MEMORY DISAMBIGUATION TO FACILITATE INSTRUCTION-LEVEL PARALLELISM COMPILATION

BY

DAVID MARK GALLAGHER

B.S., United States Air Force Academy, 1978
M.S., Air Force Institute of Technology, 1987

THESIS

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, 1995

Urbana, Illinois

MEMORY DISAMBIGUATION TO FACILITATE
INSTRUCTION-LEVEL PARALLELISM COMPILATION

David Mark Gallagher, Ph.D.
Department of Electrical and Computer Engineering
University of Illinois at Urbana-Champaign, 1995
Wen-mei W. Hwu, Advisor

To expose sufficient instruction-level parallelism (ILP) to make effective use of wide-issue superscalar and VLIW processor resources, the compiler must perform aggressive low-level code optimization and scheduling. However, ambiguous memory dependences can significantly limit the compiler's ability to expose ILP. To overcome the problem of ambiguous memory dependences, optimizing compilers perform memory disambiguation.

Both dynamic and static approaches to memory disambiguation have been proposed. Dynamic memory disambiguation approaches resolve the dependence ambiguity at run-time. Compiler transformations are performed which provide alternate paths of control to be followed based upon the results of this run-time ambiguity check. In contrast, static memory disambiguation attempts to resolve ambiguities during compilation. Compiler transformations can be performed based upon the results of this disambiguation, with no run-time checking required.

This dissertation investigates the application of both dynamic and static memory disambiguation approaches to support low-level optimization and scheduling. A dynamic approach, the *memory conflict buffer*, originally proposed by Chen [1], is analyzed across a large suite of integer and floating-point benchmarks. A new static approach, termed *sync arcs*, involving the passing of explicit dependence arcs from the source-level code down to the low-level code, is proposed and evaluated. This investigation of both dynamic and static memory disambiguation allows a quantitative analysis of the tradeoffs between the two approaches.

DEDICATION

To my wife, Kathy, and my children, Jonathan and Daniel.

Thank you for your love, support and longsuffering!

ACKNOWLEDGMENTS

First and foremost, I would like to thank God for the opportunity, ability, and strength to complete a doctoral program. To Him be all the glory!

I would like to thank my advisor, Professor Wen-mei W. Hwu, for his insight and guidance throughout my studies. Not only was it an honor to work with someone of his caliber, but it was also a pleasure. He truly cares about the needs of his students.

This research would not have been possible without the support of the IMPACT research group. Their efforts provided a unique compilation environment in which to conduct my research. The group also provided a very enjoyable work atmosphere. Members of the group were always willing to provide any help required - from research discussions to practice talks to listening to my frustrations. I deeply appreciate the willingness of Scott Mahlke and Rick Hank to answer my innumerable questions about Lcode and to act as a sounding board for my ideas. Grant Haab spent a great deal of time introducing me to the Pcode environment; he was also responsible for the Pcode data dependence which laid the foundation for much of my work. John Gyllenhaal was responsible for much of the simulation environment used in this thesis. William Chen provided me with my first introduction to IMPACT and to dynamic memory disambiguation. Many thanks to Dan Lavery, Ben Sander, Wayne Dugal, Cheng-Hsueh Hsieh, Derek Cho, and others who worked hard on the IMPACT X86 project.

Thanks to Bob Rau, Mike Schlansker, Vinod Kathail, and Sadun Anik of HP Laboratory for their valuable discussions about static memory disambiguation. Their technical ability and insight across the entire spectrum of computer architecture are remarkable.

I would like to thank my parents, Gilbert and Billye, for their continued love and encouragement. They provided a firm foundation for me both spiritually and emotionally as a child; as an adult, they continue to act as a source of strength and stability in my life. I also thank my brother, Steven, for his love and patience with his little brother.

Finally, I must thank my wife, Kathy, and my children, Jonathan and Daniel, for their love and support during this difficult time in graduate school. They are the ones who truly sacrificed to make my graduate studies possible. Kathy single-handedly maintained our household, chasing the boys from one activity to another, so that I could spend time on my research. Jonathan and Daniel have been very understanding during the many times Dad couldn't be there for a school or sports activity. Thanks, and I plan to do better in the future!

TABLE OF CONTENTS

LIST OF TABLES

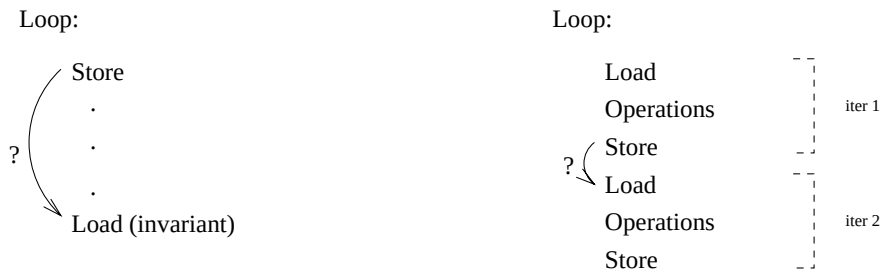
LIST OF FIGURES

CHAPTER 1

INTRODUCTION

Superscalar and VLIW processors attempt to achieve high performance by exploiting available instruction-level parallelism (ILP). The compiler is responsible for transforming the original program to expose sufficient ILP to keep the processor's functional units busy. This task of exposing parallelism requires aggressive low-level code optimization and scheduling.

A major impediment to exploiting ILP is *ambiguous memory dependences*. When two memory instructions (e.g., a load and a store) may possibly reference the same memory location, the two instructions have an ambiguous memory dependence between them. As a result of this dependence, the compiler must ensure that the memory operations are executed in the original program order. Any code transformation that would alter the order of execution is prevented. Figure 1.1 shows two examples of how ILP compilation is hindered by ambiguous memory references. In Figure 1.1(a), the load address is assumed to be loop invariant (it references the same address during all iterations of the loop). However, loop invariant code removal cannot be performed to move the load out of the loop unless it can be determined the store instruction never writes to the same memory location as the load. The ambiguous memory dependence thus inhibits an important code optimization. In Figure 1.1(b), a simple loop, assumed to consist of a load instruction, several arithmetic instructions, and a store instruction, has been unrolled in an attempt to expose greater ILP to the scheduler. Again, if it cannot be determined that the store in the first iteration always references a different memory location than the load in second iteration, the two iterations cannot be overlapped and no additional ILP is achieved.



(a) Loop invariant code removal example (b) Unrolling and scheduling example

Figure 1.1 Importance of Memory Disambiguation.

To overcome the problem of ambiguous memory dependences, optimizing compilers perform *memory disambiguation*, the process of determining whether two memory instructions might ever access the same location. Techniques for performing memory disambiguation generally are classified as either *dynamic* or *static*. Dynamic memory disambiguation determines at run-time whether two memory instructions ever reference the same location. To facilitate optimization or scheduling, the compiler provides different execution paths for the code depending upon whether the instructions are independent; at run-time, the dynamic memory disambiguation will determine which execution path is followed. In contrast, static memory disambiguation attempts to determine at compile-time the correct dependence relationship between memory instructions, using information available within the program's source code. If static memory disambiguation is successful in proving two memory instructions are independent, the compiler is able to perform optimization/scheduling at compile-time, and no run-time checking is required to ensure correct execution.

The potential benefit of dynamic and static memory disambiguation applied to low-level code optimization and scheduling has not been well-understood. Most existing dynamic memory disambiguation approaches are best suited for narrow-issue processors, and the benefit of

dynamic approaches for wide-issue superscalar or VLIW processors has not previously been demonstrated. Static memory disambiguation is most frequently applied to source-level code transformations, and its potential benefit for low-level code transformations has also not been demonstrated. Ideas for improved static disambiguation have been postulated, but few have actually been implemented in a working superscalar/VLIW compiler.

This dissertation examines both dynamic and static memory disambiguation approaches within the context of the IMPACT compiler project. Dynamic and static approaches have been implemented within the IMPACT compiler, targeted toward facilitating low-level code optimization and scheduling. Through detailed simulation, a quantitative analysis of both techniques is performed to better understand the merits of and tradeoffs between dynamic and static disambiguation.

1.1 Contributions

The four major contributions of this dissertation are discussed below.

- A dynamic memory disambiguation approach, the memory conflict buffer, is examined and developed. The memory conflict buffer is shown to be an effective means of overcoming the problem of ambiguous memory dependences, particularly for applications for which static analysis is not available. Contributions specific to this thesis include a new hardware design, development of an effective simulation environment, full integration into the IMPACT compiler, and a detailed quantitative evaluation of the benefit of the memory conflict buffer for ILP processors.

- The sync arc technique proposed in this thesis provides an effective framework for providing source-level dependence information for use by low-level optimization and scheduling. The technique is described in detail, defining the type of information to be carried by the sync arc, how the information is maintained through aggressive code transformations, and how the dependence information is used by low-level transformations. A quantitative analysis of the effectiveness of sync arcs demonstrates their potential benefit.

- The source-level dependence analysis required to support sync arcs is studied. The challenges for dependence analysis posed by the C language are discussed. In particular, the need for interprocedural analysis of C programs to support effective memory disambiguation for low-level code, and the required granularity of this analysis, is quantitatively investigated.

- The tradeoffs involved in selecting a static or dynamic memory disambiguation approach are explored. This analysis is unique in that an example of each approach has been implemented within a single compiler environment, enabling a fair comparison of the relative merits. Both approaches are shown to provide good memory disambiguation and to have applicability in different problem domains.

1.2 Overview

This dissertation is composed of eight chapters. Chapter 2 presents an overview of the organization of the IMPACT compiler. All compiler techniques discussed in this thesis are implemented within the framework of the IMPACT compiler. The simulation methodology employed in the thesis is also described. Chapter 3 discusses two approaches to deal with

ambiguous memory dependences: dynamic memory disambiguation and static memory disambiguation. The two approaches are reviewed and the tradeoffs between them are discussed.

A general technique for dynamic memory disambiguation, the memory conflict buffer, is presented in Chapter 4. This technique, which combines both hardware and compiler support, allows memory operations to be reordered during low-level code scheduling despite the presence of ambiguous memory dependences. The hardware support is responsible for detecting when truly dependent memory operations have been reordered. In the event this occurs, the compiler provides code to correct program execution.

Chapter 5 introduces sync arcs, a technique for maintaining explicit dependence information within the intermediate code. Static memory disambiguation is used to extract this dependence information from source-level code and to generate the sync arcs. A detailed discussion of how the sync arcs are preserved through and used by code transformations is presented.

Chapter 6 discusses the C dependence analysis used to provide the static memory disambiguation required for sync arcs. The interprocedural alias and side-effect analysis that supports this analysis is also presented. The experimental results using this dependence analysis and sync arcs are then presented in Chapter 7. A quantitative analysis of the benefit of improved memory disambiguation is given. This is followed by a comparative analysis of the relative benefit of the dynamic and static disambiguation approaches presented in this dissertation. Finally, Chapter 8 presents conclusions and suggests directions for future research.

