examples of these situations, but hand-manipulated versions of *crafty* designed to test the efficacy of these transformations resulted almost uniformly in degraded performance. The reason for these degradations appears to be a loss of specialization opportunities in the "compacted" versions of the code. *Crafty*'s execution reflects significant and highly input-dependent biases, so this is not entirely unanticipated. It is, however, unfortunate, as the author had hoped *crafty* would be a useful, motivational example for more highly ordered ILP transformations.

Additionally, *crafty* offers much coarser-grained parallelism that is today going unexploited. More sophisticated formation techniques and more efficient mechanisms for exploiting fine-grained parallelism could render this accessible, for even greater gains.

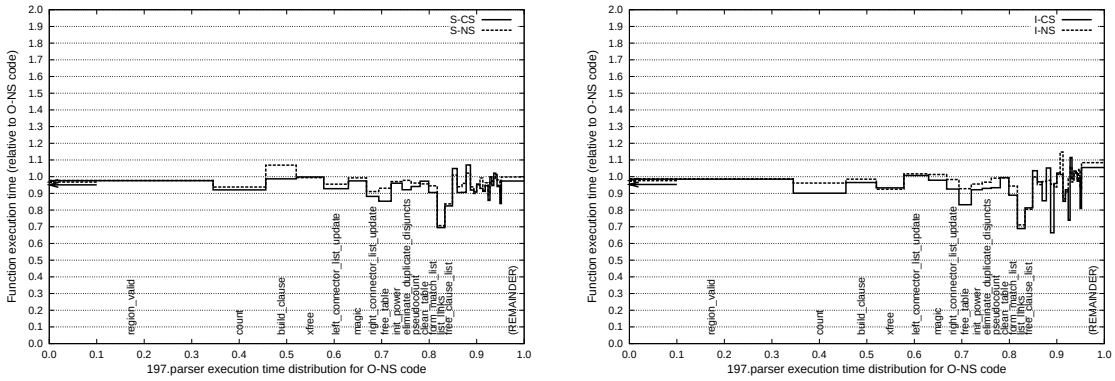
The performance of the speculative versions of *crafty* suffers from a marked increase in data translation lookaside buffer misses. This can be traced to the use of large tables in (often inlined) invocations of the functions `FirstOne()` and `LastOne()`. In these functions, the 64-bit Boolean vector representing the chessboard is broken into four 16-bit segments, which are then used in sequence to access a $2^{16} = 65536$ byte table to identify the first or last bit set to 1 in the vector. In the predicated code, with control speculation, all four of these loads are executed in a given traversal of each of these segments.

Interprocedural pointer analysis, including Omega test, yielded no tangible benefit.

C.3.6 *197.parser*

Parser is an English syntactic parser based on link grammars. It applies a dictionary and syntactic rules to input sentences, constructing a syntax graph for the input word sequence and determining whether or not it is grammatical. It is highly recursive and depends on a dynamic programming approach with a relatively expensive hash scheme at its core.

Figure C.7 shows the execution time for the program's functions, relative to the baseline (**O-NS**) version, in nonpredicated (**S-NS** and **S-CS**) and predicated (**I-NS** and **I-CS**) configurations. Tables C.12 and C.13 indicate cycle accounting and instruction counting data, respectively. This application benefits only minimally (achieving a speedup of only 1.06 relative to classically optimized code) from CFS transformation in the IMPACT compiler. *Parser* poses several difficulties for an EPIC compiler. It is highly recursive, creating challenges for the inliner. It makes extensive use of a complex, serial hashing function involving the repeated computation of moduli. IMPACT is more successful than Intel's production compiler at hiding its latency, but more could be done. Superblock-only region formation (**S-CS**) is slightly (1.03×) more effective in producing performance than the Hyperblock system (**I-CS**), as configured, suggesting that unprofitable Hyperblocks are being formed. More aggressive Hyperblock selection degraded performance slightly.



(a) (b)

Figure C.7 Execution profile for *197.parser*. Execution times are for code with (a) non-predicated and (b) predicated CFS transformation applied, relative to classically optimized code.

Table C.12 Cycle accounting data for *197.parser*

Accounting category	**O-NS**	**S-NS**	**S-CS**	**I-NS**	**I-CS**
unstalled	1.543E+11	1.432E+11	1.269E+11	1.537E+11	1.277E+11
flush br	2.377E+10	2.278E+10	2.280E+10	1.828E+10	1.860E+10
flush xpn	1.031E+07	9.872E+06	1.142E+07	9.823E+06	6.166E+07
micropipe	1.924E+10	1.892E+10	2.087E+10	1.988E+10	2.504E+10
scorebd gr/gr	4.809E+07	4.482E+07	4.263E+07	4.790E+07	4.851E+07
scorebd gr/ld	8.848E+10	8.954E+10	9.677E+10	8.975E+10	9.603E+10
scorebd fr	3.500E+10	3.435E+10	3.437E+10	3.423E+10	3.427E+10
scorebd misc	2.011E+09	2.420E+09	2.554E+09	2.434E+09	2.652E+09
reg stack	9.613E+09	1.113E+10	1.323E+10	9.352E+09	1.354E+10
front end	5.288E+09	4.027E+09	4.096E+09	3.476E+09	3.913E+09
KERNEL	1.092E+09	1.021E+09	1.063E+09	1.077E+09	1.395E+09
TOTAL	3.388E+11	3.275E+11	3.227E+11	3.323E+11	3.233E+11

Table C.13 Instruction accounting data for *197.parser*

	O-NS	**S-NS**	**S-CS**	**I-NS**	**I-CS**
Useful instr.	3.291E+11	3.203E+11	3.295E+11	3.131E+11	3.302E+11
Pred. squashed instr.	1.172E+09	9.476E+08	9.147E+08	2.446E+10	1.612E+10
`nop` instr.	1.272E+11	1.072E+11	9.912E+10	1.077E+11	9.555E+10
Total instr.	4.574E+11	4.284E+11	4.296E+11	4.453E+11	4.419E+11

Interprocedural pointer analysis yielded a 1.05× speedup in the **I-CS** configuration. Intel's reference compiler improves *parser*'s performance by approximately 10% using a patented scheme for stride-detecting software prefetching [54]. IMPACT lacks this capability and so trails Intel's performance.

The hashing algorithm could be subjected to very sophisticated global code motion (global and interprocedural common subexpression elimination). This benchmark has complex control flow and difficult-to-expose redundancy that may make it an interesting (and challenging) test case for new ILP techniques.

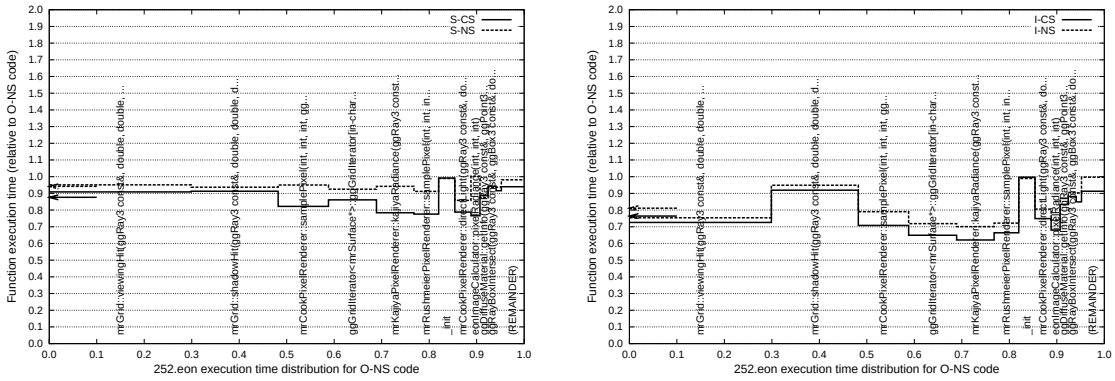
C.3.7 *252.eon*

Eon is a probabilistic ray tracer written in C++.

Figure C.8 shows the execution time for the program's functions, relative to the baseline (**O-NS**) version, in nonpredicated (**S-NS** and **S-CS**) and predicated (**I-NS** and **I-CS**) configurations. Tables C.14 and C.15 indicate cycle accounting and instruction counting data, respectively. The **I-CS** configuration delivers approximately a 1.10× speedup relative to **S-CS**. Procedure inlining, especially that of indirect function calls, was very important to *eon*'s performance, as its object-oriented nature would suggest. A call graph profile reveals that most indirect function call invocations have only one or a handful of common targets, making such transformation relatively easy and highly effective. Procedure inlining delivered a speedup of 1.40.

Eon frequently uses structure assignments, which are generally implemented as sets of double-word loads and stores. These structures frequently contain floating-point values. This creates two problems: first, memory copy propagation opportunities are missed. Second, values read and written as floating-point operations wind up in the first level cache, potentially causing penalties. These deficiencies, at a minimum, need to be fixed to make IMPACT's *eon* performance more competitive.

Eon performance benefits 2% from modulo scheduling and 3% from `Lsuperscalar` optimizations (beyond region formation and unrolling). In Figure C.8, `_init()` is C++ library code.



(a) (b)

Figure C.8 Execution profile for *252.eon*. Execution times are for code with (a) nonpredicated and (b) predicated CFS transformation applied, relative to classically optimized code.

