Field-testing IMPACT EPIC research results in Itanium 2

John W. Sias Sain-zee Ueng Geoff A. Kent
Ian M. Steiner Erik M. Nystrom Wen-mei W. Hwu

Center for Reliable and High-Performance Computing
Department of Electrical and Computer Engineering
University of Illinois at Urbana-Champaign

{sias,ueng,gkent,isteiner,nystrom,hwu}@crhc.uiuc.edu

Abstract

Explicitly-Parallel Instruction Computing (EPIC) provides architectural features, including predication and explicit control speculation, intended to enhance the compiler's ability to expose instruction-level parallelism (ILP) in control-intensive programs. Aggressive structural transformations using these features, though described in the literature, have not yet been fully characterized in complete systems. Using the Intel Itanium 2 microprocessor, the SPECint2000 benchmarks and the IMPACT Compiler for IA-64, a research compiler competitive with the best commercial compilers on the platform, we provide an in situ *evaluation of code generated using aggressive, EPIC-enabled techniques in a reality-constrained microarchitecture. Our work shows a 1.13 average speedup (up to 1.50) due to these compilation techniques, relative to traditionally-optimized code at the same inlining and pointer analysis levels, and a 1.55 speedup (up to 2.30) relative to GNU GCC, a solid traditional compiler. Detailed results show that the structural compilation approach provides benefits far beyond a decrease in branch misprediction penalties and that it both positively and negatively impacts instruction cache performance. We also demonstrate the increasing significance of runtime effects, such as data cache and TLB, in determining end performance and the interaction of these effects with control speculation.*

1 Introduction

Most contemporary, mainstream microarchitectures have increased performance by sharply decreasing the clock period, sacrificing (non-pipeline) instruction-level parallelism and adding complex and potentially inefficient replay mechanisms to meet this goal (*e.g.* Pentium 4) [1]. These systems rely on essentially classical compiler technology; in this "hectic" model of parallelism, it is the microarchitecture that ensures in a highly dynamic manner that pipeline parallelism is effectively exploited.

Explicitly-parallel instruction computing (EPIC) systems such as Intel's IA-64, however, deliver performance in an entirely different way, which might be termed a "deliberate" model of parallelism. Here, 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 crunch through the compiler's plan-of-execution. The architecture provides a suite of features, including large register files, wide issue, predication and explicit control and data speculation, to enhance the compiler's ability to exploit instruction-level parallelism (ILP) in common programs. 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 is ultimately more efficient for a particular application set is beyond the scope of this paper; the importance of strong control-intensive, general-purpose application performance to the success of EPIC systems, however, is beyond dispute.

The compiler can approach the compilation of control-intensive codes for EPIC performance in a variety of ways. One may choose an "incremental"[1] approach that uses EPIC features to enhance traditional, global-scheduling-based schemes for ILP exploitation, incrementally enhancing the application of traditional compilation models, within the existing program control structure. Contemporary production compilers operate mostly within this model [3, 4]. 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 [5, 6, 7, 8]. This approach is more consistent with EPIC's research lineage. While the literature includes some real-machine evaluations of EPIC's features [9, 10], they are based on compilers taking primarily an incremental approach.

Other research-based evaluations [8] examined the structural interaction of predication and speculation tech-

[1] 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.

Proceedings of the 31st Annual International Symposium on Computer Architecture (ISCA'04)
1063-6897/04 $ 20.00 © 2004 IEEE

IEEE COMPUTER SOCIETY

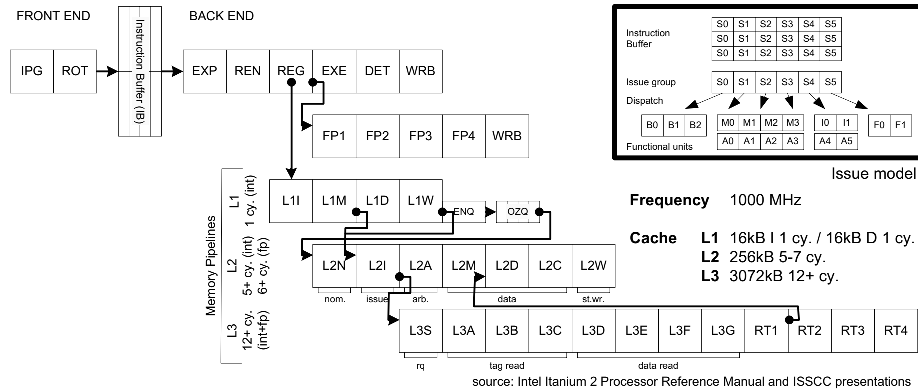


Figure 1: Intel Itanium 2 pipeline and processor configuration[2]

niques 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 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 SPEC92 and SPEC95 to SPEC2000 will show [11]). 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 [12], and a version of the IMPACT compiler that targets this machine [13, 14]. This paper provides a holistic and contemporary understanding of EPIC performance from the structural research 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 2.30 (average 1.55) over GCC and up to 1.50 (average 1.13) relative to IMPACT's classical optimization level. Excluding nondeterminisms such as data and instruction cache misses, as most simulation-based experiments [8] have done, IMPACT achieves an average speedup of 1.36 relative to the classically optimized baseline, a result comparable with past investigations. Starting from these results, we explain the increasingly important secondary effects that today need to be considered to generalize the benefit of structural EPIC transformations. This paper provides an in-depth first look at the real-machine performance that results from pairing IA-64 with one of today's most aggressive EPIC research compilers.