CHAPTER 2

COMPILATION AND SIMULATION ENVIRONMENT

The compiler techniques necessary to investigate dynamic and static memory disambiguation approaches for this thesis are implemented within the framework of the IMPACT compiler project. The IMPACT compiler is a retargetable, optimizing C compiler being developed at the University of Illinois to investigate architectural and compilation techniques to support ILP processors. A block diagram of the IMPACT compiler is presented in Figure 2.1. The compiler accepts source code written in C, as well as Fortran code translated using the *f2c* translation tool [2]. The compiler can be divided into three distinct sections, each based upon a different intermediate representation (IR).

The highest level IR, *Pcode*, is a parallel C code representation with loop constructs intact. At the Pcode level, source-level techniques such as memory dependence analysis [3], loop-level transformations [4], and memory system optimizations [5], [6] are performed. Pcode is further described in Section 2.1. The middle-level IR is referred to as *Hcode*. In Hcode, the control structure of the code has been flattened into a basic block structure with simple *if-then-else* and *go-to* control flow constructs, but expressions are still maintained hierarchically. During this phase of compilation, basic-block-level profiling, as well as profile-guided code layout and function inline expansion [7], [8], [9], are performed.

The lowest level of IR in the IMPACT compiler is referred to as *Lcode*. Lcode is a generalized register transfer language similar in structure to most load/store processor assembly instruction sets. The majority of ILP code transformations within the IMPACT compiler are

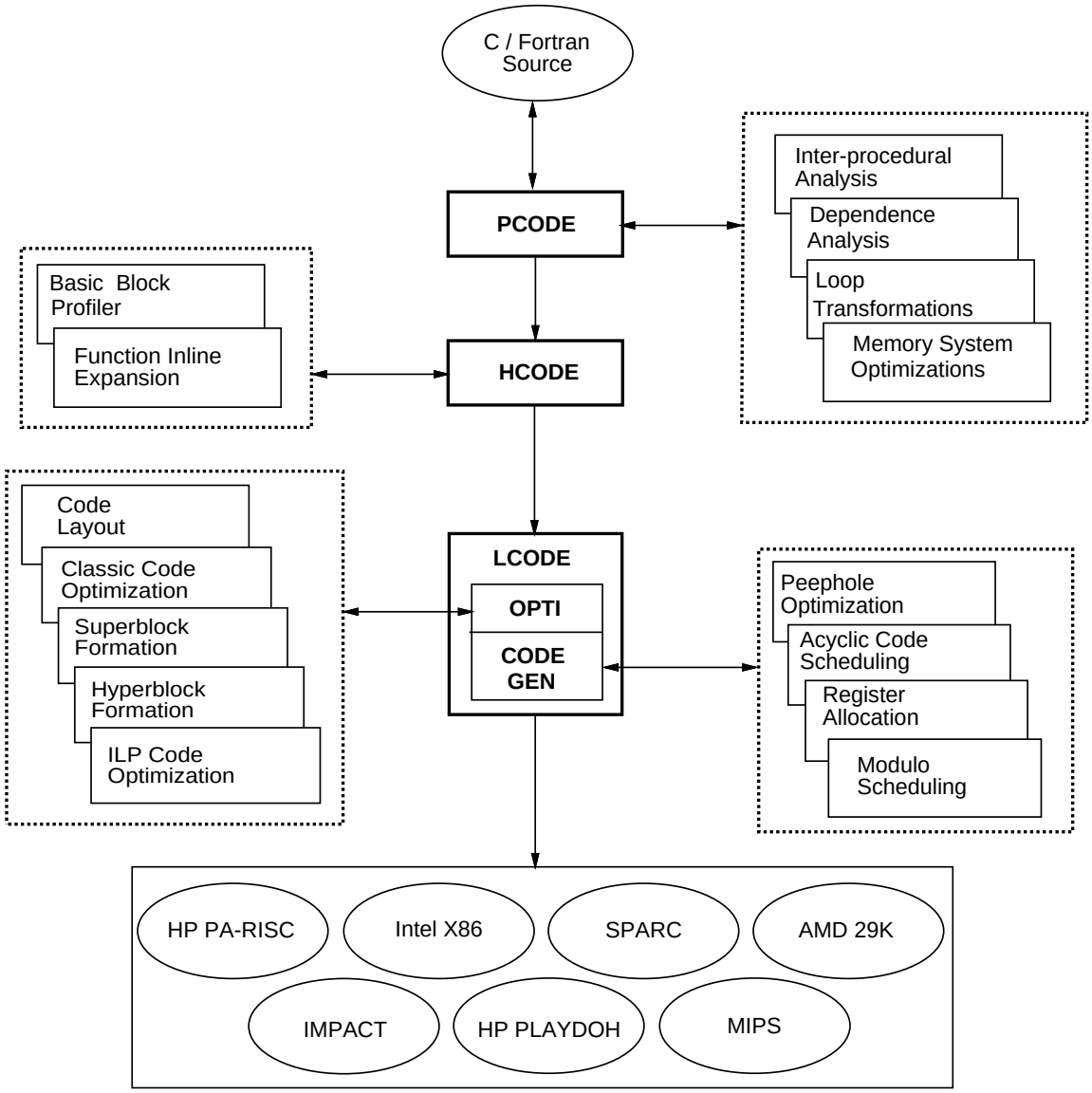


Figure 2.1 The IMPACT Compiler.

performed at the Lcode level. Section 2.2 details these code transformations. A detailed machine description database, *Mdes*, for each target architecture is available for use by all Lcode compilation modules [10].

Seven architectures are currently supported by the IMPACT compiler. These include the AMD 29K [11], MIPS R3000 [12], SPARC [13], HP PA-RISC, [1] and Intel X86 [14], [15]. The other two supported architectures, IMPACT and HP Playdoh [16], are experimental ILP architectures, which provide a framework for compiler and architectural research. The IMPACT architecture models a generic superscalar processor which executes the Lcode instruction set. After machine specific annotation of the Lcode, the IMPACT code generator can produce code for extended versions of the HP PA-RISC (IMPACT-HPPA) and the SPARC (IMPACT-SPARC) architectures. For this thesis, all experiments are based upon the IMPACT-HPPA architecture.

The remainder of this chapter details portions of the IMPACT compiler project especially important to this thesis. Sections 2.1 and 2.2 discuss the Pcode and Lcode levels of compilation. The superblock technique, which is foundational to much of IMPACT's ILP compilation, is presented in Section 2.3. Finally, the simulation environment used in this thesis is presented in Section 2.4.

2.1 Pcode

High-level analyses, transformations, and optimizations which benefit from the availability of explicit source-level information are performed at the Pcode level. Within the Pcode IR, program code is represented in an abstract syntax tree containing hierarchical statement and

[1] The HP PA-RISC code generator was developed by Richard E. Hank.

expression nodes. This hierarchical intermediate representation facilitates the manipulation of program structures such as loops and blocks of statements.

The Pcode module contains several code restructuring transformations and optimizations. General purpose loop transformations currently implemented include loop distribution or loop fission, rectangular loop interchange, loop skewing, and loop reversal [4]. These loop transformations are usually exploited as tools to improve the applicability of other transformations and optimizations. In addition, conversion of *while*-type loops into *for*-type loops to facilitate data dependence analysis is also supported. Loop parallelization is currently limited to identification of loops which can be software pipelined. Loops that are identified as good candidates for software pipelining are marked at the Pcode level, but the software pipelining transformation is actually accomplished at the Lcode level during code generation [17], [18].

Memory system optimizations include loop blocking (also called iteration space tiling) to improve cache access locality [5], software prefetching, and data relocation and prefetching [6], a hardware-assisted form of software prefetching which simultaneously relocates array data to reduce cache mapping conflicts.

To support these transformations and optimizations, Pcode performs several types of analysis. Control-flow analysis provides the structural framework upon which many of the transformations and other analyses are built. It consists of control-flow graph construction, loop detection and nesting determination (used mostly for unstructured loops), support for data dependence analysis, and unreachable code removal. Data-flow analysis determines the flow of program values, variables, and expressions through the control-flow graph. Traditional types of data-flow information are computed including sets of reaching definitions and uses, available definitions and uses, and live variables [19]. In addition, an extended type of data-flow analysis

called *loop-carried data-flow analysis* is used to calculate loop-carried reaching definitions and uses, which are useful for determining accurate dependence direction vectors for scalar variables.

Data dependence analysis [3] calculates the dependence relationship between each access pair in the function. It consists of several steps. First, a variable access table containing information for each distinct variable reference in the function is built. Next, aliases are added between accesses in the access table as necessary. These aliases may stem from several sources, such as aliases between elements of a union, pointer aliasing caused by assignment expressions, or aliases determined during interprocedural analysis. Finally, the dependence relationship between pairs of accesses is determined. The Omega Test [20], developed by William Pugh at the University of Maryland, is employed to produce the data dependence equations and inequalities used to generate distance and direction vectors for pairs of variable references.

Pcode's existing data dependence analysis lays the foundation for the dependence analysis used to support the sync arc research presented in this thesis. Chapter 6 further discusses Pcode data dependence analysis and the enhancements made to it as part of this thesis.

2.2 Lcode

The Lcode level performs low-level code optimization and scheduling to expose and exploit a program's inherent ILP. Lcode is logically subdivided into two subcomponents: machine-independent optimizations performed prior to code generation and machine-dependent optimizations performed during code generation. Although the internal data structures used during these two components of Lcode are identical, the machine-dependent portion of the Lcode is sometimes referred to as *Mcode*. The difference between Mcode and Lcode is that Mcode is broken down such that there is a one-to-one mapping between Mcode instructions and the

target machines' assembly languages. For example, when generating code for the X86 architecture, the Lcode will be in 3-operand format during machine-independent optimization, and then is converted to 2-operand format during the machine-dependent phases; once in 2-operand format, the code would be referred to as Mcode. Lcode instructions are broken up for a variety of reasons, such as limited addressing modes, limited opcode availability (e.g., no floating-point branch), ability to specify a literal operand, and field width of literal operands.

During the first step of Lcode compilation, all machine-independent classic optimizations are applied [21]. These include constant propagation, forward copy propagation, backward copy propagation, common subexpression elimination, redundant load elimination, redundant store elimination, strength reduction, constant folding, constant combining, operation folding, operation cancellation, code reordering, dead code removal, jump optimization, unreachable code elimination, loop invariant code removal, loop global variable migration, loop induction variable strength reduction, loop induction variable elimination, and loop induction variable reassociation. Additionally, analysis is performed to identify safe instructions for speculation [22].

The next step in Lcode compilation is to perform superblock code transformation and optimization. The superblock compilation structure is explained in detail in Section 2.3. When predicated execution support is available in the target architecture, hyperblocks [23] rather than superblocks are used as the underlying compilation structure. All superblock optimization techniques have also been extended to operate on hyperblocks. In addition, a set of hyperblock-specific optimizations to further exploit predicated execution support are available. For this thesis, the superblock was the primary compilation structure used for memory disambiguation experiments.