Table C.14 Cycle accounting data for *252.eon*

Accounting category	**O-NS**	**S-NS**	**S-CS**	**I-NS**	**I-CS**
unstalled	1.225E+11	1.053E+11	1.011E+11	1.055E+11	1.011E+11
flush br	1.914E+10	1.977E+10	2.004E+10	6.137E+09	5.732E+09
flush xpn	2.379E+06	2.265E+06	2.110E+06	1.959E+06	1.856E+06
micropipe	4.164E+09	5.579E+09	5.567E+09	7.084E+09	6.972E+09
scorebd gr/gr	2.572E+06	2.541E+06	2.542E+06	2.609E+06	2.570E+06
scorebd gr/ld	2.191E+10	2.390E+10	2.632E+10	2.508E+10	2.573E+10
scorebd fr	7.353E+10	6.003E+10	4.273E+10	4.595E+10	3.746E+10
scorebd misc	1.192E+09	1.060E+09	1.355E+09	1.179E+09	1.835E+09
reg stack	1.036E+10	1.074E+10	1.127E+10	1.117E+10	1.129E+10
front end	2.674E+10	3.668E+10	3.724E+10	2.428E+10	2.348E+10
KERNEL	4.882E+08	5.455E+08	5.354E+08	4.297E+08	3.846E+08
TOTAL	2.800E+11	2.636E+11	2.462E+11	2.268E+11	2.140E+11

Table C.15 Instruction accounting data for *252.eon*

	O-NS	**S-NS**	**S-CS**	**I-NS**	**I-CS**
Useful instr.	3.372E+11	3.182E+11	3.259E+11	3.201E+11	3.232E+11
Pred. squashed instr.	1.697E+09	1.719E+09	1.719E+09	1.753E+10	1.736E+10
`nop` instr.	1.316E+11	1.102E+11	1.012E+11	1.013E+11	9.159E+10
Total instr.	4.704E+11	4.301E+11	4.289E+11	4.389E+11	4.321E+11

C.3.8 *253.perlbmk*

Perlbmk is a benchmark-ized version of the Perl interpreter. Its behavior can vary widely depending on the input script, so it poses challenges for profile-based optimizers. A run of *perlbmk* with its reference input used for training yields a 1.18× speedup, indicating that profile stability is a problem in this benchmark, as is common with interpretive-style applications.

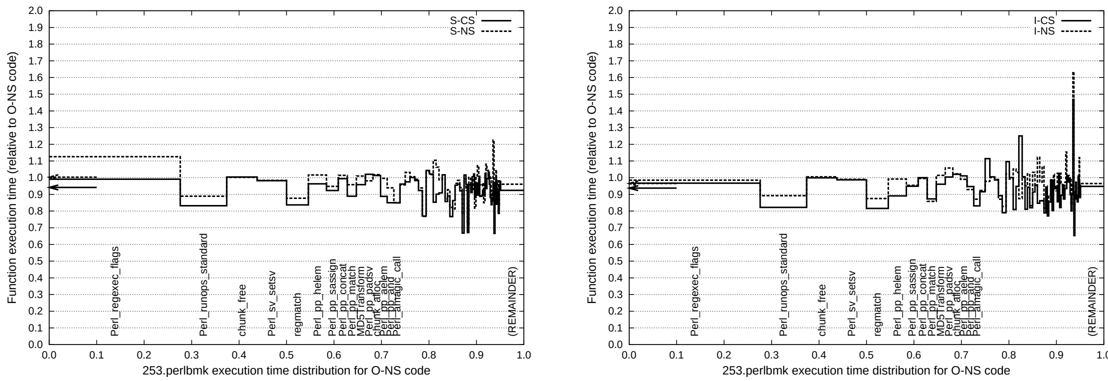
Figure C.9 shows the execution time for the program's functions, relative to the baseline (**O-NS**) version, in nonpredicated (**S-NS** and **S-CS**) and predicated (**I-NS** and **I-CS**) configurations. Tables C.16 and C.17 indicate cycle accounting and instruction counting data, respectively. This application benefits only very slightly (achieving a speedup of only 1.02 relative to classically optimized code) from CFS transformation in the IMPACT compiler, as configured in this dissertation. With slightly more aggressive predication, however, including paths of lower weight relative to the main path and allowing a greater increase in operations issued on included alternate paths, its benefit increases, yielding a speedup of 1.10.

In *perlbmk*, indirect function call inlining improved performance for all versions of the code. In CFS configurations, this transformation revealed opportunities for subsequent optimization. These opportunities also became sites of frequent wild load generation, which dramatically impacted the application's performance before the techniques described in Section 6.7 were applied.

S-CS and **I-CS** configurations yield very similar results. Interprocedural pointer analysis yielded a 1.05× speedup. In Figure C.9, `chunk_free()`, `chunk_alloc()`, and `memcpy()` are library code. The execution time of `memcpy()` is evidently reduced by improvements in instruction cache behavior, since its code is not optimized by IMPACT.

C.3.9 *254.gap*

Gap is an interpreter for computational discrete algebra. The most important component from a performance production standpoint in the SPEC CINT2000 context is an interpreter component that performs multiple-precision mathematical operations. The



(a) (b)

Figure C.9 Execution profile for *253.perlbmk*. Execution times are for code with (a) non-predicated and (b) predicated CFS transformation applied, relative to classically optimized code.

Table C.16 Cycle accounting data for *253.perlbmk*

Accounting category	**O-NS**	**S-NS**	**S-CS**	**I-NS**	**I-CS**
unstalled	1.520E+11	1.476E+11	1.310E+11	1.428E+11	1.345E+11
flush br	1.320E+10	1.176E+10	1.189E+10	1.151E+10	1.125E+10
flush xpn	2.247E+07	2.290E+07	2.512E+07	2.252E+07	4.959E+07
micropipe	2.394E+10	2.686E+10	2.688E+10	2.715E+10	2.739E+10
scorebd gr/gr	2.884E+08	2.891E+08	3.134E+08	2.916E+08	3.147E+08
scorebd gr/ld	3.630E+10	3.868E+10	3.882E+10	3.668E+10	3.659E+10
scorebd fr	2.057E+09	2.031E+09	1.866E+09	1.806E+09	1.944E+09
scorebd misc	1.579E+09	2.777E+09	2.474E+09	2.780E+09	2.817E+09
reg stack	2.200E+09	2.566E+09	2.775E+09	2.504E+09	2.574E+09
front end	1.283E+10	1.374E+10	1.509E+10	1.158E+10	1.274E+10
KERNEL	1.256E+09	1.107E+09	1.190E+09	1.158E+09	1.977E+09
TOTAL	2.457E+11	2.474E+11	2.323E+11	2.383E+11	2.322E+11

Table C.17 Instruction accounting data for *253.perlbmk*

	O-NS	**S-NS**	**S-CS**	**I-NS**	**I-CS**
Useful instr.	3.991E+11	3.936E+11	4.059E+11	3.925E+11	4.078E+11
Pred. squashed instr.	1.412E+09	1.356E+09	1.357E+09	1.927E+10	2.248E+10
`nop` instr.	8.849E+10	9.040E+10	8.019E+10	8.334E+10	7.476E+10
Total instr.	4.890E+11	4.854E+11	4.875E+11	4.951E+11	5.050E+11

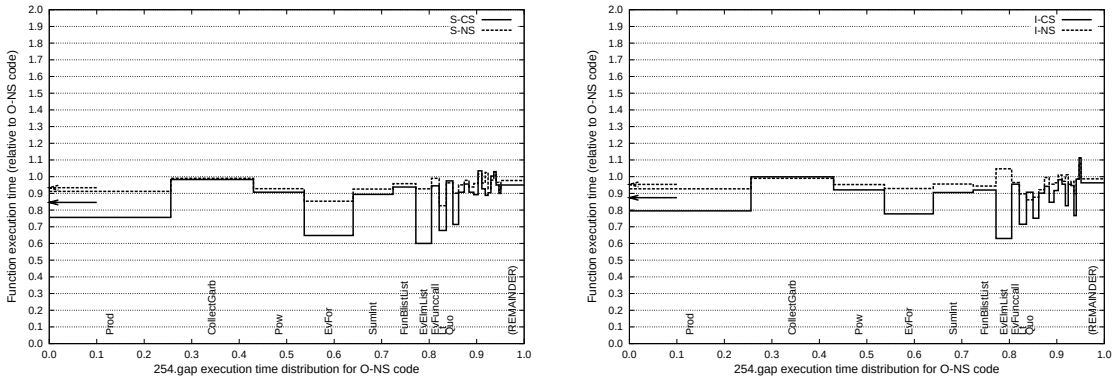
interpreter relies on a table of operator functions (an array of function pointers) to evaluate various types of expressions. Interpreted operations operate on variable-sized large numbers, represented by (containerized) array data structures allocated from a garbage-collected pool.

Figure C.10 shows the execution time for the program's functions, relative to the baseline (**O-NS**) version, in nonpredicated (**S-NS** and **S-CS**) and predicated (**I-NS** and **I-CS**) configurations. Tables C.18 and C.19 indicate cycle accounting and instruction counting data, respectively.

The interpreter structure (most invocations through function pointers) and data structures (array-based objects allocated from a custom allocator) pose special challenges for an EPIC optimizer. IMPACT exposes parallelism across indirect function invocations by performing inlining under a condition for important indirect callees (as determined in control flow graph profiling). This makes a very substantial difference (20% improvement) in *gap*'s performance, but perhaps increases profile dependence to an uncomfortable degree. This puts IMPACT's performance in the ballpark of Intel's electron compiler. Current peeling heuristics lose about 3%.

IMPACT is not successfully disambiguating accesses to the data structures used in the multiple-precision, interpreted arithmetic routines. This problem, due to the custom memory allocation scheme not being recognized by the compiler as providing unique destination "bags," prevents otherwise very profitable loop pipelining. IMPACT could extract a significant amount of performance here if it could, through either disambiguation or data speculation, reorder reads and writes in the multiple-precision arithmetic routines. Small-trip-count, nested loops (such as in multiplication of two multiple-precision numbers) present interesting region formation challenges. Cursory examination reveals unexploited peeling opportunities.