2 IA-64 and structural transforms

Our experiments characterize the effects of aggressive ILP transformations on SPECint2000 programs running on a 1GHz Itanium 2 processor with 3MB L3 cache. Configuration details are provided in Figure 1 and Table 1. For a compiler we use the OpenIMPACT framework [14], which affords a greater degree of flexibility and more aggressive utilization of EPIC features than would be available in commercial production compilers or GCC. Section 3 outlines the compiler's important features.

Except for SPEC ratios, obtained using SPEC 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 [15]. This combination allows over four hundred 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. This information has proven invaluable, not only in demonstrating performance benefits, but also in localizing performance problems.

The Itanium 2 is nominally a six-issue processor, with six general-purpose ALU (specialized as two load units,

1063-6897/04 $ 20.00 © 2004 IEEE

IEEE COMPUTER SOCIETY

two store units, and two integer units), two FPU, and three branch units, all fully pipelined. The processor is in-order and provides no register renaming. As indicated in Figure 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 [2]. Any ILP to be exploited in the microarchitecture must be included in the compiler's plan of execution. Up to six operations, as grouped by the compiler, are presented to execution units in each cycle.

Performance in control-intensive, general-purpose programs, therefore, depends heavily on the compiler's ILP-exposing transformations, which IA-64 supports with predication, explicit control speculation, data speculation, and modulo scheduling aids[16]. 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), the authors do observe many opportunities for data speculation. Aside from the obvious candidates (*eon* and *perlbmk*, in which pointer analysis is currently disabled) *gap* shows much promise. In *gap*, pointer analysis is unable to resolve critical spurious dependences in otherwise highly-parallel loops. 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 [17].

2.1 High-level results

Table 1 shows estimated SPECint2000 performance ratios (higher is better) for GNU and IMPACT compilers on a Linux system (implying 64-bit pointers, which affect data cache performance). All three versions of IMPACT-compiled code use interprocedural analysis, profile feedback (using SPEC's training inputs) and the same degree of cross-file procedure inlining. **O-NS** is a classically-optimized baseline that does not make use of predication or speculation. **ILP-NS** applies predication and ILP-formation techniques, but not support for control speculation of potentially-excepting instructions, achieving an average speedup of 1.10. **ILP-CS**, finally, adds control speculation, achieving a cumulative average speedup of 1.13.[2]

[2]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." GCC 3.2 is run with "-O3 -fomit-frame-pointer" and without profile feedback (as it is as yet largely unsupported). IMPACT pointer analysis and, consequently, modulo scheduling are disabled for *eon* and *perlbmk*, due to non-support of C++ and a scalability issue, presently being addressed. These results reflect 64-bit pointers, in contrast to most published Itanium 2 results.

Table 1: Estimated SPECint2000 performance ratios

Benchmark	GCC 3.2	O-NS	ILP-NS	ILP-CS
164.gzip	374	602	677	752
175.vpr	497	607	644	719
176.gcc	521	828	964	792
181.mcf	333	332	330	341
186.crafty	489	646	677	704
197.parser	410	520	523	541
252.eon	273	364	428	429
253.perlbmk	472	661	704	676
254.gap	375	558	573	599
255.vortex	550	843	1129	1264
256.bzip2	414	652	658	698
300.twolf	557	724	830	921
GEOMEAN	430	591	645	668

Hewlett-Packard zx6000: 2× 1GHz/3MB Itanium 2,
8GB RAM, linux 2.4.21-gspec (LP64 mode)

Key to results

O-NS	IMPACT code, classical optimization only, no control speculation
ILP-NS	IMPACT code, classical and ILP-enhancing optimizations, no control speculation
ILP-CS	IMPACT code, classical and ILP-enhancing optimizations, control speculation

As Table 1 indicates, IMPACT's performance far exceeds that of GCC on Itanium 2, even when only traditional optimizations are performed. With ILP transformations, IMPACT achieves an average speedup of 1.55 (maximum 2.30) relative to GCC. While GCC performs a very competent level of traditional optimizations, it is not equipped to deliver even minimal levels of ILP on IA-64. It performs little inlining, no interprocedural pointer analysis and no ILP-enhancing techniques. While we do not here show a comparison with Intel's production compiler for IA-64, *ecc* [4], because Intel has not to date released SPEC results for our configuration, we can state that IMPACT compiled code performance is comparable to or in excess of that we obtained using *ecc*[3] in all benchmarks except for *mcf*, in which Intel's compiler performs a highly effective data prefetching optimization not implemented in the IMPACT compiler, and *eon*, in which the disabling of IMPACT's pointer analysis and its current lack of data speculation support limit optimization potential.