Following superblock transformations, machine-specific code generation is performed for one of the seven architectures shown in Figure 2.1. Code generation within the IMPACT compiler consists of three phases. During Phase I, Lcode to Mcode conversion is performed to transform the Lcode into a one-to-one correspondence to target machine assembly. During Phase II of code generation, machine-specific optimizations, code scheduling, and register allocation are performed. Finally, during Phase III of code generation, Mcode is translated into the target architecture's assembly language.

Two of the most significant components of code generation are the instruction scheduler and register allocator, both of which are common modules shared by all code generators. Scheduling is performed via either global acyclic scheduling [22], [24] or software pipelining [17], [18]. Global acyclic scheduling is applied both before register allocation (prepass scheduling) and after register allocation (postpass scheduling) to generate an efficient schedule. Loops targeted for software pipelining are identified and marked at the Pcode level. These loops are pipelined using modulo scheduling and the remaining code is scheduled using the global acyclic scheduler. Additionally, code transformations to support the memory conflict buffer technique described in Chapter 4 are applied during code scheduling.

Register allocation is performed using a graph-coloring-based scheme [25]. The register allocator employs profile information, if available, to better prioritize virtual registers for allocation to physical registers.

For each target architecture, a set of specially tailored peephole optimizations is performed. These peephole optimizations are designed to remove inefficiencies introduced during Lcode to Mcode conversion, to take advantage of specialized opcodes available in the architecture, and

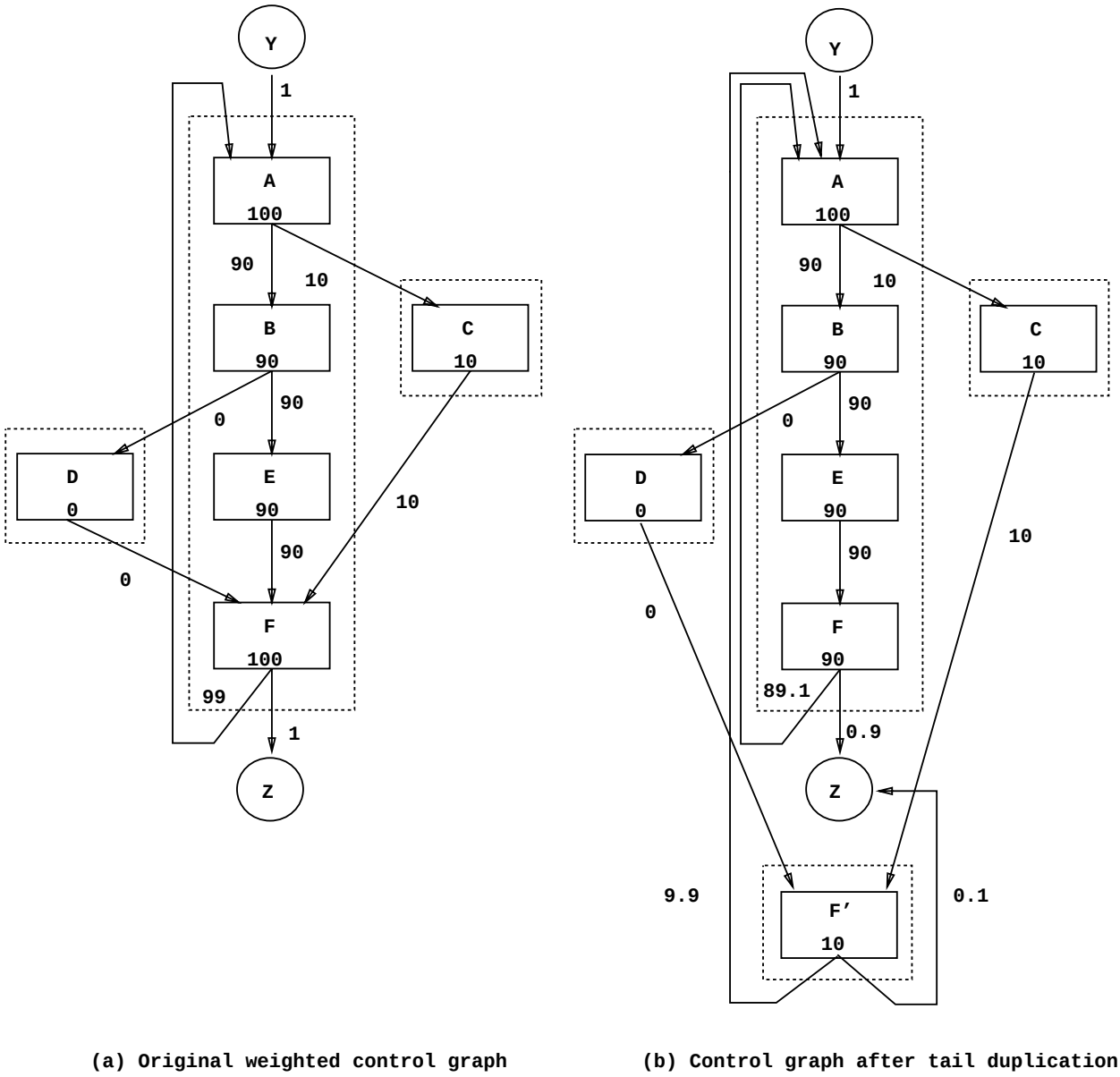
to take advantage of new optimization opportunities after spill code has been added by the register allocator.

2.3 Superblocks

For most non-numeric programs, the ILP available within individual basic blocks is extremely limited [26], [27], [28]. An ILP compiler must be able to optimize and schedule instructions across basic block boundaries to find sufficient parallelism. An effective structure for ILP compilation is the *superblock* [23], [29]. The formation and optimization of superblocks increases the ILP available to the scheduler along important execution paths by systematically removing constraints due to the unimportant paths. Superblock scheduling is then applied to exploit ILP by mapping it to the available processor resources.

A superblock is a block of instructions for which the flow of control may only enter from the top, but may leave at one or more exit points. It is formed by identifying sets of basic blocks which tend to execute in sequence (called a *trace*) [30]. These blocks are coalesced to form the superblock. Tail duplication is then performed to eliminate any side entrances into the superblock [31].

The formation of superblocks is illustrated in Figure 2.2, taken from [23]. Figure 2.2(a) shows a weighted flow graph which represents a loop code segment. The nodes in the graph correspond to basic blocks and the arcs represent the possible control transfers. The number in each node represents the execution frequency of the basic block (as determined by profiling). Likewise, the number associated with each arc represents the number of times that particular control transfer path is followed. Because the most frequent control flow is along the path $\{A, B, E, F\}$, this trace is selected for superblock formation. To eliminate side entrances to this



(a) Original weighted control graph

(b) Control graph after tail duplication

Figure 2.2 An Example of Superblock Formation.

superblock, tail duplication replicates basic block F, and control flow from blocks C and D is redirected to this duplicated block. The result is the flow graph shown in Figure 2.2(b).

Following superblock formation, ILP is further exposed through superblock optimizations. Classic optimizations are reaccomplished within the scope of the superblock. Superblock enlarging optimizations such as loop unrolling and loop peeling are employed to increase the size of superblocks, providing more visible instructions to the scheduler. Dependence-removing optimizations such as register renaming, induction variable expansion, and accumulator expansion are performed to remove data dependences, increasing available ILP. For a detailed explanation of the superblock optimizations, see [23].

The superblock compilation framework can be viewed as an attempt to reduce the impact of control transfer instructions on ILP. For an architecture that supports control speculation, the greater optimization and scheduling freedom afforded by superblocks significantly reduces the negative impact of branches on ILP. The importance of this result to this thesis is that reducing the impact of branches on ILP has exposed ambiguous memory dependences as a secondary impediment to ILP. The potential ILP exposed by superblock formation cannot be fully exploited unless effective methods are developed to overcome the restrictions imposed by memory dependences.

2.4 IMPACT Simulation Environment

All experiments performed for this thesis were done using the IMPACT simulation environment. The IMPACT simulator models in detail the target architecture's prefetch and issue unit, instruction and data caches, branch prediction mechanism, and hardware interlocks. This allows the simulator to accurately model the number of cycles required to execute a program,

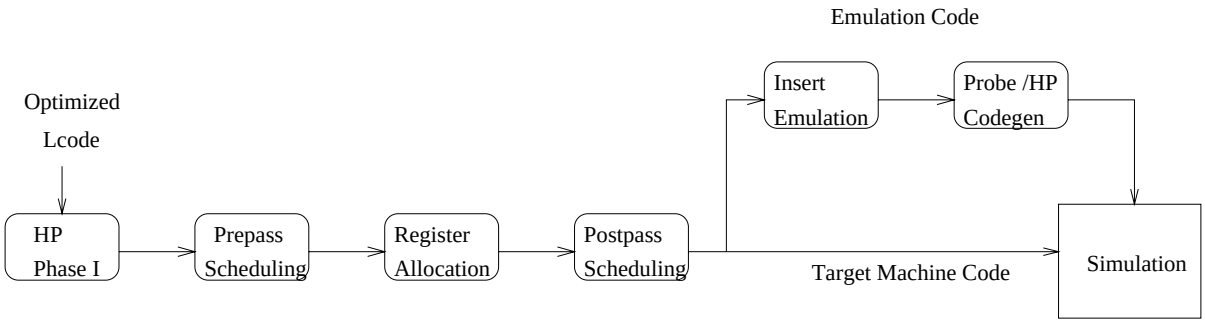


Figure 2.3 MCB Compilation Path for Simulation.

as well as provide detailed analysis such as cache hit rates or branch prediction performance. The simulator also allows proposed new hardware, such as the memory conflict buffer, to be accurately modeled and analyzed. Supported architecture types include in-order superscalar and very long instruction word (VLIW) architectures.

The IMPACT simulation approach is referred to as emulation-driven simulation. Figure 2.3 shows the compilation path for the simulation used throughout this thesis. The figure assumes the Lcode has already been compiled through classic and ILP code optimizations, including superblock formation. Because the simulation performed for this thesis assumes an instruction set architecture which is an extension of the HP PA-RISC 1.1 instruction set, the optimized code is first run through the initial phase of the HP PA-RISC code generator, which transforms the code into HP Mcode. The code is then passed through pre-pass scheduling, register allocation, and post-pass scheduling for the target architecture, using the generic IMPACT code generator. During this stage, architectural features of the simulated architecture are assumed. For example, if the architecture being simulated can issue eight instructions per cycle, the scheduler reorders the code based upon this model. For the MCB experiments detailed in Chapter 4, the MCB code transformations are performed during pre-pass scheduling.

Following this stage of compilation, the intermediate code is in a form which could be executed by the simulated architecture. However, to create an executable file to drive the simulation, any unsupported architectural features of the simulated architecture must be emulated to allow the code to execute on the *host* architecture, an HP PA-RISC 7100-based workstation. For example, if the simulated architecture contains hardware support for MCB, emulation code must be added to allow the code to execute properly on the host architecture. Following insertion of required emulation code, a second phase of register allocation, assuming host architecture register file constraints, is performed. The code is then instrumented to gather address and branch direction data for the simulation, and then the final phases of the code generation are performed to create an executable file. This executable file serves two purposes. First, because the executable can be run to provide correct program results, it verifies that code transformations have been performed correctly. Second, it generates the trace information required to drive the simulation.