Superblock-only region formation is slightly more effective than IMPACT's Hyperblock framework, delivering 1.06× more performance. The primary reason for this difference is a two-step phenomenon: First, a relatively unprofitable general Hyperblock (See Section 5.4.3) is formed in the `Lblock` phase of the compiler. Next, in the `Lsuperscalar` phase, the ILP enhancement phase, where Superblock formation is run, this Hyperblock



(a) (b)

Figure C.10 Execution profile for *254.gap*. Execution times are for code with (a) nonpredicated and (b) predicated CFS transformation applied, relative to classically optimized code.

Table C.18 Cycle accounting data for *254.gap*

Accounting category	**O-NS**	**S-NS**	**S-CS**	**I-NS**	**I-CS**
unstalled	1.154E+11	1.041E+11	8.475E+10	1.073E+11	8.916E+10
flush br	6.907E+09	6.475E+09	6.539E+09	5.787E+09	5.854E+09
flush xpn	1.057E+07	1.068E+07	3.332E+08	1.051E+07	3.470E+08
micropipe	1.565E+10	1.495E+10	1.643E+10	1.622E+10	1.733E+10
scorebd gr/gr	3.034E+09	3.503E+09	3.697E+09	3.812E+09	3.729E+09
scorebd gr/ld	4.351E+10	4.235E+10	4.131E+10	4.295E+10	4.192E+10
scorebd fr	3.227E+09	2.938E+09	2.876E+09	3.141E+09	2.806E+09
scorebd misc	7.350E+08	3.740E+08	9.385E+08	5.127E+08	1.574E+09
reg stack	6.535E+08	8.841E+08	1.082E+09	7.912E+08	1.635E+09
front end	4.027E+09	3.961E+09	5.092E+09	3.286E+09	4.209E+09
KERNEL	8.685E+08	7.104E+08	5.557E+09	5.762E+08	5.832E+09
TOTAL	1.940E+11	1.802E+11	1.686E+11	1.843E+11	1.744E+11

Table C.19 Instruction accounting data for *254.gap*

	O-NS	**S-NS**	**S-CS**	**I-NS**	**I-CS**
Useful instr.	2.258E+11	2.173E+11	2.202E+11	2.179E+11	2.244E+11
Pred. squashed instr.	1.501E+09	1.487E+09	1.472E+09	8.613E+09	7.732E+09
`nop` instr.	7.259E+10	6.123E+10	4.589E+10	5.868E+10	4.421E+10
Total instr.	2.999E+11	2.800E+11	2.675E+11	2.852E+11	2.764E+11

prevents the formation of a desirable (and more profitable) Superblock region. The author has enhanced the Superblock former to be able to form Superblocks through existing Hyperblocks where it is possible to do so without computing new predicates. This compiler improvement, which relaxes a previously annoying phase ordering constraint, closes approximately half the gap between the Hyperblock-enabled model and the more profitable Superblock-only scheme.

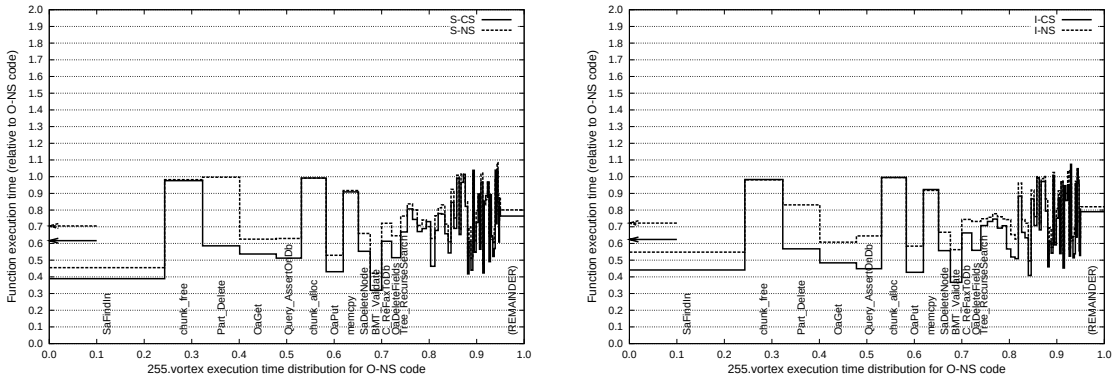
A run of *gap* with its reference input used for training yields a 1.06× speedup, indicating that profile stability is a problem in this benchmark, as is common with interpretive-style applications. Interprocedural pointer analysis yielded a 1.06× speedup. Intel's reference compiler improves *gap*'s performance by approximately 15% using a patented scheme for stride-detecting software prefetching [54]. IMPACT lacks this capability and so trails Intel's performance, but only by a small fraction of this difference.

S-CS and **I-CS** results are strikingly similar. The **S-CS** result benefits 3% from branch target expansion. Both **S-CS** and **I-CS** are incurring a tripling of kernel cycle time for *gap*. This is due to an unhandled instances of wild loads, as described in Section 6.7.

C.3.10 *255.vortex*

Vortex (Virtual Object Runtime Expository) is an object-oriented database application. Three inputs simulate various dataset sizes and insert, delete, and lookup patterns. Much of *vortex*'s code checks for error conditions that simply never occur. Profile-based inlining, trace-based optimization, and control dependence height reduction are therefore very successful in providing high performance. Predication seems to be relatively unimportant, compared to simple Superblock trace formation. A traditional Superblock region approach achieves similar performance results to the complex Hyperblock former used typically.

Figure C.11 shows the execution time for the program's functions, relative to the baseline (**O-NS**) version, in nonpredicated (**S-NS** and **S-CS**) and predicated (**I-NS** and **I-CS**) configurations. Tables C.20 and C.21 indicate cycle accounting and instruction counting data, respectively.



(a) (b)

Figure C.11 Execution profile for *255.vortex*. Execution times are for code with (a) non-predicated and (b) predicated CFS transformation applied, relative to classically optimized code.

Table C.20 Cycle accounting data for *255.vortex*

Accounting category	O-NS	S-NS	S-CS	I-NS	I-CS
unstalled	1.056E+11	6.320E+10	4.193E+10	6.391E+10	4.489E+10
flush br	1.706E+09	1.268E+09	1.374E+09	1.107E+09	1.073E+09
flush xpn	3.827E+07	3.770E+07	3.769E+07	3.796E+07	3.996E+07
micropipe	2.933E+10	2.752E+10	2.912E+10	2.953E+10	2.970E+10
scorebd gr/gr	2.956E+08	1.824E+08	1.009E+08	1.938E+08	1.046E+08
scorebd gr/ld	5.476E+10	5.170E+10	5.267E+10	5.333E+10	5.066E+10
scorebd fr	4.329E+09	3.742E+09	2.621E+09	2.697E+09	2.085E+09
scorebd misc	3.241E+08	4.395E+08	2.688E+08	4.260E+08	3.067E+08
reg stack	1.963E+09	2.461E+09	2.791E+09	2.874E+09	3.689E+09
front end	2.036E+10	4.042E+09	4.131E+09	4.072E+09	3.940E+09
KERNEL	1.404E+09	1.098E+09	1.151E+09	1.236E+09	1.165E+09
TOTAL	2.201E+11	1.557E+11	1.362E+11	1.594E+11	1.377E+11

Table C.21 Instruction accounting data for *255.vortex*

	O-NS	S-NS	S-CS	I-NS	I-CS
Useful instr.	2.350E+11	1.577E+11	1.595E+11	1.639E+11	1.650E+11
Pred. squashed instr.	5.218E+09	5.211E+09	5.221E+09	1.067E+10	1.136E+10
`nop` instr.	7.551E+10	3.319E+10	1.773E+10	3.859E+10	1.487E+10
Total instr.	3.158E+11	1.961E+11	1.825E+11	2.131E+11	1.913E+11

Vortex performance is very sensitive to inlining, as indicated in Chapter 7, which enables formation of large execution traces. These traces are then exploited with heavy utilization of control speculation.

Instruction cache and instruction-side translation lookaside buffer (TLB) performance vary substantially with optimization level. Comparing I-CS to O-NS, the optimized version issues 39% fewer instruction cache accesses with the L1I hit rate simultaneously improving from 95% to 98%. The number of instruction-related accesses to L2 is cut by a factor of five. The number of L1I TLB misses is also reduced by 44%. Specialization, although it substantially increases the total application code footprint, has a net beneficial effect on instruction fetch and issue in this program.

Predication buys little additional performance in *vortex*, with a Superblock-only region formation strategy being approximately as effective. In several procedures, predication achieves a benefit, but a substantial loss in `SaFindIn()` more than offsets these gains. In this function, the inclusion of a small piece of infrequently executed error checking code in the main loop Hyperblock results in a loss of key optimization potential (register promotion-enabled constant propagation) for the remainder of the block. Optimization of logical conditions and the predicate define network might also yield improvements. The situation could be improved with substantial improvements in the optimizer, resulting in a positive margin of a few percentage points improvement for **I-CS** relative to **S-CS**.

Vortex was particularly responsive to interprocedural pointer analysis, with FULCRA IPA [59] achieving a speedup of 1.18× relative to non-IPA optimization; a relatively simple application of IPA (field-insensitive, context-insensitive, Andersen's), however, was sufficient to accomplish this improvement (See Section B.6).

Vortex garners about a 5% improvement in performance due to modulo scheduling. Post-Superblock-formation optimizations (performed in `Lsuperscalar`) achieved a remarkable 11% improvement in performance in *vortex*. It was the only benchmark to respond with more than a 3% change to these optimizations.