When we examine previous, simulation-derived EPIC results, such as the performance of the nine SPECint92 and SPECint95 benchmarks in [8], we find a speedup of 1.17 for ILP techniques and 1.68 for ILP with speculation on the IMPACT EPIC simulator. The fact that this far exceeds our results on IA-64 is explained in three ways. First, past "clean" simulations did not model data and instruction cache stall cycles or other dynamic events. When we correct for this using measurements of these effects from performance monitoring counters, we find much better agreement with these past results. Figure 2 shows the relation

[3]version 7.1, -O3 -ipo, with profile guided optimization

1063-6897/04 $ 20.00 © 2004 IEEE
IEEE COMPUTER SOCIETY

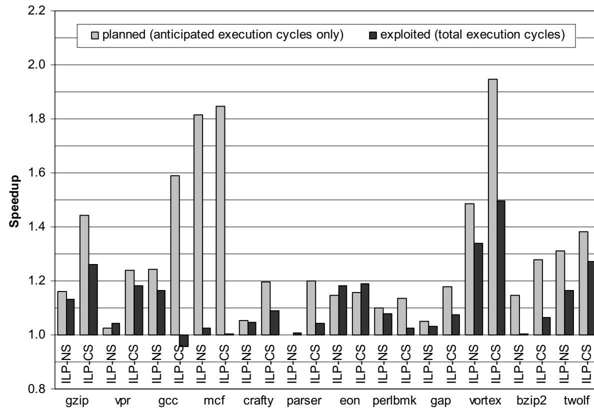


Figure 2: IMPACT SPECint2000 speedup: planned (subtracting stalls) and exploited

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 1. The "planned" speedup, on the other hand, measures the change in the number of execution cycles statically anticipable by the compiler.[4] Considering only these cycles, as in past simulations, IMPACT achieves an average speedup of 1.36, 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 1.21. Clearly, this is an aspect that needs to be addressed to strengthen the compiler's ability to plan for high EPIC performance.) Second, the SPECint2000 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 benchmarks of [8] benefited from data speculation, which IMPACT does not currently exploit on IA-64.

2.2 Impediments to ILP

Before examining specifics of the IMPACT compiler's behavior on IA-64, we should summarize the obstacles the compiler needs to clear to expose ILP in an EPIC system; these fall into four broad categories:

Control. By definition, in control-intensive programs, control operations are frequent. 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. 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.

False dependences. Memory accesses and subroutine calls can pose barriers to code motion, impeding both optimization and scheduling, if their dependences are resolved only conservatively. Alias and array dependence analysis aim to determine the true, minimum set of dependences needing to be drawn among these operations to preserve program correctness. IMPACT applies a sophisticated interprocedural pointer analysis algorithm [18] and Pugh's Omega Test [19] to reduce spurious dependences.

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 [1]. This allows them usually to reorder "mostly independent" operations, something the compiler cannot do statically without additional hardware support for data speculation [16, 20]. The IMPACT compiler does not currently make use of this feature on IA-64, although it holds promise.

Non-determinism. 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. Complicating the basic scheduling problem, when speculated, each off-path instance of these operations becomes a potential performance "landmine." Various software prefetching schemes and microarchitectural extensions have been proposed to help "smooth over" these events [21, 22]. Section 4.2 describes the interaction of these events with structural optimization.

2.3 Releasing parallelism from control

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. Compiler-based trace selection and EPIC features together allow for

[4]This measure includes the **unstalled** and the three **scoreboard** components of Figure 5; 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.

IEEE COMPUTER SOCIETY

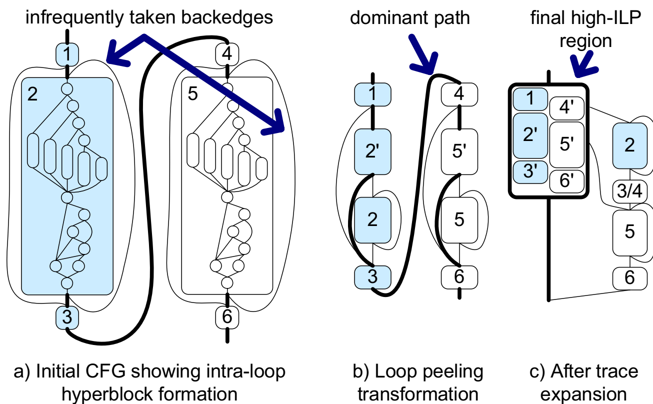


Figure 3: Exploiting ILP across *crafty* loops

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* [6]. This increases the scalability of trace formation, as it counters the 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, 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.

2.4 A motivating example from *crafty*

Crafty, a chess program, serves as one of the most control-intensive of the SPECint2000 benchmarks [11]. 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 SPECint2000 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.

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(a). Both loops contain substantial internal control flow, each loop has little inherent, intra-loop ILP due to serial data dependences, and each loop body typically executes exactly once.[5] 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 backedges, are merged to form a single scheduling region. Predication 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 SPECint2000 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 ILP transformations and benefits

The structural transformations of an ILP compiler like IMPACT must be both effective, producing regions of code suitable for highly instruction-parallel execution, and efficient, not unduly disrupting the work of other transformations or the execution of the program. While various individual ILP enhancement techniques are well-known [5, 6, 23, 24, 25, 26], their collective effect on contemporary programs is not well characterized. This is the aim of our experiments with IMPACT and IA-64.