Simulation is performed on the modeled architecture's code, using address and branch direction data from the emulation path. The result is a highly accurate measure of the number of cycles required to execute the program on the simulated architecture. Due to the complexity of simulation, *sampling* [32] is used to reduce simulation time for large benchmarks. For sampled benchmarks, a minimum of 10 million instructions are simulated, with at least 50 uniformly distributed samples of 200,000 instructions each. Testing has shown sampling error to be less than 1% for all benchmarks.

Further details of the architecture being modeled for various experiments is provided within the experimental sections of this thesis.

CHAPTER 3

OVERVIEW OF MEMORY DISAMBIGUATION

Control flow instructions (e.g., branches and function calls) have been widely recognized as the major impediment to exposing ILP. Because such a high percentage of instructions (20-30%) in typical C programs are control flow instructions, the compiler must be able to search beyond the individual basic block for parallelism. Techniques such as trace scheduling [30], superblocks [29], and hyperblocks [23] have been developed to expand the size of blocks in which the compiler performs optimization and scheduling. *Speculative execution* techniques have been developed to allow code motion between basic blocks [33], [34], [35]. As a result of these techniques, the impact of control flow instructions on ILP can be significantly reduced.

However, this reduction of the impact of control flow instructions on ILP has exposed a secondary impediment to ILP: ambiguous memory dependences [1]. In much the same way that branches can restrict code optimization and scheduling, ambiguous memory dependences also prohibit these important transformations. In particular, dependences between loads and stores result in the greatest restriction to ILP.

Dependences between two loads (referred to as an *input dependence*) usually have little or no impact on ILP. Dependences between two store operations (referred to as an *output dependence*) rarely restrict optimization, and there tends to be limited benefit from reordering stores during code scheduling. However, dependences between load and store operations are a much more serious problem for the compiler. During code scheduling, *flow dependences* (the situation when a load operation sequentially follows a dependent store operation) often severely restrict

code reordering. *Anti dependences* (when a load sequentially precedes a dependent store) tend to be a minor problem during scheduling because load operations tend to move upward and stores tend to move downward during code scheduling. During optimization, both flow and anti dependences severely restrict transformations.

To overcome the problem of ambiguous memory dependences, optimizing compilers perform memory disambiguation. To illustrate the potential benefit of memory disambiguation, an experiment was conducted using a suite of twelve benchmarks, including five unix benchmarks, five SPEC-CINT92 benchmarks, and two SPEC-CFP92 benchmarks. The benchmarks were scheduled using the three different models of disambiguation. In the first model, no memory disambiguation was performed, i.e., all memory operations were assumed to be dependent on all other memory operations. The second model used the existing (prior to this thesis) IMPACT low-level memory disambiguation. This disambiguation model is typical of the static analysis performed on low-level code by current commercial compilers. The analysis is strictly intraprocedural and uses only information available within the low-level code for its analysis, i.e., no source-level information is used to aid the analysis. It is designed to be fast and fully safe, but is limited in its effectiveness. The final model used in this experiment is ideal disambiguation, where all memory operations are considered to be independent unless the static analysis proves them to be dependent. Note that this disambiguation model makes unsafe assumptions and may result in incorrect code if truly dependent operations are reordered. However, this model provides an upper bound on the performance which could potentially be achieved from scheduling with improved memory disambiguation.

For this experiment, an 8-issue architecture with 64 integer and 64 floating-point registers is assumed. No restrictions are placed on the combination of instructions that may be issued

in a cycle. Instructions latencies of the HP PA-7100 are also assumed. Because the executable generated from the ideal disambiguation model could execute incorrectly, the experiment was performed using cycle count estimates provided by the scheduler rather than the full simulation approach described in Section 2.4. To estimate the number of cycles required for execution, the code was profiled prior to scheduling to determine the execution frequency of each control block. The code was then scheduled, using the various levels of disambiguation, to determine the number of cycles each block would take to execute. From this, an accurate estimate of required execution cycles can be determined, excluding cache effects and branch-misprediction penalties.

Figure 3.1 presents the results of this experiment. The vertical bars on the graph reflect the relative speedup of the current static and ideal disambiguation models over the baseline no-disambiguation case. Thus, a speedup of 1.0 indicates equivalent performance to the baseline case. Several items should be noted from this figure. First, the ideal disambiguation results indicate that by eliminating memory dependences a significant amount of potential ILP can be exposed. For six of the twelve benchmarks tested, more than 50% speedup could be achieved if all ambiguous memory dependences could be eliminated. A second point to note is that IMPACT's current low-level memory disambiguation is in large part ineffective at removing ambiguous memory dependences and exposing parallelism. It appears that a significantly more powerful disambiguation technique is required to eliminate ambiguous dependences and provide performance closer to the ideal disambiguation case. Finally, the experiment demonstrates that although improved memory disambiguation can significantly increase performance, it is not a panacea which can increase performance on all benchmarks. For example, in the benchmark *023.eqntott*, over 80% of the execution time is spent in an inner loop which contains no store

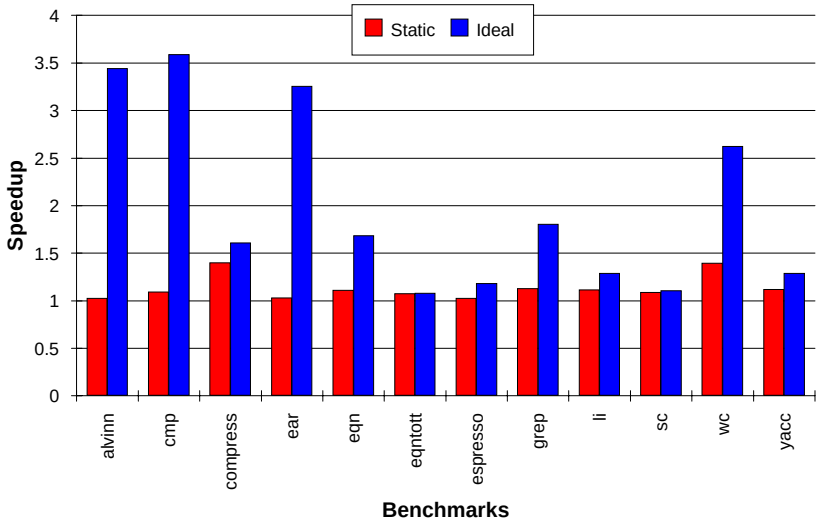


Figure 3.1 Effect of Memory Disambiguation on Performance.

operations; thus, improved memory disambiguation has little effect on performance. Note that this experiment only measures the potential benefit of improved memory disambiguation during scheduling; one would also expect a significant benefit for optimization. This benefit is quantified in Chapter 6.

The remainder of this chapter examines two general memory disambiguation approaches: dynamic and static. Tradeoffs between these two approaches are examined, followed by a review of related work for each of the two approaches.

3.1 Tradeoffs Between Dynamic and Static Approaches

Various solutions have been proposed to provide improved memory disambiguation. In general, these solutions can be categorized as either dynamic or static. Static memory disambiguation, also referred to as *dependence analysis*, attempts to determine the relationship between pairs of memory references at compile time. Once the compiler has determined the

Table 3.1 Tradeoffs of Dynamic and Static Memory Disambiguation.

Dynamic	Less compile-time investment More accurate Compiler support confined to backend Useful when source not available
Static	Requires no hardware support Requires no instruction overhead May be nearly as accurate in practice

dependence relationship for memory references, this information can be used to safely direct subsequent code transformations. In contrast, dynamic memory disambiguation attempts to determine at run-time whether two references could possibly reference the same memory location. The compiler may speculatively perform a code transformation based upon an assumed dependence relationship between memory references, and then provide some means of dynamically determining at run-time if the assumed relationship was correct. In the event of an incorrect assumption, the dynamic approach must provide a mechanism to ensure correct execution.

Both dynamic and static memory disambiguation approaches are targeted toward increasing processor performance. Tradeoffs exist between the approaches; a particular implementation may employ static techniques, dynamic techniques, or some combination of both. Table 3.1 highlights some of the relative advantages of the two approaches.

Dynamic approaches usually require significantly less compile-time investment than static approaches. In general, the compiler algorithms to support most currently proposed dynamic approaches do not significantly impact the overall compilation time. In contrast, static memory disambiguation requires an in-depth analysis which can dominate the time required for compilation. Languages such as C, which require interprocedural analysis to provide high accuracy, require an even greater investment in compilation time. For applications requiring extremely

```
for ( i = 0; i < M; i++)
   for ( j = 0; j < N; j++)
      A[j] = A[ B[i] ];
```

(a) indirect references

```
for ( i = 0; i < M; i++)
   A[i] = A[M-i];
```

(b) occasional dependence

```
for ( i = 0; i < M; i++)
   A[i] = A[i*i + 3];
```

(c) non-linear references

```
while (ptr != 0)
   ptr = ptr->next;
```

(d) pointer references

Figure 3.2 Limitations of Static Memory Disambiguation.

fast compilation, a dynamic memory disambiguation technique may prove to be a better approach.

Because dynamic approaches do not attempt to determine the dependence relationship between two operations until run-time, they are inherently more accurate than static approaches. During program execution, the dynamic approach knows the exact memory address being accessed by each reference and, thus, can determine dependence relationships with complete accuracy. Although static memory disambiguation approaches can be highly accurate for many applications, current techniques are unable to accurately determine dependence relationships in certain circumstances. Figure 3.2 highlights code segments for which static memory disambiguation is less effective. In Figure 3.2(a), the reference to $A[B[i]]$ is an indirect reference through a second array. Because the static analysis cannot determine the value stored in the location $B[i]$, it is unable to accurately determine the dependence relationship of this reference of the array A to other references to the same array.

Figure 3.2(b) illustrates the problem that occasional dependences cause for static memory disambiguation. In this example, a loop-carried dependence exists between the two references to the array A. However, the dependence distance (the number of loop iterations from the iteration in which one reference accesses a certain memory location until the other reference accesses the same address) is not constant between loop iterations. Thus, the static dependence analysis cannot accurately determine the dependence relationship. If the value for M cannot be determined by the compiler, the static analysis also cannot determine when and if a zero distance (non-loop carried) dependence exist between the two references. In Figure 3.2(c), the problem of non-linear references is shown. Because most static memory disambiguation approaches cannot handle non-linear array indices, the reference to $A[i * i + 3]$ cannot be disambiguated from other references to the same array. Finally, Figure 3.2(d) shows an example of the problem with performing static analysis for languages which support pointers. In this example, a simple loop that walks a linked list data structure is shown. Unless the dependence analysis is able to somehow determine that the list in acyclic, the dependence relationship between the references to ptr and $ptr \rightarrow next$ cannot be accurately determined.

Supporters of static memory disambiguation would likely contend that the array examples shown in Figure 3.2 do not occur frequently enough in most applications to result in significant loss of accuracy for static analysis. Little or no data exist to quantify how often these situations occur on real applications. Static analysis of pointers has improved greatly in recent years, reducing this accuracy advantage of dynamic approaches. Thus, although dynamic approaches are inherently more accurate, static approaches may prove to be nearly as accurate for most applications.