The functions `chunk_free()`, `chunk_alloc()`, and `memcpy()`, the three prominent functions with little benefit in Table C.20, are provided from gcc-compiled system libraries. The substantial contribution of these currently unoptimizable functions motivates library and cross-module compilation using IMPACT. *Vortex* spends a quarter of its execution time dealing with data address translation and could benefit somewhat from larger pages (as many accesses wind up traversing the HPW).

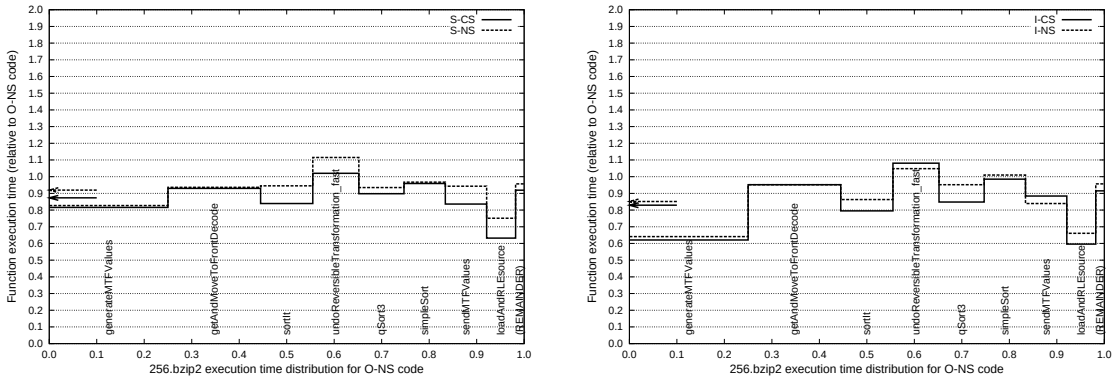
C.3.11 *256.bzip2*

Bzip2 is a Burrows-Wheeler compressor, as embodied in Julian Seward's familiar *NIX utility.

Figure C.12 shows the execution time for the program's functions, relative to the baseline (**O-NS**) version, in nonpredicated (**S-NS** and **S-CS**) and predicated (**I-NS** and **I-CS**) configurations. Tables C.22 and C.23 indicate cycle accounting and instruction counting data, respectively.

Of all the SPEC CINT2000 benchmarks *bzip2* benefits most markedly from the use of predication, which achieves a 1.20× speedup relative to Superblock-only code. Surprisingly, almost none of this benefit is due to traditional Hyperblock formation techniques. The majority of this large benefit is due to the combination of two factors: first, the use of predication to implement the new stage predicate unrolling schema described in Section 8.4; and, second, the development of store-load dependence avoidance techniques in the dependence test and instruction scheduling portions of the compiler. This reduction in execution cycles, spread across several loops in the various prominent functions of *bzip2*, appears as a prominent reduction in L1D micropipeline and load dependence stalls. Prior to the implementation of this technique, predication yielded only a 1.03× speedup relative to the Superblock-only code. This is not surprising, since even with the technique, branch misprediction stalls are reduced by 10%, amounting to only 1% of execution time. Unstalled execution cycles are hardly impacted.

Interprocedural pointer analysis, including Omega test, yielded a 1.12× speedup. This is almost entirely due to manipulation of short-term store-load dependences in the critical



(a) (b)

Figure C.12 Execution profile for *256.bzip2*. Execution times are for code with (a) non-predicated and (b) predicated CFS transformation applied, relative to classically optimized code.

Table C.22 Cycle accounting data for *256.bzip2*

Accounting category	**O-NS**	**S-NS**	**S-CS**	**I-NS**	**I-CS**
unstalled	1.258E+11	1.048E+11	9.272E+10	1.075E+11	9.398E+10
flush br	1.882E+10	1.835E+10	1.918E+10	1.755E+10	1.749E+10
flush xpn	4.247E+06	5.362E+06	6.813E+06	6.604E+06	2.189E+07
micropipe	3.856E+10	4.081E+10	4.246E+10	3.857E+10	4.084E+10
scorebd gr/gr	2.658E+09	2.288E+09	2.283E+09	1.936E+09	2.238E+09
scorebd gr/ld	5.151E+10	6.483E+10	5.762E+10	3.323E+10	3.658E+10
scorebd fr	2.885E+05	2.568E+05	1.170E+05	1.606E+05	1.383E+05
scorebd misc	9.837E+07	9.802E+07	4.818E+04	1.012E+08	2.876E+06
reg stack	5.822E+04	1.190E+05	8.276E+07	2.638E+07	1.234E+08
front end	3.522E+09	3.675E+09	3.835E+09	3.263E+09	3.396E+09
KERNEL	8.986E+08	8.645E+08	9.533E+08	9.859E+08	1.585E+09
TOTAL	2.419E+11	2.357E+11	2.191E+11	2.032E+11	1.963E+11

Table C.23 Instruction accounting data for *256.bzip2*

	O-NS	**S-NS**	**S-CS**	**I-NS**	**I-CS**
Useful instr.	3.040E+11	2.801E+11	2.858E+11	2.858E+11	2.965E+11
Pred. squashed instr.	1.437E+09	1.459E+09	1.446E+09	1.238E+10	1.151E+10
`nop` instr.	7.230E+10	6.866E+10	6.388E+10	4.786E+10	3.956E+10
Total instr.	3.778E+11	3.502E+11	3.512E+11	3.461E+11	3.476E+11

loop mentioned previously. *Bzip2* performance benefited by 2% from `Lsuperscalar` optimizations (beyond region formation and unrolling).

C.3.12 *300.twolf*

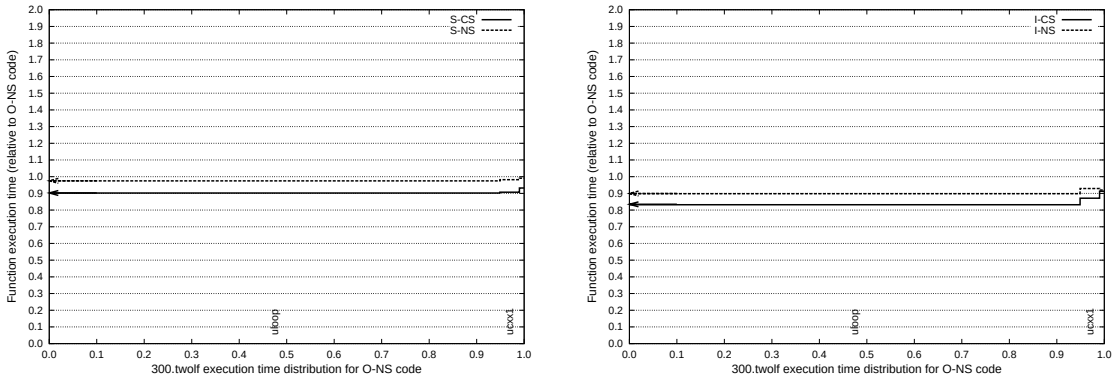
Twolf is based on the TimberWolfSC placement and routing application for microprocessor lithography. Given a collection of standard cells, *twolf* iteratively arranges, rotates, and interconnects them. The algorithm used relies on a simulated annealing technique; a pseudorandom number generator is frequently used to select the next cell to reposition.

Figure C.13 shows the execution time for the program's functions, relative to the baseline (**O-NS**) version, in nonpredicated (**S-NS** and **S-CS**) and predicated (**I-NS** and **I-CS**) configurations. Tables C.24 and C.25 indicate cycle accounting and instruction counting data, respectively.

IMPACT in the **I-CS** configuration achieves good performance in *twolf*. IMPACT is very successful at forming good Hyperblock loop regions, but software pipelining only achieves a 1% speedup due to constraining initiation intervals. Predication achieves a 1.12× speedup in *twolf* relative to Superblock-only code, attributable to three sources: (1) the overlapping of independent control constructs allowed by peeling and if-conversion transformations; (2) the elimination of branch misprediction penalty; and (3) more instruction-cache-efficient exploitation of ILP.

Peeling is very active in *twolf*, overlapping the initial iterations of loops with each other and with long, previous divide operations (IMPACT inlines all divide macro operations on the Itanium architecture). As was indicated in Section 5.7.2, *twolf* benefits from peeling, but peeling causes it to suffer an increase in instruction cache misses. Nonetheless, examination of code suggests that, in certain cases, more aggressive peeling may be warranted. More selective peeling methods could achieve increased benefit, if they could allow more peeling in profitable cases while curtailing code growth in less-promising locales.

Front end bubble cycles are increased by 50% in *twolf* **I-CS**, due to the extensive code replication incurred by peeling, but even so they account for less than 5% of execution time. (**S-CS** even more heavily impacts instruction fetch performance.) **I-CS** reduces



(a) (b)

Figure C.13 Execution profile for *300.twolf*. Execution times are for code with (a) non-predicated and (b) predicated CFS transformation applied, relative to classically optimized code.