In our experiments on the *six-issue* Itanium 2 microprocessor, the IMPACT compiler (**ILP-CS**) today schedules, on average, 2.63 useful (non-**nop**, non-predicate-squashed)

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

IEEE COMPUTER SOCIETY

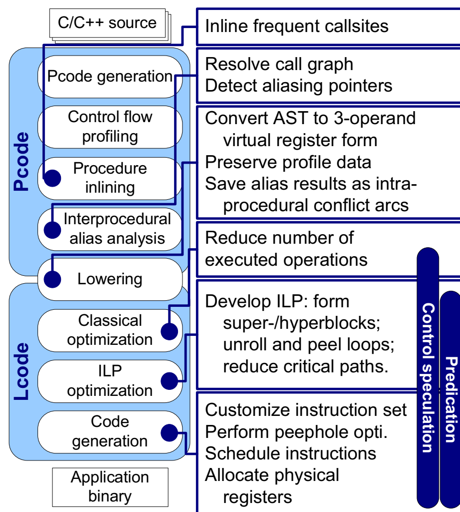


Figure 4: The IMPACT compiler for Itanium 2

operations per cycle[6] and causes the microarchitecture to sustain execution of 1.23 useful operations per cycle. (See Figure 6 for per-benchmark values.) This result, an increase from 2.00 planned / 1.10 achieved for **O-NS** code and 2.21 planned / 1.12 achieved for **ILP-NS**, is very competitive with good production compilers, and illustrates the ILP-enhancing character of IMPACT's transformations. Improvements in this result can come from two sources: first, increasingly aggressive ILP transformations, increasing the amount of planned parallelism; and, second, trying to close the gap between planned and exploited parallelism by mitigating stalls due to runtime events, including primarily data cache misses (*cf.* Figure 5).

Figure 4 shows the major phases comprising the IMPACT compiler. IMPACT's "structural" emphasis means that the transformation process first simplifies the control flow graph, forming large, stable execution regions, and then customizes and schedules code within these regions. This section highlights the important parts of this process and their participation in the experimental results that follow. One of the common features of all these phases is that specialization for ILP is "purchased" at the cost of code expansion, whether, for example, by performing procedure inlining or by forming region traces without side entrances to enhance scheduling. As interesting programs become increasingly larger and more control-bound, exposing a constant level of ILP requires more aggressive specialization. At the same time, this specialization must also operate more surgically, avoiding secondary costs like instruction cache thrashing. Our experiments are beginning to expose the importance of these traditionally second-order effects. We will consider these after a brief survey of the positive effects of IMPACT's transformations.

[6] In well-scheduled kernels, the full issue width of 6 is often filled; *gzip*, *gcc* and *vortex* all have average planned IPC over 3.0.

3.1 High-level analyses and transformations

C source code is converted to Pcode, IMPACT's high-level intermediate representation, which is then annotated with control flow profiling results from a training run of the program. Procedure inlining depends on profile information to expand callsites in priority order ($\textbf{priority} = \frac{\textbf{exec_weight}}{\sqrt{\textbf{callee_size}}}$) until the amount of touched code is expanded by a factor of 1.6 (an empirically determined value; because this transformation is performed at the high level, this is only approximate). This tended to provide enough inlining to achieve good levels of ILP while not unduly impacting instruction cache performance. Again using profile information, IMPACT converts selected indirect function calls to direct calls by inserting control flow and "specialized" direct calls to the most popular callee functions, which then may be inlined according to the normal method. This is important for programs, like *eon* and *gap*, that make extensive and often very biased use of indirect calls (in the C++ program *eon*, monomorphic virtual invocations). Procedure inlining sets the stage for subsequent, effective ILP optimizations, influencing performance outcomes by up to 20% in our experiments. Although it is typically considered part of the ILP exposing apparatus, we applied it to the IMPACT baseline as well as ILP compilation paths, to improve the comparability of the results.

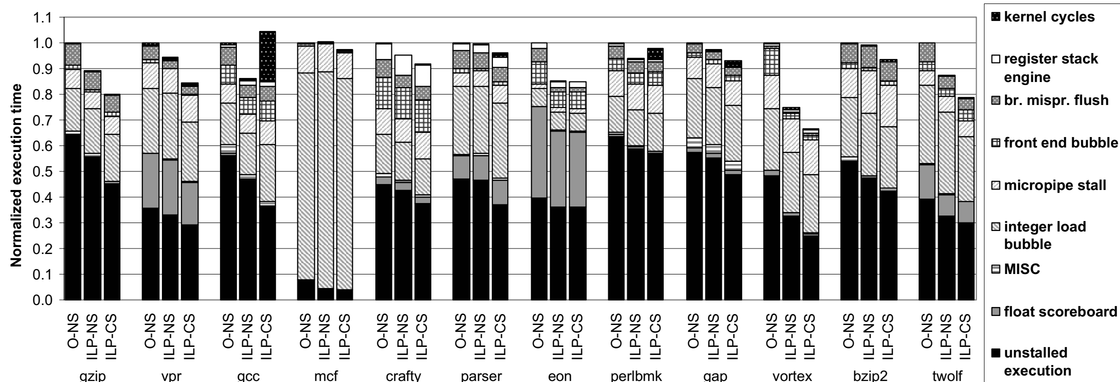
Interprocedural alias analysis, as discussed in Section 2.2, provides later stages with dependence arcs for loads, stores and subroutine calls [18]. It too has a substantial effect on output code quality. While no explicit ILP transformations take place in the high-level phase, it is responsible for setting the stage for subsequent optimizations.

3.2 Low-level analyses and transformations

The bulk of EPIC transformation, including predication and speculation, is performed in Lcode, a low-level representation. First, classical optimization, similar to that in GCC or any other compiler, is performed. This classically optimized code is scheduled and register allocated to produce the baseline for our experiments, referred to as **O-NS**, for "optimized, no speculation." Even this level of optimization handily outstrips the performance of GCC on IA-64, but because IMPACT does not apply global scheduling in the manner of production compilers [3], this code is usually slightly inferior to their compiled code.