Another advantage of dynamic memory disambiguation approaches is that their compiler support is usually confined to the back end of the compiler. Code transformations to support dynamic approaches are normally performed on the low-level form of the code being compiled. Thus, a single implementation of a dynamic approach can provide memory disambiguation for an application that supports numerous front-end source languages. The transformations are independent of the source language. Static approaches, on the other hand, are normally performed on high-level source code. If an application that must support multiple source languages employs static memory disambiguation, a unique implementation will likely be required for each of the supported languages.

The primary advantage of static memory disambiguation is that it requires no support beyond the analysis performed by the compiler. In contrast, dynamic approaches can require several types of overhead. First, dynamic approaches usually require the insertion of extra instructions into the code stream to provide the run-time checking. Even for wide-issue architectures, these additional instructions may result in a performance penalty. Thus, for a static approach and a dynamic approach that provide comparable accuracy, the static approach will likely have better performance due to the instruction overhead of the dynamic approach. Second, dynamic approaches often require instruction-set architecture (ISA) support, in the form of new instructions. This requirement limits the application of some dynamic approaches for existing architecture families. Also, the addition of new instructions to the ISA which require additional bits may be very difficult and require extensive redesign. Finally, some dynamic approaches require significant hardware support. The hardware cost of the approach must be considered along with the potential performance improvements from the improved memory disambiguation, i.e., does the performance improvement provided by the dynamic technique

outweigh the potential improvement if the same amount of chip area was applied to some alternate hardware feature (e.g., a larger cache)?

It should not be construed that the tradeoffs discussed above, or the performance comparisons provided later in this thesis, are intended to imply that either static or dynamic memory disambiguation is a better approach for all applications. The strengths and weaknesses of the approaches may make either, or possibly a combination of both, the best solution for a particular application. In fact, certain applications force the use of one approach or the other. For example, a static memory disambiguation approach that relies on use of source-level information is not possible for an application such as binary translation, in which no source-level information is available. For applications that must be compatible across an architecture family, such as the X86 family, a dynamic approach requiring changes to the ISA would not be useful.

3.2 Dynamic Memory Disambiguation Approaches

Dynamic memory disambiguation attempts to determine at run-time whether two references could possibly reference the same memory location. Most dynamic approaches deal specifically with memory flow dependences, attempting to remove these dependences and allow loads to execute before ambiguous stores. When a load operation, and possibly the load's dependent operations (flow dependences associated with the destination register of the load), bypass an ambiguous store operation, the operations that pass the store are being executed speculatively before it is determined that the value the load accesses is valid. This is termed *data speculation*. In contrast to *control speculation*, in which operations are executed before it is determined that they should have been executed according to original program control flow, data speculation executes instructions before it is determined that the data being used are valid.

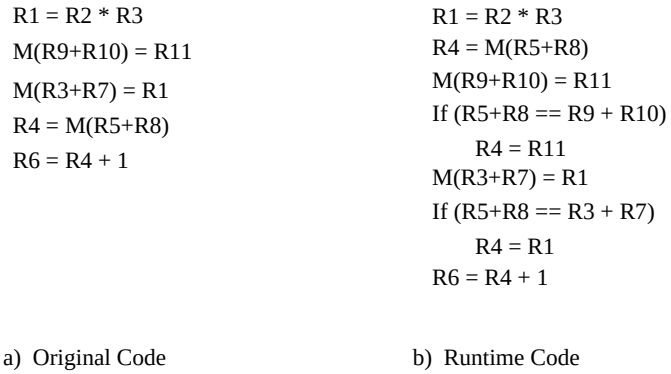


Figure 3.3 Run-time Memory Disambiguation Example.

In this section, several models of data speculation are discussed, requiring varying degrees of architectural support. These models also vary in what instructions can be speculated, i.e., whether only load instructions, or both the load and its dependent operations, are allowed to bypass stores. First, a compiler-only model known as *run-time disambiguation* is presented. As presented, this model allows only load instructions to be speculated, but it could easily be extended to also allow dependent operations to be speculated. Next, five models that use a combination of architectural and compiler support are examined. Two of these allow only loads to be speculated, and the remainder allow dependent operations to be speculated also. Finally, the hardware-only model of dynamic memory disambiguation is reviewed.

3.2.1 Run-time disambiguation

Nicolau has proposed a software-only data speculation technique known as *run-time disambiguation* [36]. Run-time disambiguation inserts explicit address comparisons and conditional branch instructions into the code which allow memory flow dependences to safely be removed, enabling load instructions to percolate upward past ambiguous stores during code scheduling. Figure 3.3 illustrates the application of run-time disambiguation. The original code segment in

Figure 3.3(a) has two store operations followed by an ambiguous load. Figure 3.3(b) reflects the application of run-time disambiguation to the code: the load has been moved above both stores, and explicit address comparison code has been added. When the address comparison code determines that the addresses of the load and the store were identical, the value being stored is simply moved into the destination register of the load. In the example shown, if the address of the load operation ($R5 + R8$) is the same as the store address ($R9 + R10$), then the subsequent move operation places the value from $R11$ in the load's destination ($R4$). Similar code is also added following the second store operation.

The major advantage of run-time disambiguation is that it requires no ISA or hardware support, and thus could be applied to existing architectures or families of architectures. However, it has several major limitations, particularly when applied to ILP processing. First, the technique can result in an extremely large amount of code growth when used with ILP compilation techniques. The number of address comparison and conditional branch instructions inserted can be prohibitive as a result of aggressive code reordering: if m loads bypass n stores, $m \times n$ comparisons and branches are required. Second, the technique adds branch instructions to the code. Although superscalar and VLIW processors can issue and execute many operations each cycle, they are typically very limited in the number of branch operations they can execute (usually only one branch per cycle). These added branches are usually highly predictable, but they may impact performance in branch intensive code. A third limitation of run-time disambiguation is that it does not readily address the *access width* problem. Simple address comparison is insufficient to detect ambiguity in the presence of memory instructions of different size (e.g., an integer store followed by a character load). To ensure correct execution, a number of the least-significant-bits of the addresses must be ignored during address comparison, requiring

further instruction overhead. Finally, the technique as proposed allows only load operations, and not their dependent operations, to bypass stores. This significantly limits the scheduling freedom necessary for exploiting ILP. Run-time disambiguation could be extended to allow the load's dependent operations to bypass the store, using the compilation techniques described in Chapter 4.

3.2.2 Preload register update

A major limitation of run-time disambiguation discussed above is the requirement that explicit address comparisons be added to the code. The *preload register update* technique proposed by Chen et al. [37] attempts to relieve this problem by using hardware to perform the address comparisons and move operations. A *preload* instruction informs the hardware that a load is being speculated above ambiguous stores, and therefore requires its address be saved and checked against subsequent store addresses. If a match occurs between a store and load address, the hardware "updates" the destination register of the load with the store value. Address comparisons for the preload continue until a commit instruction is executed. A commit instruction is needed so that the coherence mechanism (the checking of store and load addresses and execution of potential updates) can be turned off for this particular load. This ensures that only stores that were originally located before a speculated load are allowed to update the load's destination register. The compiler must ensure that a commit instruction is not moved above or below a store instruction during code transformations.

Figure 3.4 demonstrates the preload register technique using the previous code example. Note in Figure 3.4(b) that the load has again bypassed the stores, and has been marked as a preload. Following the last store, a commit instruction has been added. Note also that, like

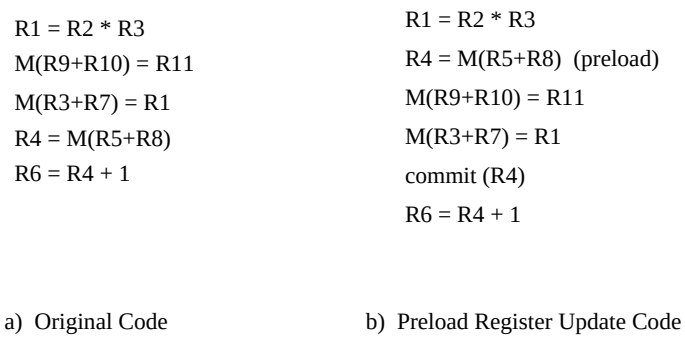


Figure 3.4 Preload Register Update Example.

run-time disambiguation, the dependent operation ($R6 = R4 + 1$) is not allowed to bypass the stores.

Preload register update successfully eliminates the code growth problem of run-time disambiguation, and it does not require the addition of branches. Hardware mechanisms could also be provided to overcome the access width problem. The major limitations of the technique are that it requires both ISA and hardware support and that it does not allow the load's dependent operations to bypass stores.

3.2.3 HP smart load

Hewlett Packard has developed a scheme, similar to preload register update, which also allows load instructions to be moved above ambiguous stores [38]. Every speculated load defines a watch window which indicates how many instructions above its original position the load has been speculated. The register file is modified to store the address of the preload in addition to the data, and includes counters used to determine when the preload's watch window is no longer active. Additionally, a 2-bit flag records whether the register contains an active (being watched) load value and whether a subsequent store has matched the speculated load's address.

If an incoming instruction is a load and it has been speculated, its destination register has to be initialized. This initialization includes setting the counter to the number of instructions above its original position that a load has been speculated and setting the flag to indicate the register contains an active load. When a store instruction is issued, its address must be checked against all the active load addresses found in the register file. If a match is found, the corresponding bit in the register file is set to record the match. When the original position of a speculated load instruction is reached, a new load is generated if the flag state indicates that a store address match has occurred.

The compiler support required for this technique is also very similar to preload register update. Rather than marking the load as a preload as shown in Figure 3.4(b), the smart load technique would mark it as being speculated two instructions. Thus, after the two store operations it bypassed have been executed, the counter associated with $R4$ would have the value zero and the load would be committed. No explicit commit instruction is required.

One variation on this scheme utilizes forwarding. If a store address conflicts with a speculative load address, the data contained in the store are used instead of the data obtained by the load. This method is very similar to preload register updating, and makes generation of the extra load instruction unnecessary.

3.2.4 Speculative disambiguation

Huang et al. have proposed *speculative disambiguation* [39], a combined hardware and compiler technique to allow aggressive code reordering using predicated instructions. It is similar to run-time disambiguation, but employs compiler techniques that allow both a load and its dependent instructions to bypass an ambiguous store. The method also allows two

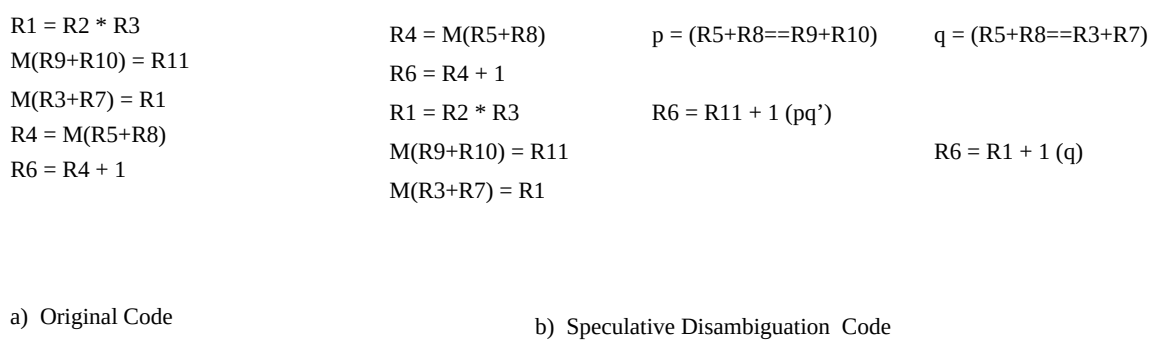


Figure 3.5 Speculative Disambiguation Example.

ambiguous stores to be reordered. This is accomplished by generating code for both the case when the two instructions are independent and for when they are dependent. The two versions of the code are conditioned by opposite predicates, so that only one version of the code is actually executed.