Table C.24 Cycle accounting data for *300.twolf*

Accounting category	**O-NS**	**S-NS**	**S-CS**	**I-NS**	**I-CS**
unstalled	1.600E+11	1.310E+11	1.344E+11	1.508E+11	1.269E+11
flush br	3.234E+10	4.069E+10	4.024E+10	1.374E+10	1.350E+10
flush xpn	3.438E+06	3.387E+06	3.281E+06	3.099E+06	3.010E+06
micropipe	2.265E+10	2.395E+10	2.438E+10	2.548E+10	2.698E+10
scorebd gr/gr	2.431E+06	2.431E+06	2.431E+06	2.431E+06	2.431E+06
scorebd gr/ld	1.239E+11	1.330E+11	1.268E+11	1.362E+11	1.220E+11
scorebd fr	5.503E+10	4.609E+10	2.871E+10	3.338E+10	3.074E+10
scorebd misc	1.537E+09	8.339E+08	8.739E+07	4.392E+08	4.930E+08
reg stack	1.533E+08	2.843E+08	3.170E+08	7.280E+08	7.161E+08
front end	1.151E+10	1.995E+10	2.524E+10	1.760E+10	1.832E+10
KERNEL	1.014E+09	1.136E+09	1.001E+09	8.969E+08	8.128E+08
TOTAL	4.081E+11	3.969E+11	3.812E+11	3.793E+11	3.405E+11

Table C.25 Instruction accounting data for *300.twolf*

	O-NS	**S-NS**	**S-CS**	**I-NS**	**I-CS**
Useful instr.	3.414E+11	3.290E+11	3.371E+11	3.224E+11	3.397E+11
Pred. squashed instr.	2.044E+09	2.133E+08	2.243E+08	3.115E+10	3.005E+10
`nop` instr.	1.269E+11	1.248E+11	1.178E+11	1.148E+11	1.012E+11
Total instr.	4.704E+11	4.541E+11	4.551E+11	4.684E+11	4.709E+11

branch misprediction flush cycles markedly relative to both approaches. As in *crafty*, the author experimented with combining common code segments between control-exclusive regions to reduce code size. (In *twolf*, this would have required either rewriting the application or doing complicated semantic matching in the compiler, after inlining, so these transformations were done by hand.) These experiments did not produce positive results, as the versioning allowed for by correct, individual profiles was more valuable to performance than any achievable reduction in code size.

The **S-CS** performance improved 2% with branch target expansion, and could benefit another 2% from a more aggressive application. Increasing losses from instruction cache pressure limit the gains achievable within this model, however, relative to the **I-CS** approach. Interprocedural pointer analysis delivers a 1.07× speedup, requiring a field-sensitive approach to do so.

Comparing to a production compiler, `icc -O3 -ipo -prof_gen|prof_use` achieves a score of 923, while the `-O2` version delivers only a 798. These scores bookend IMPACT's **I-CS** result. `icc -O3` offers high-level loop transformations and prefetching, which are not accessible to IMPACT. The benefits from these transformations, which are measurable as decreases in integer load stall time, would likely add orthogonally to the performance of *twolf* if combined in a common infrastructure.

Finally, *twolf* has many performance-influencing integer division operations whose divisors (such as the number of pins) are variable but likely stable. Profiling could probably identify common cases for specialization as cheaper operation sequences than full divisions.

REFERENCES

[1] R. Ronen, A. Mendelson, K. Lai, S.-L. Lu, F. Pollack, and J. Shen, "Coming challenges in microarchitecture and architecture," *Proceedings of the IEEE*, vol. 89, no. 3, pp. 329–340, 2001.

[2] International Technology Roadmap for Semiconductors, "International Technology Roadmap for Semiconductors," 2003, http://public.itrs.net/.

[3] D. Culler and J. Singh, *Parallel Computer Architecture: A Hardware/Software Approach*. San Francisco, CA: Morgan Kaufmann, 1999.

[4] M. S. Schlansker, B. R. Rau, S. Mahlke, V. Kathail, R. Johnson, S. Anik, and G. Abraham, "Achieving high levels of instruction-level parallelism with reduced hardware complexity," Hewlett-Packard Laboratory, Tech. Rep. HPL-96-120, November 1994.

[5] Standard Performance Evaluation Corporation, "SPEC CINT2000 benchmarks," 2000, http://www.spec.org/cpu2000.

[6] N. P. Jouppi and D. W. Wall, "Available instruction-level parallelism for superscalar and superpipelined machines," in *Proceedings of the 3rd International Conference on Architectural Support for Programming Languages and Operating Systems*, April 1989, pp. 272–282.

[7] M. D. Smith, M. Johnson, and M. A. Horowitz, "Limits on multiple instruction issue," in *Proceedings of the 3rd International Conference on Architectural Support for Programming Languages and Operating Systems*, April 1989, pp. 290–302.

[8] D. W. Wall, "Limits of instruction-level parallelism," in *Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems*, April 1991, pp. 176–188.

[9] M. A. Postiff, D. A. Greene, G. S. Tyson, and T. N. Mudge, "The limits of instruction level parallelism in SPEC95 applications," *Computer Architecture News*, vol. 27, pp. 31–34, March 1999.

[10] M. S. Lam and R. P. Wilson, "Limits of control flow on parallelism," in *Proceedings of the 19th International Symposium on Computer Architecture*, May 1992, pp. 46–57.

[11] Standard Performance Evaluation Corporation, "SPEC CINT95 benchmarks," 1995, http://www.spec.org/cpu95.

[12] H.-H. Lee, Y. Wu, and G. Tyson, "Quantifying instruction-level parallelism limits on an epic architecture," in *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, April 2000, pp. 21–27.

[13] M. Hrishikesh, N. Jouppi, K. Farkas, D. Burger, S. Keckler, and P. Shivakumar, "The optimal logic depth per pipeline stage is 6 to 8 FO4 inverter delays," in *Proceedings of the 29th International Symposium on Computer Architecture*, June 2002, pp. 14–24.

[14] E. Sprangle and D. Carmean, "Increasing processor performance by implementing deeper pipelines," in *Proceedings of the 29th Annual International Symposium on Computer Architecture*, May 2002, pp. 14–24.

[15] V. Agarwal, M. Hrishikesh, S. Keckler, and D. Burger, "Clock rate versus IPC: The end of the road for conventional microarchitectures," in *Proceedings of the 27th International Symposium on Computer Architecture*, June 2000, pp. 248–259.

[16] G. Moore, "No exponential is forever...but we can delay forever," Keynote address at the 2004 International Solid State Circuits Conference, Feburary 2003.

[17] A. Hartstein and T. Puzak, "Optimum power/performance pipeline depth," in *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*, December 2003, pp. 117–125.

[18] R. M. Tomasulo, "An efficient algorithm for exploiting multiple arithmetic units," *IBM Journal of Research and Development*, vol. 11, pp. 25–33, January 1967.

[19] F. Pollack, "New microarchitecture challenges in the coming generations of CMOS process technologies." Keynote at the 32nd International Symposium on Microarchitecture, December 1999.

[20] D. Carmean, "The Pentium 4 processor." Hot Chips 13, Stanford University, Palo Alto, CA, August 2001.

[21] V. Kathail, M. S. Schlansker, and B. R. Rau, "HPL PlayDoh architecture specification: Version 1.0," Hewlett-Packard Laboratories, Palo Alto, CA, Tech. Rep. HPL-93-80, February 1994.

[22] W. W. Hwu, S. A. Mahlke, W. Y. Chen, P. P. Chang, N. J. Warter, R. A. Bringmann, R. G. Ouellette, R. E. Hank, T. Kiyohara, G. E. Haab, J. G. Holm, and D. M. Lavery, "The Superblock: An effective technique for VLIW and superscalar compilation," *The Journal of Supercomputing*, vol. 7, pp. 229–248, January 1993.

[23] M. S. Schlansker and B. R. Rau, "EPIC: Explicitly Parallel Instruction Computing," *IEEE Computer*, vol. 33, pp. 37–45, February 2000.

[24] J. C. Dehnert, P. Y. Hsu, and J. P. Bratt, "Overlapped loop support in the Cydra 5," in *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems*, April 1989, pp. 26–38.

[25] P. G. Lowney, S. M. Freudenberger, T. J. Karzes, W. D. Lichtenstein, R. P. Nix, J. S. O'Donnell, and J. C. Ruttenberg, "The Multiflow Trace scheduling compiler," *The Journal of Supercomputing*, vol. 7, pp. 51–142, January 1993.

[26] B. R. Rau and J. A. Fisher, "Instruction-level parallel processing: History, overview, and perspective," *The Journal of Supercomputing*, vol. 7, pp. 9–50, January 1993.

[27] M. Schlansker and B. R. Rau, "EPIC: An architechure for instruction parallel processors," Hewlett-Packard Laboratory, Tech. Rep. HPL-1999-111, February 2000.

[28] R. D. Barnes, E. M. Nystrom, J. W. Sias, N. Navarro, S. J. Patel, and W. W. Hwu, "Beating in-order stalls with 'flea-flicker' two-pass pipelining," in *Proceedings of 36th Annual International Symposium on Microarchitecture*, December 2003, pp. 387–398.

[29] P. H. Wang, H. Wang, J. D. Collins, E. Grochowski, R. M. Kling, and J. P. Shen, "Memory latency-tolerance approaches for Itanium processors: Out-of-order execution vs. speculative precomputation," in *Proceedings of the 8th International Symposium on High-Performance Computer Architecture*, February 2002, pp. 167–176.

[30] A. Glew, "MLP yes! ILP no," in *Proceedings of the ASPLOS Wild and Crazy Ideas Forum*, October 1998, http://www.cs.berkeley.edu/~kubitron/ asplos98/slides/andrew_glew.pdf.

[31] L. Gwennap, "Intel, HP make EPIC disclosure," *Microprocessor Report*, vol. 11, pp. 1–9, October 1997.

[32] J. A. Fisher and S. M. Freudenberger, "Predicting conditional branch directions from previous runs of a program," in *Proceedings of 5th International Conference on Architectual Support for Programming Languages and Operating Systems*, October 1992, pp. 85–95.

[33] B. R. Rau, M. S. Schlansker, and P. P. Tirumalai, "Code generation schema for modulo scheduled loops," in *Proceedings of the 25th Annual International Symposium on Microarchitecture*, December 1992, pp. 158–169.