Rather than applying global scheduling techniques at this point (as would an incremental approach), however, IMPACT transforms the program control structure into one more conducive to efficient optimization and subsequent execution. Frequently traversed traces are identified; those sharing substantial code and having compatible dependence heights and resource utilization are combined using predication into hyperblocks [6] and singleton traces are expanded into superblock regions [5]. To increase the amount

1063-6897/04 $ 20.00 © 2004 IEEE
IEEE COMPUTER SOCIETY



Category **MISC** includes **int scoreboard**, **misc. scoreboard** and **exception flush**.

Figure 5: IMPACT compiled-code cycle accounting, relative to **O-NS** baseline

of code eligible for these transformations, tail duplication (node splitting) is performed. This enables continuation of a single-entry trace region across a control flow merge point by replicating subsequent code. Loop peeling [8], as illustrated in Figure 3(b), can be viewed as an extension of tail duplication. While the complex heuristics controlling these transformations are beyond the scope of this paper, our branch removal rate (27% of dynamic branches are eliminated) indicates the aggressiveness of IMPACT's region-forming transformations. On average, IMPACT's superblock and hyperblock heuristics apply tail duplication to an extent that causes a 21% increase in static code size; peeling adds an additional 2%.

After region formation, additional code optimizations take place, including control and data height reduction, pre- and post-pass local instruction scheduling, and modulo scheduling-based software pipelining. In the **ILP-CS** mode, control speculation, both in the form of moving operations above side-exit branches and in the form of predicate promotion (weakening the predicates that guard the execution of operations), is performed as required to enable optimizations and freer scheduling. IMPACT's optimizations are also enhanced to use control speculation, *e.g.* in the form of speculative partial redundancy elimination, and predication, *e.g.* to sink non-exit-dominating, potentially-excepting operations out of loops.

3.3 Structural transformation for ILP

Let us now turn to the data to consider the performance effects of these transformations. Figure 5 shows a breakdown of execution cycles into nine categories, for each of three IMPACT compilations of each program.[7] The total height of each bar reflects the program execution time, normalized to the baseline level of optimization. The bottom three segments, **unstalled execution**, **float scoreboard** and **MISC** (**scoreboard** contributors only; **exception flush** is insignificant), those used to compute the "planned" speedup of Figure 2, are statically anticipable by the compiler. As the optimization level is increased, most of the performance gain derives from a reduction in these cycle counts, reflecting increased planned ILP. Creation of larger, single-entry scheduling regions allows extensive, low-cost instruction scheduling. Within these regions, control speculation frees operations from dependences imposed by branches and predicate defining instructions, increasing scheduling freedom. As shown in the peel-and-merge example of the previous section, additional gain comes from the overlapping of independent control constructs. On Itanium 2, these mechanisms, not removal of branch misprediction penalty, yield most of the benefit of region formation.

Dynamic effects, harder to anticipate or accommodate at compile-time, are less sensitive to IMPACT's transformations. Stalls on variable-latency events, such as data cache misses (**integer load bubble**), not scheduled by the compiler, are affected in various ways (as described further in Section 4.2). Usually, and on average, positive and negative effects "cancel out," rendering ILP transformations relatively neutral with respect to these events. There are, however, more consistent effects on two particular dynamic events. Branch misprediction stall cycles (**br. mispr. flush**) are reduced according to the number of branches removed by if-conversion (Section 3.5). Abrupt increases in kernel cycles (*e.g.* in *gcc*) are due to wild loads, as will be discussed in Section 4.3.

3.4 Effects on dynamic instruction count and cache performance

Unlike the conceptual IMPACT EPIC machine of [8], Itanium 2 is not simply a general six-issue processor; the IA-64 architecture treats instructions in bundles of three

[7] **micropipe** refers to a collection of microarchitectural stalls, here related to the memory subsystem. The increase with optimization seen in this category for *bzip*, for example, reflects spurious store-to-load forwarding detections that become more costly as loops are tightened by optimization. See [2] for Itanium 2 performance monitoring information.