Figure 3.5 illustrates this technique using the running code example. In Figure 3.5(b), the predicated code is shown in several columns, corresponding to different predicate cases. The first column shows the case in which the load is independent of the stores, and it can be freely scheduled past the stores. The second column handles the case in which the first store conflicts with the load, but not the second. This is indicated by predicate p being true and predicate q being false. In this case, the add instruction which originally used $R4$ now uses the store value, $R11$, as its input. Note that the load is not re-executed in the case of a conflict, but all uses of the load's destination register are re-executed using the alternate value. The third column shows the case when the load conflicts with the second store, indicated by predicate q being true. Here, the add instruction is re-executed using the value stored in $R1$.

Note the code growth from a single load with only one dependent operation bypassing two stores. In the presence of aggressive code reordering, code growth would be prohibitive.

Additionally, the issue bandwidth of the ILP processor would quickly be saturated. Thus, the technique cannot be generally applied to support ILP compilation, and is more suitable for narrow-issue processors requiring only minimal code reordering. The primary advantages of the technique are that it requires no additional hardware overhead (for processors already supporting predication) and that it does allow the load's dependent operations to bypass stores to a limited extent.

3.2.5 Unsafe loads

Silberman and Ebcioglu presented a dynamic memory disambiguation scheme as part of their framework for supporting heterogeneous instruction set architectures [40]. This framework was developed to allow applications written for one instruction set to be migrated to a higher performance architecture without a significant investment by the user or developer.

They use both a base machine engine which executes the original instruction set architecture and a native machine engine with a higher performance architecture (e.g., a RISC engine) to implement their scheme. Two versions of the code are generated, one for each engine. For best performance, the goal is to execute the native version of the code as much as possible, periodically updating the base machine state at predetermined checkpoints. The approach employs the concept of both architected registers (registers present in original instruction set) and nonarchitected registers (extra registers present in the native engine).

In the native version of the code, they allow loads whose destination is a nonarchitected register to be speculated above ambiguous stores. The nonarchitected registers have additional fields which store the memory address of the load, its length, and an extension tag which indicates whether an address conflict has occurred.

At each checkpoint, the nonarchitected registers are copied into the architected registers to update state. If the extension flag of the nonarchitected register is set, an address conflict has occurred since the previous checkpoint and the native machine state may not be valid. In this case, the processor re-executes the section of code containing the speculated load in the base machine engine and re-enters the native engine at the next opportunity.

Although applied within the context of the heterogeneous instruction set architectures, this approach to dynamic disambiguation has general application. The hardware requirements are very similar to the hardware requirements for the memory conflict buffer approach presented in Chapter 4. Although requiring extensive hardware support, the technique allows both loads and their dependent instructions to bypass stores.

3.2.6 Hardware-only disambiguation

Hardware-only dynamic memory disambiguation techniques have been widely used for architectures which employ dynamic instruction scheduling. Early dynamic architectures such as the IBM 360/91 [41] and the CDC 6600 [42] employed simple store queues which allowed subsequent loads to execute out-of-order. The HPS architecture [43] proposed *node tables* which buffered memory operations awaiting operands, allowing subsequent memory operations to execute. A store queue was also employed to allow loads to bypass stores. Franklin and Sohi proposed the *address resolution buffer* [44], which also allows dynamic reordering of memory operations. It provides special support allowing subsequent memory operations to execute even if the address of an earlier store operation has not been resolved. For wide-issue ILP architectures, each of these dynamic scheduling techniques is limited by the size of the visible instruction win-

dow, restricting the aggressive code reordering necessary to obtain high utilization of multiple functional units.

3.3 Static Memory Disambiguation Approaches

Static memory disambiguation, or dependence analysis, attempts to determine the relationship between two references at compile-time. Most frequently, dependence analysis has been applied at the source code level, and is used to facilitate source-to-source code transformations. In-depth static analysis has seldom been applied to assist compilation of low-level code; in most commercial compilers, memory disambiguation for low-level code is performed using only information available within the low-level code (i.e., no source-level information). A few of the newest optimizing compilers attempt to pass some limited source information to the intermediate code, but this information is usually limited to array references. Because so little work has previously been done on performing dependence analysis to facilitate low-level code optimization and scheduling, the discussion in this section focuses on techniques which are being developed primarily to support source-level transformations. This related work is presented in the context of the several complications to dependence analysis which must be addressed to provide accuracy.

The first complication to dependence analysis is disambiguating array references, particularly in the context of loops. To test for dependence between array references, compilers have traditionally relied on several well-known algorithms based on a set of Diophantine equations [45], [46]. More recently, techniques have been developed which are able to handle multi-dimensional arrays and more complex array subscripts [47], [48], [49], [50]. Although array dependence analysis has reached a fair level of maturity, current techniques may achieve inex-

act results due to complicated reference patterns or occasional dependence, as was illustrated in Figure 3.2.

A second complication to dependence analysis is the presence of *aliasing*, the situation when two or more distinct variables simultaneously refer to the same memory location. In languages without pointers, such as Fortran, aliasing occurs most frequently due to the binding of formal parameters upon subroutine entry. Alias analysis for such languages is well understood [51], [52]. However, languages which allow pointers (e.g., C) severely complicate dependence analysis. Numerous interprocedural analysis techniques have been proposed to resolve pointer aliasing [53], [54], [55]. These techniques can provide good pointer disambiguation, but may be limited in application due to their compilation time and memory requirements. Dependence analysis for C will be discussed further in Chapter 6.

A third complication of dependence analysis is the presence of recursive data structures. As discussed earlier, it may be difficult to disambiguate between different elements of a linked list unless the analysis can prove the list is not circular. Much of the research in this area has been focused on automatically identifying the nature of abstract data structures (i.e., is the structure cyclic, a directed acyclic graph, or a tree) [56], [57]. Hendren has proposed augmenting the source language to allow the user to describe data structures, providing the compiler with additional information for disambiguating these structures [58]. Hummel et al. have proposed an axiom-based dependence test for references to recursive data structures, using the principles of theorem proving [59].

Much progress has been made toward overcoming these complications. However, little work has been done to apply this type of source-level analysis to aid low-level optimization and

scheduling. In Chapter 5, a technique to propagate the results of a source-level analysis down to the low-level code is proposed.

CHAPTER 4

DYNAMIC MEMORY DISAMBIGUATION USING THE MEMORY CONFLICT BUFFER

Dynamic memory disambiguation may be a good choice for resolving dependences for many applications, including those requiring rapid compilation and those for which static analysis is otherwise not practical. However, the solutions examined in Section 3.2 have limited application to ILP compilation. These solutions either suffer from prohibitive code growth when applied during ILP compilation, or fail to expose sufficient ILP because they do not allow a load's dependent operations to bypass stores. Testing performed as part of this thesis indicates that scheduling without allowing the load's dependent operations to percolate past stores provides only about 20% of the potential performance benefit as compared to allowing both the load and its dependent operations to bypass stores.

The memory conflict buffer (MCB) scheme, first proposed in Chen's thesis [1], provides a good solution to both of these problems. Code growth is bounded, and full scheduling freedom is allowed. The MCB approach extends the idea of run-time memory disambiguation by introducing a set of hardware features to eliminate the need for explicit address comparison instructions. The MCB approach involves the introduction of two new instructions: 1) *preload*, which performs a normal load operation, but signals the hardware that a possible dependence violation exists for this load; and 2) *check*, which directs the hardware to determine if a violation has occurred and to branch to conflict correction code if required. Figure 4.1 demonstrates the MCB approach using the code example from Chapter 3. In the original code prior to

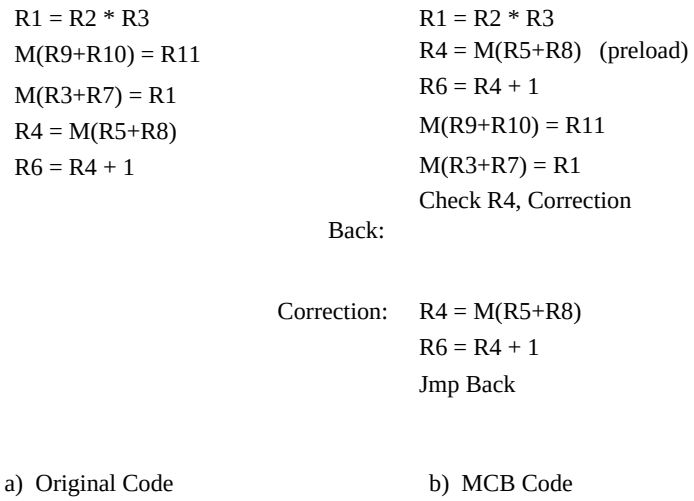


Figure 4.1 Memory Conflict Buffer Example.

scheduling (Figure 4.1(a)), the load operation ($R4 = M(R5 + R8)$) and its register dependent operation ($R6 = R4 + 1$) follow two ambiguous stores. In Figure 4.1(b), both the load and its dependent operation have bypassed these stores. Note the load has been changed to a preload, and a check instruction has been inserted at the original location of the load. If the hardware determines an address conflict has occurred, the check instruction will branch to correction code, which re-executes the load and any dependent instructions. In contrast to run-time memory disambiguation, only one check operation is required regardless of the number of store instructions bypassed by the preload. As a result, the MCB scheme allows the compiler to perform aggressive code reordering with significantly less code expansion and execution overhead than other dynamic memory disambiguation techniques. The drawback of the approach is that it requires a significant ISA and hardware investment.

4.1 Architectural Support

With the introduction of the preload and check opcodes, the compiler is free to move load instructions and their dependent operations past ambiguous stores. The MCB hardware supports such code reordering by 1) detecting the situation in which the ambiguous reference pair each access the same location and 2) invoking a correction code sequence supplied by the compiler to restore the correctness of program execution. The situation in which a preload and an ambiguous store access the same location will be referred to as a *conflict* between the two instructions. When this occurs, the reordered load and any dependent instructions which bypassed the store must be re-executed.

In order to detect conflicts as they occur, the MCB hardware records address information for each preload instruction when it is issued. The addresses of subsequent store instructions are then compared to this address information to determine whether a conflict has occurred. The hardware records the occurrence of the conflict; when the corresponding check instruction is encountered, the hardware performs a conditional branch to correction code if a conflict has been recorded. The correction code re-executes necessary instructions and then returns to normal program execution. In this section, the MCB hardware to detect and record load-store conflicts is presented and other issues affecting the hardware are discussed.