[34] J. Bharadwaj, K. Menezes, and C. McKinsey, "Wavefront scheduling: Path based data representation and scheduling of subgraphs," in *Proceedings of 32nd Annual International Symposium on Microarchitecture*, December 1999, pp. 262–271.

[35] J. Bharadwaj, W. Chen, W. Chuang, G. Hoflehner, K. Menezes, K. Muthukumar, and J. Pierce, "The Intel IA-64 compiler code generator," *IEEE Micro*, vol. 20, pp. 44–53, October 2000.

[36] S. A. Mahlke, D. C. Lin, W. Y. Chen, R. E. Hank, R. A. Bringmann, and W. W. Hwu, "Effective compiler support for predicated execution using the Hyperblock," in *Proceedings of the 25th International Symposium on Microarchitecture*, December 1992, pp. 45–54.

[37] W. W. Hwu, R. E. Hank, D. M. Gallagher, S. A. Mahlke, D. M. Lavery, G. E. Haab, J. C. Gyllenhaal, and D. I. August, "Compiler technology for future microprocessors," *Proceedings of the IEEE*, vol. 83, pp. 1625–1640, December 1995.

[38] D. I. August, D. A. Connors, S. A. Mahlke, J. W. Sias, K. M. Crozier, B. Cheng, P. R. Eaton, Q. B. Olaniran, and W. W. Hwu, "Integrated predicated and speculative execution in the IMPACT EPIC architecture," in *Proceedings of the 25th International Symposium on Computer Architecture*, June 1998, pp. 227–237.

[39] Y. Choi, A. Knies, L. Gerke, and T. Ngai, "The impact of if-conversion and branch prediction on program execution on the Intel Itanium Processor," in *Proceedings of the 34th International Symposium on Microarchitecture*, December 2001, pp. 182–191.

[40] S. Triantafyllis, M. Vachharajani, N. Vachharajani, and D. August, "Compiler optimization-space exploration," in *Proceedings of the 2003 International Symposium on Code Generation and Optimization*, March 2003, pp. 204–215.

[41] C. McNairy and D. Soltis, "Itanium 2 processor microarchitecture," *IEEE Micro*, vol. 23, pp. 44–55, March 2003.

[42] W. W. Hwu, J. W. Sias, M. C. Merten, E. M. Nystrom, R. D. Barnes, C. J. Shannon, S. Ryoo, and J. V. Olivier, "Itanium performance insights," Microprocessor Forum, San Jose, CA, October 2001.

[43] Gelato/UIUC OpenIMPACT Effort, "The OpenIMPACT IA-64 compiler," 2005, http://gelato.uiuc.edu/.

[44] J. A. Fisher, "Trace scheduling: A technique for global microcode compaction," *IEEE Transactions on Computers*, vol. C-30, pp. 478–490, July 1981.

[45] M. S. Lam, "Software pipelining: An effective scheduling technique for VLIW machines," in *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation*, June 1988, pp. 318–328.

[46] P. P. Chang, S. A. Mahlke, W. Y. Chen, and W. W. Hwu, "Profile-guided automatic inline expansion for C programs," *Software Practice and Experience*, vol. 22, pp. 349–370, May 1992.

[47] S. A. Mahlke, W. Y. Chen, W. W. Hwu, B. R. Rau, and M. S. Schlansker, "Sentinel scheduling for VLIW and superscalar processors," in *Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1992, pp. 238–247.

[48] D. I. August, W. W. Hwu, and S. A. Mahlke, "A framework for balancing control flow and predication," in *Proceedings of the 30th Annual International Symposium on Microarchitecture*, December 1997, pp. 92–103.

[49] A. Aho, R. Sethi, and J. Ullman, *Compilers: Principles, Techniques, and Tools*. Reading, MA: Addison-Wesley, 1986.

[50] J. Bharadwaj, K. Menezes, and C. McKinsey, "Wavefront scheduling: Path based data representation and scheduling of subgraphs," *Journal of Instruction-Level Parallelism*, vol. I, May 2000, http://www.jilp.org/vol2/v2paper12.pdf.

[51] Intel Corporation, *Intel Itanium 2 Processor Reference Manual for Software Development and Optimization*, Document Number 251110-001, June 2002.

[52] J. W. Sias, S.-Z. Ueng, G. A. Kent, I. M. Steiner, E. M. Nystrom, and W. W. Hwu, "Field testing IMPACT EPIC research results in Itanium 2," in *Proceedings of the 31st Annual International Symposium on Computer Architecture*, June 2004, pp. 26–37.

[53] S. G. Abraham, R. A. Sugumar, D. Windheiser, B. R. Rau, and R. Gupta, "Predictability of load/store instruction latencies," in *Proceedings of the 26th International Symp. on Microarchitecture*, December 1993, pp. 139–152.

[54] Y. Wu, "Efficient discovery of regular stride patterns in irregular programs and its use in compiler prefetching," in *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, 2002, pp. 210–221.

[55] J. W. Sias, W. W. Hwu, and D. I. August, "Accurate and efficient predicate analysis with binary decision diagrams," in *Proceedings of 33rd Annual International Symposium on Microarchitecture*, December 2000, pp. 112–123.

[56] R. P. Colwell, R. P. Nix, J. J. O'Donnell, D. B. Papworth, and P. K. Rodman, "A VLIW architecture for a trace scheduling compiler," in *Proceedings of the 2nd International Conference on Architectural Support for Programming Languages and Operating Systems*, April 1987, pp. 180–192.

[57] Intel Corporation, *Intel IA-64 Architecture Software Developer's Manual Volume 1: Application Architecture*, Document Number 245317-003, December 2001.

[58] J. Huck, D. Morris, J. Ross, A. Knies, H. Mulder, and R. Zahir, "Introducing the IA-64 architecture," *IEEE Micro*, vol. 20, pp. 12–23, September 2000.

[59] E. M. Nystrom, "FULCRA pointer analysis framework," Ph.D. dissertation, Department of Electrical and Computer Engineering, University of Illinois at Urbana-Champaign, 2005.

[60] W. Pugh, "The Omega Test: A fast and practical integer programming algorithm for dependence analysis," in *Proceedings of Supercomputing 1991*, November 1991, pp. 4–13.

[61] S. Muchnick, *Advanced Compiler Design and Implementation*. San Francisco, CA: Morgan Kaufmann Publishers, 1997.

[62] J. Henning, "SPEC CPU2000: Measuring CPU performance in the new millennium," *IEEE Computer*, vol. 33, pp. 28–35, July 2000.

[63] S. Eranian, "Perfmon: linux performance monitoring for IA64," 2003, http://www.hpl.hp.com/research/linux/perfmon/.

[64] J. Lin, T. Chen, W. Hsu, P. Yew, R. Ju, T. Ngai, and S. Chan, "A compiler framework for speculative analysis and optimizations," in *Proceedings of PLDI 2003*, 2003, pp. 289–299.

[65] B. L. Deitrich and W. W. Hwu, "Speculative hedge: Regulating compile-time speculation against profile variations," in *Proceedings of the 29th International Symposium on Microarchitecture*, December 1996, pp. 70–79.

[66] S. McFarling, "Reality-based optimization," in *Proceedings of the 2003 Conference on Code Generation and Optimization*, March 2003, pp. 59–68.

[67] M. C. Merten, "A framework for profile-driven optimization in the IMPACT binary reoptimization system," M.S. thesis, Department of Electrical and Computer Engineering, University of Illinois at Urbana-Champaign, 1999.

[68] M. S. Schlansker, S. A. Mahlke, and R. Johnson, "Control CPR: A branch height reduction optimization for EPIC architectures," in *Proceedings of the ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation*, May 1999, pp. 155–168.

[69] R. E. Hank, W. W. Hwu, and B. R. Rau, "Region-based compilation: An introduction and motivation," in *Proceedings of the 28th Annual International Symposium on Microarchitecture*, December 1995, pp. 158–168.

[70] R. E. Hank, "Region based compilation," Ph.D. dissertation, Department of Electrical and Computer Engineering, University of Illinois at Urbana-Champaign, 1996.

[71] W. Pugh and D. Wonnacott, "Eliminating false data dependences using the Omega Test," in *Proceedings of the ACM SIGPLAN 1992 Conference on Programming Language Design and Implementation*, June 1992, pp. 140–151.

[72] R. D. Ju, K. Nomura, U. Mahadevan, and L.-C. Wu, "A unified framework for control and data speculation," in *Proceedings of the 2000 International Conference on Parallel Architectures and Compilation Techniques*, October 2000, pp. 157–168.

[73] D. M. Gallagher, W. Y. Chen, S. A. Mahlke, J. C. Gyllenhaal, and W. W. Hwu, "Dynamic memory disambiguation using the memory conflict buffer," in *Proceedings of 6th International Conference on Architectual Support for Programming Languages and Operating Systems*, October 1994, pp. 183–193.

[74] B. C. Cheng, "A profile-driven automatic inliner for the IMPACT compiler," M.S. thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, 1997.

[75] J. Ferrante, K. J. Ottenstein, and J. D. Warren, "The program dependence graph and its use in optimization," *ACM Transactions on Programming Languages and Systems*, vol. 9, pp. 319–349, July 1987.

[76] J. C. Park and M. S. Schlansker, "On predicated execution," Hewlett Packard Laboratories, Palo Alto, CA, Tech. Rep. HPL-91-58, May 1991.

[77] N. J. Warter, G. E. Haab, K. Subramanian, and J. W. Bockhaus, "Enhanced modulo scheduling for loops with conditional branches," in *Proceedings of the 25th Annual International Symposium on Microarchitecture*, December 1992, pp. 170–179.

[78] B. Kernighan and D. Ritchie, *The C Programming Language*. Englewood Cliffs, NJ: Prentice-Hall, Inc., 1988.