1063-6897/04 $ 20.00 © 2004 IEEE
IEEE COMPUTER SOCIETY

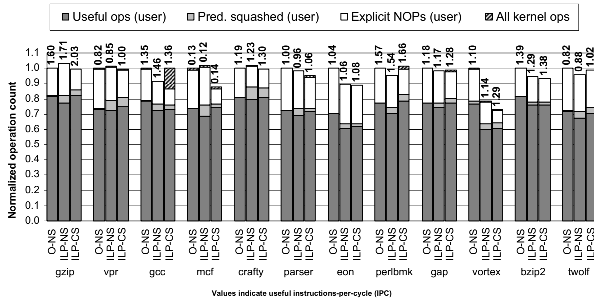


Figure 6: Operation accounting and IPC

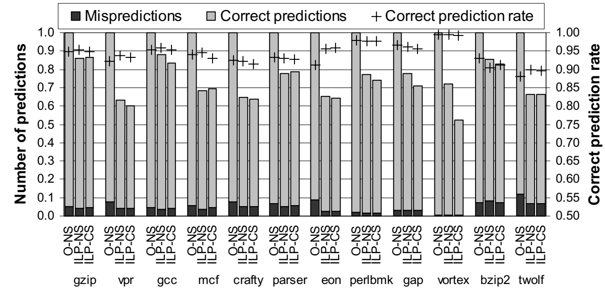


Figure 7: Effects on branch prediction

and sometimes requires encoding of **nop** instructions when slots cannot be filled with independent operations. This elicits the curious property that, because code-replicating transformations allow specialization of replicated paths, as long as these transformations allow more parallel instruction issue, instruction cache performance and fetch efficiency may actually be increased. This phenomenon is due to the use of fewer **nop** instructions in better-scheduled code. This effect can be observed (indirectly) in Figure 6, in which the number of **nop**s retired is almost universally decreased in ILP-optimized code, and in Figure 5, in which ILP optimization typically reduces instruction cache miss cycles (**front end bubble**), averaging a 15% reduction (more sequential fetch may also contribute to this improvement). In short, increasing specialization makes each access contain more useful operations (reducing the number of L1I cache line fetches by 10% on average), but can increase total footprint if multiple versions need to be resident. Excessive code replication can overwhelm the instruction cache; we revisit this phenomenon in Section 4.

While control speculation causes operations to execute more frequently than in the original program, Figure 6 shows that our speculation benefits reflect only a fairly selective speculation of operations (a pronounced rise in executed operations is not observed in the speculation-enabled, **ILP-CS** version). 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" ops increases between **ILP-NS** and **ILP-CS** because of operation speculation, since in this context "useful" means "non-**nop**, pred=1" operations. Careful use of profile data and selective region formation allow few excess operations to be fetched or executed.

3.5 Effects on branches and prediction

Figure 7 shows the change in the number of branches and mispredicted branches between baseline and ILP-optimized code, as well as in the branch prediction rate. In our experiments, region formation (including code-expanding transformations) reduces the number of branches by 27%, on average, and misprediction stall cycles by an average of 22%. 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.

A previous study [9] on the Itanium (1) processor, using Intel's production compiler, found only a 7% reduction in the number of executed branches at what were considered "reasonable" levels of aggressiveness in forming predicated regions. As the results of [9] 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 limits 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.

4 Side-effects of EPIC transforms

At the time most EPIC techniques were developed, industry standard benchmarks had, or could easily be transformed to have, small, highly optimized kernels which dominated program execution time. Effects on the rest of the program, no matter how egregious, were not a primary concern. This is true to a great extent even in the SPEC95 benchmarks used in [8]. Today, in larger and increasingly control-intensive programs, however, these once-secondary effects can emerge as primary limiters on performance gain. An effective EPIC compiler needs to manage these costs without hobbling crucial transformations.

4.1 Effects of code-expanding transforms

Even with highly inclusive hyperblock formation heuristics, code growth in the form of tail duplication is not al-

1063-6897/04 $ 20.00 © 2004 IEEE
IEEE COMPUTER SOCIETY

ways completely avoidable, as including infrequent or incongruous paths into hyperblock regions can degrade performance. Blocks along these paths must be excluded from regions, and the subsequent copying necessary to render the regions from which they were excluded single-entry creates code bloat. Likewise, code-expanding transformations such as loop peeling, as we have seen, are necessary facilitators of region formation. This is one of the fundamental tensions of EPIC compilation: the specialization required for performance entails code expansion, which may cause deleterious effects of its own.

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.

As a representative case, we note *crafty*, a benchmark with a large instruction cache footprint. Even at baseline levels of optimization, it spends much time waiting on the instruction cache (**front end bubble** in Figure 5). While IMPACT ordinarily improves instruction cache behavior by improving the efficiency of instruction fetch (in *crafty* L1I accesses are decreased 8.7%), the code bloat involved in transforming *crafty* instead causes these fewer accesses to miss more often, in the end increasing I-cache stalls by 5%. Using hardware performance monitoring samples and code annotations by the compiler, this increase can be traced to lukewarm code copies created in formation of otherwise desirable customized regions. Tail duplicated code accounted for 4.4% of L1I misses and 6.4% of expensive L2I misses. Residual loops (those having had iterations peeled out of them, as in the example of Figure 3) accounted for an additional 2.4% of L1I and L2I misses. A similar phenomenon occurs in *twolf* (14.4% L1I access rate reduction), where a loop is peeled and the remainder loop, which is itself lukewarm, is then specialized, creating two new, lukewarm regions. Here, I-cache stall time increases 35%. In an unconstrained environment, these would all in fact be 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 both *crafty* and *twolf* benefited in the net from aggressive transformation (achieving speedups of 1.09 and 1.38, respectively), their gains would have been larger apart from these code bloat effects. In the aggregate, however, it is important to note that IMPACT's I-cache effects are *positive*, delivering a 15% decrease in stalls. Thus there is great potential for a better means of regulating these effects.

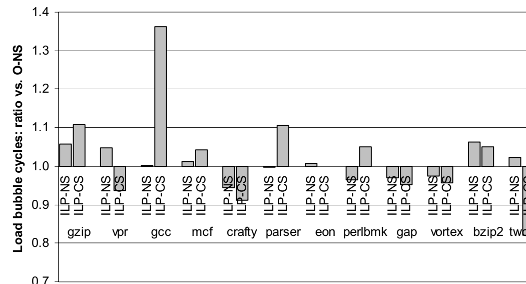


Figure 8: Effects on data cache stall cycles

4.2 Off-path speculative execution

Speculating operations to reduce dependence height or allow removal of program control apparatus causes these operations to execute more frequently. Executions not in the original program may incur unexpected latencies or additional exception handling overhead. Aggressive control speculation of loads (as variable latency operations) and their consumers may therefore cause an increase in the number of cycles spent stalled waiting on memory.