4.1.1 MCB design

The MCB hardware is responsible for storing preload address information for comparison to subsequent store addresses. In his thesis, Chen [1] discusses three possible hardware designs for the MCB: 1) fully associative, 2) set associative, and 3) hashing. Perhaps the most direct approach of the three is to store all address bits in some form of fully associative structure.

However, a fully associative search of any reasonably-sized MCB implementation would likely impose constraints upon processor pipeline timing. Additionally, the hardware costs to record 32 or more bits of address information for each preload would be expensive. Chen's fully associative design required one MCB entry corresponding to each architectural register. Thus, the design does scale well as the number of architectural registers increases. As an alternative, Chen proposed a set associative MCB design, similar in concept to a set associative cache. The preload address is used to select a set in the MCB array, and both the preload address and the destination register number are then stored in an available entry. Although this approach eliminates the requirement for a fully associative search of the MCB, it still suffers from large storage requirements. Also, neither the fully associative nor the set associative design addresses the access width problem, which occurs when accesses of different sizes conflict even though their addresses are not identical. Thus, Chen proposed the hashing MCB design, which uses a direct-mapped approach. Unlike the other two approaches, the load address is not explicitly stored within the MCB array. Only the destination register number is stored. The size of the MCB is significantly smaller, but *false conflicts* arise when two different addresses map to the same MCB location. To minimize these false conflicts, a hashing scheme is used to map the incoming preload or store address to a particular MCB array location. The hashing scheme was primarily developed to address the access width problem; this will be explored further in Section 4.1.3. Only the fully associative and hashing schemes were evaluated in Chen's thesis.

This thesis proposes an MCB hardware design which combines the best features of Chen's set associative and hashing schemes. This design, shown in Figure 4.2, was developed with scalability, access time, and physical size constraints in mind. The MCB hardware consists of

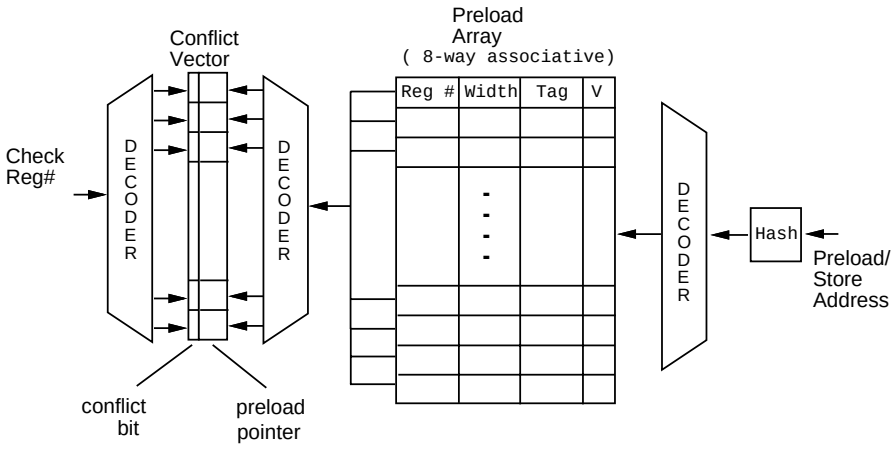


Figure 4.2 Set Associative MCB Design.

two primary structures, corresponding to the needs to store address information and to record conflicts which occur: 1) the *preload array*; and 2) the *conflict vector*.

The preload array is a set associative structure similar in design to a cache. Each entry in the preload array contains four fields: 1) the preload destination register number; 2) the preload access width; 3) an address *signature* or *tag*; and 4) a *valid* bit indicating whether the entry currently contains valid data. The preload register field simply contains the register number of the preload destination. The address signature contains bits which contain a hashed version of the preload address. Rather than storing the entire address as in Chen's set associative scheme, only a few bits are stored to reduce false conflicts. The access width field contains two bits to indicate whether the preload was of type character, half-word, word, or double word; additionally, this field contains the three least significant bits of the preload address. The use of the access width field will be discussed in Section 4.1.3.

The conflict vector is equal in length to the number of physical registers, with one entry corresponding to each register. Each entry contains two fields: the conflict bit and the preload

pointer. The conflict bit is used to record that a conflict has occurred for a preload to this register. The preload pointer specifies which preload array line currently holds the preload associated with this register and allows the preload entries to be invalidated by the check instruction.

When a preload instruction is executed, the address of the preload is hashed to select which set in the preload array will store the preload. (The hardware to perform this hashing, as well as address signature generation, is detailed in the next section.) The preload array is set associative; selecting an entry in which to store the preload information is identical to selecting an entry in a set associative cache. If there is an entry within the set which does not have its valid bit set, the preload information can be placed in this entry. When no invalid entry exists, a random replacement algorithm is used to select which entry to replace. If a valid entry is replaced, a *load-load conflict* has occurred; in this situation safe disambiguation can no longer be provided for the preload which is being removed from the array. It must therefore be assumed a conflict has occurred for this entry and the conflict bit corresponding to the register number being removed must be set. Note that for processors which support the execution of multiple preload instructions per cycle, the preload array must be multiported to allow simultaneous insertion of multiple preloads.

Having determined which entry in the preload array will be used for the current preload instruction, the destination register number and access width information are stored in the array. A second, independent hash of the preload address is performed to create the preload's address signature, which is stored in the signature field of the array. Unlike the tag field of a cache which must provide exact matching, this signature field can be hashed to reduce its size; the MCB can tolerate the occasional *false conflicts* which result from hashing. Simultaneously

with storing the preload in the preload array, the conflict vector associated with the load's destination register is updated, resetting the conflict bit and establishing the pointer back to the preload array.

When a store instruction is executed, its address is hashed identically to the preload to determine the corresponding set in the preload array and to determine the store's address signature. The store's access width data (2 size bits and 3 LSBs) are also presented to the array. To determine whether a conflict has occurred, the store's signature and access width information are compared with the data stored within each entry of the selected set. For each entry in the set which is determined to conflict with the store, the conflict bit corresponding to the preload register is set; this requires that the conflict array be multiported to a degree equivalent to the associativity of the preload array. Two types of conflicts can arise when a store instruction is executed. If the load address and store address were identical or overlap, a *true conflict* has occurred. However, if the two addresses were different, and the conflict resulted from the hashing scheme used, this is termed a *false load-store conflict*.

Thus, bits within the conflict vector can be set in one of three ways: 1) a true conflict; 2) a false load-store conflict resulting from the hashing scheme; or 3) a false load-load conflict resulting from exceeding the set associativity of the preload array. Regardless of the source of the conflict, the hardware must assume it is valid and execute correction code to ensure program correctness. This is accomplished using the check instruction. The format for the check instruction is *check Reg, Label*, where *Reg* is a general purpose register, and *Label* specifies the starting address of the correction code supplied by the compiler. When a check instruction is executed, the conflict bit corresponding to *Reg* is examined. If the conflict bit is set, the processor performs a branch to the correction code marked by *Label*. The correction code

provides for re-execution of the preload and its dependent instructions. A branch instruction at the end of the correction code brings the execution back to the instruction immediately after the check, and normal execution resumes from this point.

The conflict bits are reset in two ways. First, a check instruction resets the conflict bit for register *Reg* as a side effect. Second, any preload that deposits a value into a general purpose register also resets the corresponding conflict bit. The valid bits within the preload array are reset upon execution of the corresponding check instruction, using the pointer within the conflict vector. Note that in the event the flow of control causes the check instruction not to be executed, the preload valid bits will remain set. However, this causes no performance impact because another preload of the destination register must occur before another check instruction can occur, resetting any spurious conflict.

Note that only preloads, stores, and checks have to access the address registers and the conflict vector. Accesses to the preload array are performed using the virtual address to avoid address translation delay. For store instructions, these accesses can be performed as soon as the store address is calculated; it is not necessary to wait until the store data have been computed. For load instructions, MCB accesses are performed in parallel with the data cache access. Because the MCB is very similar to a cache in design and smaller than most caches, it is unlikely that the MCB will affect the processor pipeline timing. However, further study of MCB timing is required within the context of a specific pipeline architecture.

4.1.2 MCB address hashing

Incoming preload and store addresses are used to select a corresponding set in the preload array. The most direct method to select one of n MCB lines is to simply decode $log_2 n$ bits of

the address. However, testing revealed that this approach resulted in a higher rate of *load-load conflicts* than a baseline software hashing approach, most likely due to strided array access patterns causing additional conflicts. As a result, the MCB employs the permutation-based hardware hashing scheme proposed by Chen [1].

Mathematically, the hardware hashing approach can be represented as a binary matrix multiplication problem, where matrix A is a non-singular matrix and $hash\_address = load\_address * A$. For example, consider the following 4x4 A matrix, used to hash 4-bit addresses:

```
1001
0010
1110
0101
```

To mathematically compute the hash address for incoming address 1011, the address is simply multiplied by the matrix, obtaining hash address 0010. If matrix A is non-singular, an effective hash of the incoming address is assured [60]. When mapping this scheme to hardware, each bit in the hash address is simply computed by XORing several of the incoming address bits, corresponding to the 1's in each column of the matrix. Thus $h3$, the most significant bit of the hash address, is the XOR of $a3$ and $a1$ of the incoming address; $h2$ is the XOR of $a1$ and $a0$, etc. This simple hardware scheme provides excellent hashing with only a small cost in time and hardware.

This same hashing approach is used to generate the address signature for incoming preload and store instructions. The signature is hashed in order to reduce the size of the MCB and to speed signature comparison. The signature is stored in the MCB for each preload, and is compared to the signature for incoming store instructions to determine if a *conflict* has occurred.

4.1.3 Handling variable access sizes

Many instruction set architectures allow memory references to have byte addressability and variable access sizes. Thus, there arises the possibility that two memory references could access slightly different addresses, yet actually conflict. For example, the references:

```
load_char R1, 0x40000001

store_int 0x40000000, R1
```

represent a true load-store conflict. Although conflicts such as this are rare, they can occur in real code. An example where this might occur is the *union* construct in *C*. To provide correctness, any code reordering scheme based upon memory disambiguation must account for the possibility of conflicts by memory operations with different access widths. One solution to this problem is to legislate it away; hardware designers can simply declare that accessing the same location with different width instructions is a poor programming practice and decide their hardware will not support it. A more general solution would require that any disambiguation technique provide adequate checks to ensure program correctness in the presence of variable width accesses.