[79] S. A. Mahlke, "Exploiting instruction level parallelism in the presence of conditional branches," Ph.D. dissertation, University of Illinois at Urbana-Champaign, 1995.

[80] Intel Corporation, *Intel IA-64 Architecture Software Developer's Manual Volume 3: Instruction Set Reference*, Document Number 245319-003, December 2001.

[81] S. A. Mahlke, R. E. Hank, J. McCormick, D. I. August, and W. W. Hwu, "A comparison of full and partial predicated execution support for ILP processors," in *Proceedings of the 22th International Symposium on Computer Architecture*, June 1995, pp. 138–150.

[82] B. R. Rau, D. W. L. Yen, W. Yen, and R. A. Towle, "The Cydra 5 departmental supercomputer," *IEEE Computer*, vol. 22, pp. 12–35, January 1989.

[83] D. I. August, "Hyperblock performance optimizations for ILP processors," M.S. thesis, Department of Electrical and Computer Engineering, University of Illinois at Urbana-Champaign, 1996.

[84] D. I. August, J. W. Sias, J. Puiatti, S. A. Mahlke, D. A. Connors, K. M. Crozier, and W. W. Hwu, "The program decision logic approach to predicated execution," in *Proceedings of the 26th International Symposium on Computer Architecture*, May 1999, pp. 208–219.

[85] L. Carter, B. Simon, B. Calder, L. Carter, and J. Ferrante, "Predicated single static assignment," in *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, October 1999, pp. 245–255.

[86] A. E. Eichenberger, W. Meleis, and S. Maradani, "An integrated approach to accelerate data and predicate computations in Hyperblocks," in *Proceedings of the 33rd Annual International Symposium on Microarchitecture*, December 2000, pp. 101–111.

[87] V. H. Allan, J. Janardhan, R. M. Lee, and M. Srinivas, "Enhanced region scheduling on a program dependence graph," in *Proceedings of the 25th Annual International Symposium on Microarchitecture*, December 1992, pp. 72–80.

[88] S. Ryoo, "Partial code elimination in the IMPACT compiler framework," M.S. thesis, Department of Electrical and Computer Engineering, University of Illinois at Urbana-Champaign, 2004.

[89] W. W. Hwu, D. I. August, and J. W. Sias, "Program decision logic optimization using predication and control speculation," *Proceedings of the IEEE*, vol. 89, pp. 1660–1675, November 2001.

[90] R. E. Bryant, "Symbolic Boolean manipulation with ordered binary decision diagrams," School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, Tech. Rep. CMU-CS-92-160, October 1992.

[91] J. W. Sias, "Condition awareness support for predicate analysis and optimization," M.S. thesis, Department of Electrical and Computer Engineering, University of Illinois at Urbana-Champaign, 1999.

[92] D. August, "Systematic compilation for predicated execution," Ph.D. dissertation, Department of Electrical and Computer Engineering, University of Illinois at Urbana-Champaign, 2000.

[93] R. Johnson and M. Schlansker, "Analysis techniques for predicated code," in *Proceedings of the 29th International Symposium on Microarchitecture*, December 1996, pp. 100–113.

[94] D. M. Gillies, D. R. Ju, R. Johnson, and M. Schlansker, "Global predicate analysis and its application to register allocation," in *Proceedings of the 29th International Symposium on Microarchitecture*, December 1996, pp. 114–125.

[95] R. C. Johnson and M. S. Schlansker, "Compiling a Predicated Code with Direct Analysis of the Predicated Code," U. S. Patent No. 5,920,716, July 1999.

[96] A. E. Eichenberger and E. S. Davidson, "Register allocation for predicated code," in *Proceedings of the 28th Annual International Symposium on Microarchitecture*, December 1995, pp. 180–191.

[97] T.-Y. Yeh and Y. N. Patt, "Alternative implementation of two-level adaptive branch prediction," in *Proceedings of the 19th International Symposium on Computer Architecture*, May 1992, pp. 124–134.

[98] S. A. Mahlke, R. E. Hank, R. A. Bringmann, J. C. Gyllenhaal, D. M. Gallagher, and W. W. Hwu, "Characterizing the impact of predicated execution on branch prediction," in *Proceedings of the 27th International Symposium on Microarchitecture*, December 1994, pp. 217–227.

[99] G. S. Tyson, "The effects of predicated execution on branch prediction," in *Proceedings of the 27th International Symposium on Microarchitecture*, December 1994, pp. 196–206.

[100] D. N. Pnevmatikatos and G. S. Sohi, "Guarded execution and branch prediction in dynamic ILP processors," in *Proceedings of the 21st International Symposium on Computer Architecture*, April 1994, pp. 120–129.

[101] J. W. Sias, H. C. Hunter, and W. W. Hwu, "Enhancing loop buffering of media and telecommunications applications using low-overhead predication," in *Proceedings of 33rd Annual International Symposium on Microarchitecture*, December 2001, pp. 262–273.

[102] D. A. Connors, J.-M. Puiatti, D. I. August, K. M. Crozier, and W. W. Hwu, "An architecture framework for introducing predicated execution into embedded microprocessors," in *Proceedings of the 5th Annual Euro-Par Conference*, August 1999, pp. 1301–1311.

[103] J. E. Smith, "A study of branch prediction strategies," in *Proceedings of the 8th International Symposium on Computer Architecture*, May 1981, pp. 135–148.

[104] W. W. Hwu and Y. N. Patt, "Checkpoint repair for high performance out-of-order execution machines," in *Proceedings of the 14th International Symposium on Computer Architecture*, June 1987, pp. 18–26.

[105] J. Smith and G. Sohi, "The microarchitecture of superscalar processors," *Proceedings of the IEEE*, vol. 83, no. 12, pp. 1609–1624, 1995.

[106] A. Moshovos, S. E. Breach, T. N. Vijaykumar, and G. S. Sohi, "Dynamic speculation and synchronization of data dependences," in *Proceedings of the 1997 International Symposium on Computer Architecture*, December 1997, pp. 181–193.

[107] S. A. Mahlke, W. Y. Chen, R. A. Bringmann, R. E. Hank, W. W. Hwu, B. R. Rau, and M. S. Schlansker, "Sentinel scheduling for VLIW and superscalar processors," *ACM Transactions on Computer Systems*, vol. 11, pp. 376–408, November 1993.

[108] M. D. Smith, "Architectural support for compile-time speculation," in *The Interaction of Compilation Technology and Computer Architecture*, Boston, MA: Kluwer Academic Publishers, 1994, pp. 13–49.

[109] R. A. Bringmann, "Compiler-controlled speculation," Ph.D. dissertation, Department of Computer Science, University of Illinois at Urbana-Champaign, 1995.

[110] J. Knoop, O. Rüthing, and B. Steffen, "Lazy code motion," in *Proceedings of the ACM SIGPLAN 1992 Conference on Programming Language Design and Implementaton*, June 1992, pp. 224–234.

[111] J. E. Smith and A. R. Pleszkun, "Implementation of precise interrupts in pipelined processors," in *Proceedings of the 12th Annual International Symposium on Computer Architecture*, June 1985, pp. 36–44.

[112] Hewlett-Packard Company, Cupertino, CA, *PA-RISC 1.1 Architecture and Instruction Set Reference Manual*, 1990.

[113] W. W. Hwu, D. A. Connors, D. I. August, and J. W. Sias, "Method and apparatus for enhancing instruction-level parallelism," United States Patent No. 6,640,315, October 2003.

[114] R. A. Bringmann, S. A. Mahlke, R. E. Hank, J. C. Gyllenhaal, and W. W. Hwu, "Speculative execution exception recovery using write-back suppression," in *Proceedings of 26th Annual International Symposium on Microarchitecture*, December 1993, pp. 214–223.

[115] M. D. Smith, M. S. Lam, and M. A. Horowitz, "Boosting beyond static scheduling in a superscalar processor," in *Proceedings of the 17th International Symposium on Computer Architecture*, May 1990, pp. 344–354.

[116] M. D. Smith, M. A. Horowitz, and M. S. Lam, "Efficient superscalar performance through boosting," in *Proceedings of the Fifth International Conference on Architecture Support for Programming Languages and Operating Systems*, October 1992, pp. 248–259.

[117] Intel Corporation, *Intel IA-64 Architecture Software Developer's Manual Volume 2: System Architecture*, Document Number 245318-003, December 2001.

[118] L.-C. Wu, R. Mirani, H. Patil, B. Olsen, and W. W. Hwu, "A new framework for debugging globally optimized code," in *Proceedings of the ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation*, May 1999, pp. 181–191.

[119] S. A. Mahlke, "Design and implementation of a portable global code optimizer," M.S. thesis, Department of Electrical and Computer Engineering, University of Illinois at Urbana-Champaign, 1991.

[120] B. R. Rau, "Iterative modulo scheduling: An algorithm for software pipelining loops," in *Proceedings of the 27th International Symposium on Microarchitecture*, December 1994, pp. 63–74.

[121] F. Somenzi, "CUDD: CU Decision Diagram package, release 2.30," 1998, http://vlsi.colorado.edu/~fabio/CUDD/.

[122] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck, "Efficiently computing static single assignment form and the control dependence graph," *ACM Transactions on Programming Languages and Systems*, vol. 13, pp. 451–490, October 1991.

[123] P. Markstein, *IA-64 and Elementary Functions: Speed and Precision*. Upper Saddle River, New Jersey: Prentice Hall PTR, 2000.

[124] V. Kathail, M. S. Schlansker, and B. R. Rau, "HPL-PD architecture specification: Version 1.1," Hewlett-Packard Laboratories, Palo Alto, CA, Tech. Rep. HPL-93-80 (R.1), February 2001.