Figure 8 shows the variation in data cache stall cycles with optimization (relative to **O-NS**). Changes in **ILP-NS** are related to the aforementioned scheduling phenomena. Where increases occur between **ILP-NS** and **ILP-CS**, control speculation, often in the form of predicate promotion (the weakening of a predicate guarding an instruction), has allowed loads that miss in cache (and their consumers) to execute more frequently, generating stalls not observed in the original program. Where the opposite is true, loads freed from control dependence have been scheduled farther away from their consumers, accommodating some miss latency. We used performance monitor sampling to identify the speculative loads most influential in causing these increases; even though these extra load executions (in many cases, due to predicate promotion) occasionally generate extra data cache miss stalls, the benefit to schedule height achieved by their control speculation usually outweighs the cost. The data cache effect of off-path speculative loads was predicted in [8], although it is less pronounced in this experimental context, probably due to more selective constraints on instruction speculation.

4.3 Control speculation model

Several schemata have been proposed to support explicit control speculation of potentially-excepting instructions (PEI), each of which has its own set of benefits and costs. IA-64 provides for two models: *sentinel speculation with*

IEEE COMPUTER SOCIETY

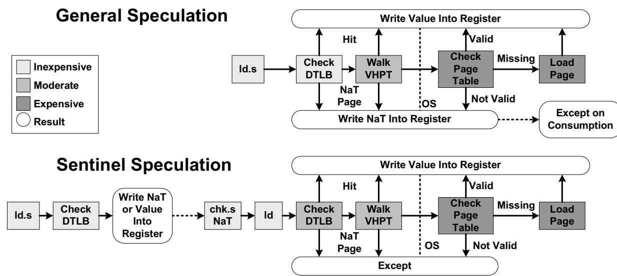


Figure 9: General and sentinel speculation models

explicit recovery blocks and *general speculation*; it is the prerogative of the operating system developer to support either or both of these models [16]. As the IMPACT compiler on IA-64 currently supports only the general speculation model, we modified the Linux kernel to support that schema. This patch is available from the authors.

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

Figure 9 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.

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, but do not update entries to "cache" their results, and can thus be very expensive.[8] This phenomenon is evident in four benchmarks (*gcc* in a prominent way, causing it to spend 20% of its execution time chasing spurious page faults, and less so in *parser*, *perlbmk* and *gap*), as indicated by the amount of kernel time incurred in these benchmarks in the control-speculative compiled code (**ILP-CS**) in Figure 5. Ongoing work suggests pointer analysis-based heuristics may help to identify and avoid speculating these dangerous, improving the economy of the general speculation model.

4.4 Register utilization

Exploitation of ILP by overlapping independent strands of computation requires allocation of many register names, even given predicate-aware dataflow and register allocation techniques [27, 28] that reduce the number of live range conflicts in predicated code. 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 activity (**register stack engine** in Figure 5).

4.5 Finer-grain results

Linux kernel support and the *Pfmon* performance monitoring tool allow binning of sampled events by instruction address. Using this facility, we can approximate per-function performance comparisons between two versions of compiled code. We used this capability 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 10 shows a comparison of **O-NS** code to **ILP-NS** code (a) and **O-NS** code to **ILP-CS** code (b) for the benchmark *vortex*. The horizontal space taken by a function is its contribution to **O-NS** execution time; the height of each is the ratio of ILP execution time to **O-NS** execution time. The arrow on the left indicates the total benchmark runtime relative to **O-NS**. The three prominent functions with little benefit, `chunk_free()`, `chunk_alloc()`, and `memcpy()`, are provided from gcc-compiled system libraries. The substantial contribution of these currently unoptimizable functions motivates library and cross-module compilation using IMPACT. Other functions in general show improvement from ILP formation techniques and from control speculation, and this holds true for the most part across the suite. In order for these comparisons between compilation versions to make sense, procedure contents must be consistent across the versions; this

[8] Common NULL dereferences are handled using a special, architected NaT page at address 0; these typically execute with only a 2-cycle penalty in either model.

1063-6897/04 $ 20.00 © 2004 IEEE

IEEE COMPUTER SOCIETY

(a) ILP-NS execution time relative to O-NS

(b) ILP-CS execution time relative to O-NS

Figure 10: Function-level execution time, *255.vortex*

was the primary reason for allowing the **O-NS** baseline the same inlining aggressiveness as the **ILP** versions.

4.6 Profile variation

A real concern for EPIC compilation effectiveness is the provision of correct profile data to the compiler. This data is used to guide all important ILP formation phases, including procedure inlining, region formation, block layout and instruction scheduling. SPEC practices model some degree of variation, as benchmarks are "trained" with inputs different from those used for performance evaluation. A difference in behavior between these two input sets, a "profile variation," can result in reduced performance gain. To examine the magnitude of this effect for ILP transformations in SPECint2000, we compiled the benchmarks using the reference inputs for training, for comparison with the ordinary runs. We identified three measurable differences: *crafty*'s performance improved by 5%, *perlbmk*'s by 10%, and *gap*'s by 3%. These benchmarks are quite sensitive to inlining decisions, and *crafty* and *perlbmk* have large instruction cache footprints. Since ensuring representative training inputs for "real" programs could be considered to be much more difficult than for a carefully constructed set of benchmarks, this poses important issues for EPIC compilation. We are therefore exploring means of improving transformation stability across profile changes.

5 Related work

August *et al.* presented a hypothetical EPIC system and expounded on the constructive collaboration between if-conversion and speculation techniques [8]. We show 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 [29]. 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.* 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 only changes in how, when, and to what degree existing optimizations are applied [10]. Their work pointed out interactions similar to those indicated here, within a very different compiler infrastructure. Choi *et al.* performed a focused evaluation of if-conversion on the Itanium processor, in the context of a more conservative production compiler [9]. Our relation to this work was discussed in Section 3.5.

6 Conclusions

We have demonstrated how IMPACT's "structural" approach to compilation for Itanium 2 provides a 1.13 average speedup (including single-benchmark speedups of up to 1.50) relative to traditional optimization of code at the same inlining and interprocedural analysis levels and a 1.55 speedup relative to GCC. We show that the benefit of these techniques far exceeds the removal of branch misprediction and that their code size expansion cost is usually performance-justified. We observe significant improvements in instruction fetch efficiency 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 quantitative understanding of potential risks: in a few cases, excessive replication led to deleterious I-cache effects; elsewhere, control speculation of "wild loads" exacted a heavy toll. 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, we point to the increasing significance of runtime effects, such as data cache and DTLB stalls, in determining end application performance. This calls for better microarchitectural management of these events or new research that puts them within the compiler's understanding.

IEEE COMPUTER SOCIETY

Acknowledgments

The authors extend their thanks to Intel and Hewlett-Packard for their generous support of this work, including software and hardware contributions; to Stephane Eranian (HP Labs) and Allan Knies (Intel) for performance measurement tools; to the other present and former members of the IMPACT Research Group and Robert Kidd of OpenIMPACT/UIUC for their compiler work; to Rick Hank (HP), Wei Li (Intel), David August (Princeton) and Sanjay Patel (UIUC) for their valuable pre-publication feedback; and, finally, to the reviewers for their many helpful comments. This work was partially supported by MARCO FCRP GSRC and C2S2 research centers.

References

[1] D. Carmean, "The Pentium 4 processor." Hot Chips 13, Stanford University, Palo Alto, CA, August 2001.

[2] Intel Corporation, *Intel Itanium 2 Processor Reference Manual for Software Development, Document Number 251110-001*, June 2002.

[3] 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.

[4] J. Bharadwaj, W. Chen, W. Chuang, G. Hoflehner, K. Menezes, K. Muthukumar, and J. Pierce, "The Intel IA-64 compiler code generator," *IEEE Micro*, pp. 44–53, October 2000.

[5] 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.

[6] 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*, pp. 45–54, December 1992.

[7] 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.

[8] 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*, pp. 227–237, June 1998.

[9] 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*, pp. 182–191, December 2001.

[10] 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*, pp. 204–215, 2003.

[11] Standard Performance Evaluation Corporation, "SPEC CINT2000 benchmarks." http://www.spec.org/cpu2000/CINT2000.

[12] C. McNairy and D. Soltis, "Itanium 2 processor microarchitecture," *IEEE MICRO*, vol. 23, pp. 44–55, March 2003.

[13] 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.

[14] UIUC OpenIMPACT Effort, "The OpenIMPACT IA-64 Compiler." http://gelato.uiuc.edu/.

[15] S. Eranian, "Perfmon: linux performance monitoring for ia64." Downloadable software with documentation, http://www.hpl.hp.com/research/linux/perfmon/, 2003.

[16] Intel Corporation, *Intel IA-64 Architecture Software Developer's Manual Volume 1: Application Architecture, Document Number 245317-003*, December 2001.

[17] 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*, pp. 289–299, 2003.

[18] B. C. Cheng and W. W. Hwu, "Modular interprocedural pointer analysis using access paths: Design, implementation, and evaluation," in *Proceedings of the ACM SIGPLAN '00 Conference on Programming Language Design and Implementation*, pp. 57–68, June 2000.

[19] W. Pugh and D. Wonnacott, "Eliminating false data dependences using the Omega Test," in *Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation*, pp. 140–151, June 1992.

[20] 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*, pp. 183–193, October 1994.

[21] 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*, pp. 167–176, February 2002.

[22] 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 *To appear in: Proceedings of 36th Annual International Symposium on Microarchitecture*, December 2003.

[23] J. A. Fisher, "Trace scheduling: A technique for global microcode compaction," *IEEE Transactions on Computers*, vol. C-30, pp. 478–490, July 1981.

[24] 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*, pp. 318–328, June 1988.

[25] 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.

[26] S. A. Mahlke, W. Y. Chen, W. W. Hwu, B. R. Rau, and M. S. Schlansker, "Sentinel scheduling for VLIW and supercalar processors," in *Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 238–247, October 1992.

[27] 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*, pp. 112–123, December 2000.

[28] 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*, pp. 114–125, December 1996.

[29] V. Kathail, M. S. Schlansker, and B. R. Rau, "HPL PlayDoh architecture specification: Version 1.0," Tech. Rep. HPL-93-80, Hewlett-Packard Laboratories, Palo Alto, CA, February 1994.