[125] T. Kiyohara, W. W. Hwu, and W. Chen, "Memory conflict buffer for achieving memory disambiguation in compile-time code schedule," United States Patent No. 5,694,577, December 1997.

[126] D. Bacon and P. Sweeney, "Fast static analysis of C++ virtual function calls," in *Proceedings of OOPSLA 1996*, 1996, pp. 324–341.

[127] S. McFarling, "Procedure merging with instruction caches," in *Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation*, June 1991, pp. 71–79.

[128] S. Triantafyllis, M. Vachharajani, and D. I. August, "Procedure boundary elimination for EPIC compilers," in *Proceedings of the 2nd Workshop on Explicitly Parallel Instruction Computing (EPIC) Architectures and Compiler Techniques*, November 2002, pp. 70–76.

[129] K. Hazelwood and D. Grove, "Adaptive online context-sensitive inlining," in *Proceedings of the 2003 International Symposium on Code Generation and Optimization*, March 2003, pp. 253–264.

[130] T. Ball and J. R. Larus, "Efficient path profiling," in *Proceedings of 29th Annual International Symposium on Microarchitecture*, December 1996, pp. 46–57.

[131] T. Ball, P. Mataga, and M. Sagiv, "Edge profiling versus path profiling: The showdown," in *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, January 1998, pp. 134–148.

[132] D. Melski and T. Reps, "Interprocedural path profiling," Department of Computer Sciences, University of Wisconsin Madison, Tech. Rep. CS-TR-98-1382, September 1998.

[133] A. M. Holler, "Optimization for a superscalar out-of-order machine," in *Proceedings of the 26th Annual International Symposium on Microarchitecture*, November 1996, pp. 336–348.

[134] D. M. Lavery and W. W. Hwu, "Unrolling-based optimizations for modulo scheduling," in *Proceedings of the 28th International Symposium on Microarchitecture*, November 1995, pp. 327–337.

[135] J. F. Collard and D. M. Lavery, "Optimizations to prevent cache penalties for the Intel Itanium 2 Processor," in *Proceedings of the 2003 International Symposium on Code Generation and Optimization*, March 2003, pp. 105–114.

[136] D. Landskov, S. Davidson, B. Shriver, and P. W. Mallet, "Local microcode compaction techniques," *ACM Computing Surveys*, vol. 12, pp. 261–294, September 1980.

[137] R. E. Hank, "Machine independent register allocation for the IMPACT-I C compiler," M.S. thesis, Department of Electrical and Computer Engineering, University of Illinois at Urbana-Champaign, 1995.

[138] M. Tokoro, E. Tamura, and T. Takizuka, "Optimization of microprograms," *IEEE Transaction on Computers*, vol. C-30, pp. 491–504, July 1981.

[139] J. Ellis, *Bulldog: A Compiler for VLIW Architectures*. Cambridge, MA: The MIT Press, 1985.

[140] J. A. Fisher, "Global code generation for instruction-level parallelism: Trace scheduling-2," Hewlett-Packard Laboratory, Tech. Rep. HPL-93-43, June 1993.

[141] M. Schlansker, V. Kathail, and S. Anik, "Height reduction of control recurrences for ILP processors," in *Proceedings of the 27th International Symposium on Microarchitecture*, December 1994, pp. 40–51.

[142] M. Schlansker and V. Kathail, "Critical path reduction for scalar programs," in *Proceedings of the 28th International Symposium on Microarchitecture*, December 1995, pp. 57–69.

[143] K. Ebcioğlu, E. Altman, S. Sathaye, and M. Gschwind, "Optimizations and oracle parallelism with dynamic translation," in *Proceedings of the 29th Annual International Symposium on Microarchitecture*, November 1999, pp. 284–295.

[144] K. Pettis and R. C. Hansen, "Profile guided code positioning," in *Proceedings ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation*, June 1990, pp. 16–27.

[145] A. H. Hashemi, D. R. Kaeli, and B. Calder, "Efficient procedure mapping using cache line coloring," in *Proceedings of the ACM SIGPLAN 1997 Conference on Programming Language Design and Implementation*, 1997, pp. 171–182.

[146] P. H. Wang, H. Wang, R. M. Kling, K. Ramakrishnan, and J. P. Shen, "Register renaming and scheduling for dynamic execution of predicated code," in *Proceedings of the 7th International Symposium on High-Performance Computer Architecture*, January 2001, pp. 15–25.

[147] R. Barnes and W. Hwu, "Multipass pipelining," in *Proceedings of the Fourth Workshop on Explicitly Parallel Instruction Computer Architectures and Compiler Technology*, March 2005, http://rogue.colorado.edu/EPIC4/.

[148] G. Ottoni, R. Rangan, N. Vachharajani, and D. August, "Decoupled software pipelining: A promising technique to exploit thread level parallelism," in *Proceedings of the Fourth Workshop on Explicitly Parallel Instruction Computer Architectures and Compiler Technology*, March 2005, http://rogue.colorado.edu/EPIC4/.

[149] Y. Chou, B. Fahs, and S. Abraham, "Microarchitecture optimizations for exploiting memory-level parallelism," in *Proceedings of the 31st Annual International Symposium on Computer Architecture*, June 2004, pp. 76–87.

[150] B. C. Cheng and W. W. Hwu, "Modular interprocedural pointer analysis using access paths: Design, implementation, and evaluation," in *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, June 2000, pp. 57–68.

[151] B. C. Cheng, "Compile-time memory disambiguation for C programs," Ph.D. dissertation, Department of Computer Science, University of Illinois at Urbana-Champaign, 2000.

[152] R. Ghiya, D. Lavery, and D. Sehr, "On the importance of points-to analysis and other memory disambiguation methods for C programs," in *Proceedings of the ACM Symposium on Programming Language Design and Implementation*, June 2001, pp. 47–57.

[153] S. Triantafyllis, M. Vachharajani, and D. August, "Compiler optimization-space exploration," *Journal of Instruction-Level Parallelism*, vol. 7, February 2005, http://www.jilp.org/vol7/v7paper3.pdf.

[154] F. Mueller and D. B. Whalley, "Avoiding unconditional jumps by code replication," in *Proceedings of the ACM SIGPLAN 1992 Conference on Programming Language Design and Implementation*, June 1992, pp. 332–330.

[155] F. Mueller and D. B. Whalley, "Avoiding conditional branches by code replication," in *Proceedings of the ACM SIGPLAN 1995 Conference on Programming Language Design and Implementation*, June 1995, pp. 55–66.

[156] J. C. Gyllenhaal, "An efficient framework for performing execution-constraint-sensitive transformations that increase instruction-level parallelism," Ph.D. dissertation, University of Illinois at Urbana-Champaign, 1997.

[157] J. C. Gyllenhaal, B. R. Rau, and W. W. Hwu, "HMDES version 2.0 specification," IMPACT, University of Illinois at Urbana-Champaign, Tech. Rep. IMPACT-96-03, 1996.

[158] S.-Z. Ueng, "Template building for EPIC architectures," M.S. thesis, Department of Electrical and Computer Engineering, University of Illinois at Urbana-Champaign, 2004.

[159] D. M. Lavery, "Modulo scheduling for control-intensive general-purpose programs," Ph.D. dissertation, Department of Electrical and Computer Engineering, University of Illinois at Urbana-Champaign, 1997.

[160] B. R. Rau, M. S. Schlansker, and P. Tirumalai, "Code generation schemas for modulo scheduled do-loops and while-loops," Hewlett Packard Labs, Tech. Rep. HPL-92-47, April 1992.

[161] G. J. Chaitin, "Register allocation and spilling via graph coloring," in *Proceedings of the ACM SIGPLAN 82 Symp. on Compiler Construction*, June 1982, pp. 98–105.

[162] F. C. Chow and J. L. Hennessy, "The priority-based coloring approach to register allocation," *ACM Transactions on Programming Languages and Systems*, vol. 12, pp. 501–536, October 1990.

[163] P. Briggs, K. D. Cooper, and L. Torczon, "Rematerialization," in *Proceedings of the ACM SIGPLAN 1992 Conference on Programming Language Design and Implementation*, June 1992, pp. 311–321.

[164] GCC Project, "GNU Compiler Collection version 3.2," 2003, http://gcc.gnu.org/.

[165] Intel Corporation, "Intel C/C++ Compiler 8.1 for Linux," 2004, http://www.intel.com/software/products/compilers/clin/.

AUTHOR'S BIOGRAPHY

John Wollenburg Sias earned his Bachelor of Science in Computer Engineering (Highest Honors, University Honors) and Master of Science in Electrical Engineering degrees from the University of Illinois at Urbana-Champaign in 1997 and 1999, respectively. Since 1996 he has served as a research assistant in the Illinois Microarchitecture Project Utilizing Advanced Compiler Technology (IMPACT) Research Group, under Professor Wen-mei W. Hwu. (The IMPACT Research Group is affiliated with the Coordinated Science Laboratory and the Center for Reliable and High-Performance Computing.) John architected the re-development of the IMPACT Research Compiler to target the Itanium processor family and leads compiler development in the research domain and, under the Gelato Initiative's OpenIMPACT program, the public domain. He has also collaborated extensively with experts at Intel and Hewlett-Packard Corporations and worked four summers at the IBM Centre for Advanced Research in Toronto, Canada. While John's work has focused primarily on compiler technology for instruction-level parallelism in explicitly-parallel instruction computing systems, he has also participated in developing compiler techniques for nontraditional, ultra-efficient computer architectures.

John has enrolled in the Master of Divinity program at Concordia Theological Seminary, Fort Wayne, Indiana. The Lord willing, he will, upon completion of this program, be called and ordained as a pastor in the Lutheran Church—Missouri Synod.