The hashing MCB design proposed by Chen addressed the access width problem by removing the two least significant bits (LSBs) from the address hash and allowing four register numbers to be stored in each location in MCB array. The four register locations in each entry correspond to the different values of the two LSBs. Figure 4.3 shows how this concept is used to solve the access width problem. In Figure 4.3(a), an incoming byte load deposits its destination register, $R1$, in the appropriate slot in the MCB based upon its size and the LSB values of its address. When the subsequent word-size store, shown in Figure 4.3(b), occurs, the store

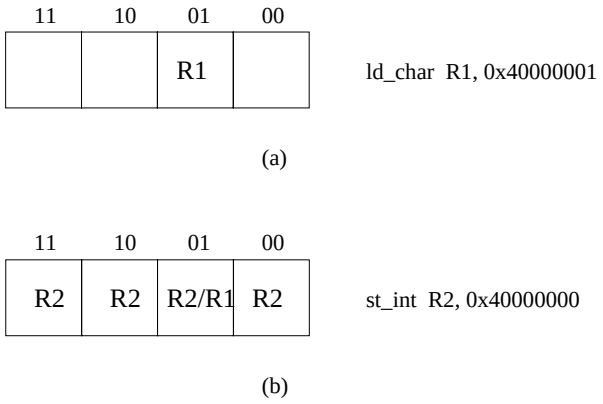


Figure 4.3 Hashing MCB Array Entry.

essentially conflicts with any register in any of the four locations, and the conflict bit associated with $R1$ must be set.

For the set associative design proposed in this thesis, the MCB does not use the three LSBs of preload and store instructions when hashing to select the preload array line corresponding to the memory reference. Instead, these three bits, as well as two bits indicating the access size, are stored within the array for preload instructions. When a store occurs, its five bits are evaluated with the five bits stored for the preload to determine whether a conflict has truly occurred. A simple design for determining conflicts for these two five-bit vectors requires only seven gates and two levels of logic, assuming the architecture enforces aligned memory accesses. Thus, the proposed design provides accurate disambiguation for variable access sizes with a straightforward and less costly approach than Chen's hashing MCB design.

4.1.4 Handling context switches

Whenever a general purpose register must be saved to memory due to context switches, neither the MCB conflict vector nor the preload array must be saved. The only requirement is

for the hardware to set all the conflict bits when the register contents are restored from memory. This simple scheme causes performance penalty only when the context switch occurs after a preload instruction has been executed but prior to the corresponding check instruction. Setting all conflict bits ensures all conflicts that were interrupted by the context switch are honored, but may cause some unnecessary invocations of correction code. The scheme also handles virtual address aliasing across multiple contexts.

4.1.5 Speculative execution

Executing an instruction before knowing that its execution will be effective is termed *speculative execution*. If an instruction is moved above preceding conditional branches prior to resolving their direction, *control speculation* has been performed. The MCB approach applies *data speculation*, in which instructions are executed before knowing whether the data are valid. In particular, a preload and its dependent instructions are executed before knowing if the value loaded by the preload is valid. The execution of these speculative instructions must be corrected if a conflict occurs.

There are two aspects of correcting the execution of speculative instructions. First, the values generated by these instructions must be corrected. The compiler algorithm described in Section 4.2 is responsible for ensuring these values are corrected. Second, the program state must be correctly maintained in the event an exception occurs. Because the value preloaded into the register may not be correct, there is a chance that a flow dependent instruction that uses the preload result may cause an exception which otherwise would not have occurred. In the example in Figure 4.4, taken from [1], if $R1$ equals $R2$, the value 7 is loaded into $R3$ in the original code segment. However, the value 0 may be preloaded into $R3$, in which case the

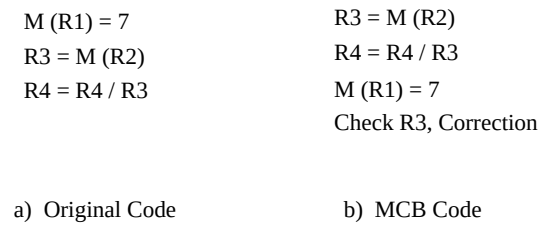


Figure 4.4 Speculative Execution of Excepting Instructions.

divide instruction will cause an exception. Since the exception is due to an incorrect execution sequence, it must be ignored.

One solution is to provide architectural support to suppress the exceptions for speculative instructions [61]. A potential trap-causing instruction executed speculatively should be converted into the non-trapping version of the instruction. Therefore, the exception caused by the divide instruction in the example above would be ignored. However, the exception should be reported if there is no conflict between the preload and the store. Several schemes for precise exception detection and recovery have been proposed [34], [35], [62].

4.1.6 Discussion of hardware requirements

Chen estimated the hardware requirements for a 2-way set associative MCB with 32 sets in CMOS technology to be 60,100 transistors [1]. He also estimated the critical path through the MCB to be 13 gate delays, for both preload and store instructions. The set associative design employed for this thesis would have similar hardware requirements. However, because this design stores only an address signature rather than the entire address, the main MCB array size would be significantly smaller (17 bits per MCB entry versus 35 bits with Chen's design). Scaling Chen's estimates to account for the smaller MCB array, the proposed design would require less than 40,000 transistors.

Because of the similarity of the MCB design to a cache, the MCB size can be estimated in terms of number of cache bytes it is equivalent to. This comparison may be particularly meaningful because the inclusion of MCB hardware into a design could potentially require a corresponding reduction in the size of the on-chip cache. Using this comparison, the main MCB array of a 64-entry MCB (8 sets of 8 entries each) requiring 17 bits per entry would have approximately the same storage requirement as 128 bytes of cache. Although this comparison does not take into account the overhead support logic for the MCB design, Chen's estimates indicate that most of the MCB hardware cost is in the main array.

4.2 Compiler Support

To take full advantage of the MCB hardware support, the compiler must remove ambiguous memory dependences, allowing store/load pairs to be reordered, and insert code to ensure correct execution in the event truly dependent instructions are reordered. The compiler must also take into account the side effects of aggressive code reordering. For example, over-speculating preload instructions can significantly increase register pressure and could result in a loss of performance due to spilling. In this section, the algorithms implemented in the IMPACT C compiler for exploiting the MCB hardware support are discussed.

4.2.1 Basic MCB scheduling algorithm

To expose sufficient instruction-level parallelism to allow effective code scheduling, the compiler must be able to look beyond basic block boundaries. In the IMPACT compiler, basic blocks are coalesced to form *superblocks*, which reflect the most frequently executed paths through the

code (see Section 2.3). The superblock is the basic structure for scheduling in the IMPACT compiler.

The basic MCB scheduling algorithm involves the following steps for each frequently executed superblock:

(1) Build the dependence graph.

(2) Add a check instruction immediately following each load instruction, inserting necessary dependences.

(3) For each load, remove dependences to preceding stores.

(4) Schedule the superblock, removing any unnecessary check instructions.

(5) Insert required correction code.

The initial preparations for code scheduling, including building the dependence graph, are unchanged by the MCB algorithm. After the dependence graph has been built, a check instruction is added after each load instruction in the superblock. The destination register of the load becomes the source operand of the check, making the check instruction flow dependent upon the load. Initially, the correction block of the check is not defined. During code scheduling, the check instruction must maintain correct dependences; thus, it must be dependent upon the load and also inherit some of the load's dependences. Because we want flow dependent instructions of the load to be able to bypass the check, the check inherits only memory and control dependences from the load. Dependences to the previous and subsequent branch instructions are also added to the check instruction to ensure it remains within the load's original basic block. Figures 4.5(a) and 4.5(b) show unscheduled code from the earlier example (with two instructions added to highlight dependences) and the code after the check instruction and its dependences are inserted.

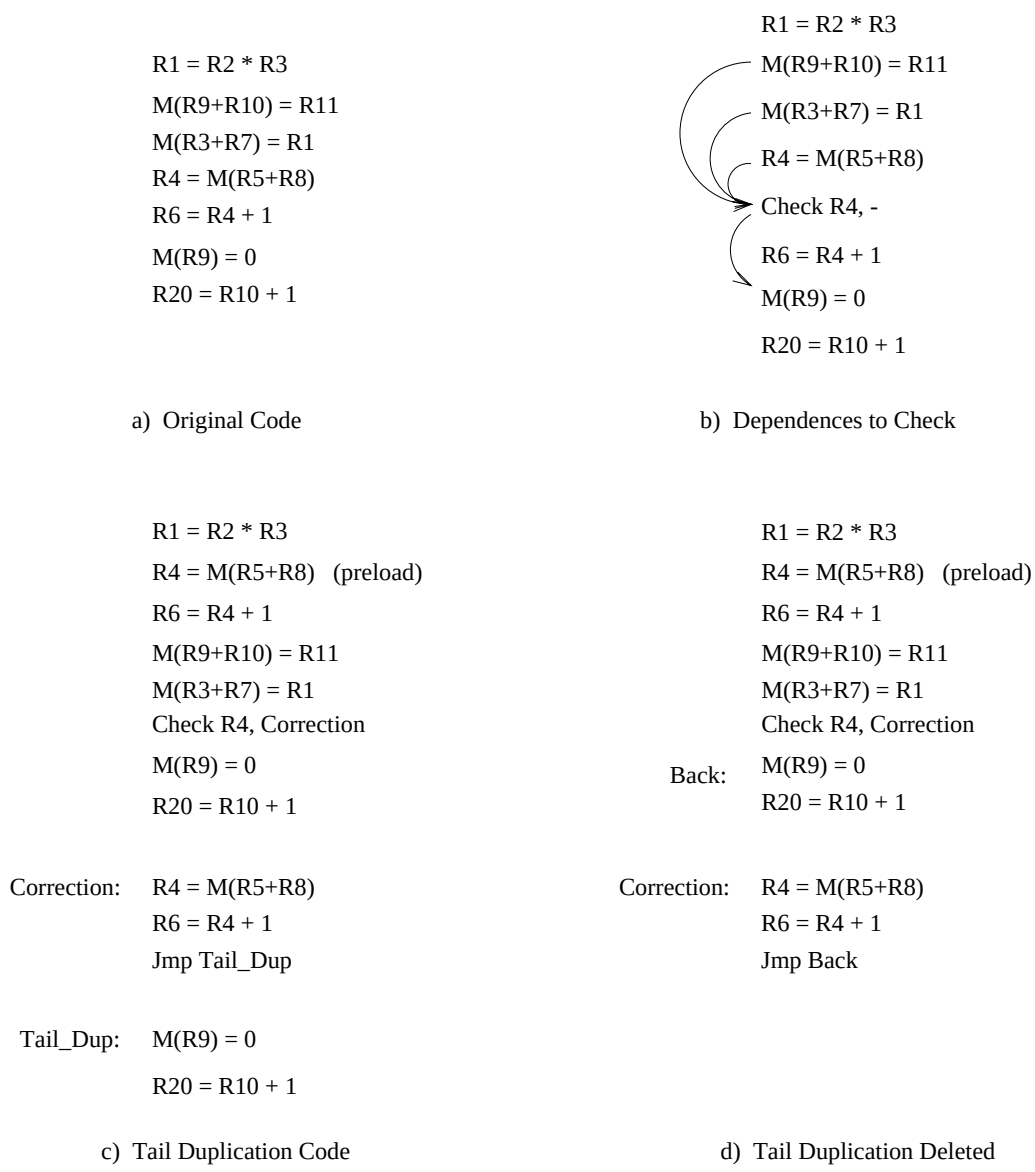


Figure 4.5 MCB Code Compilation.

The next step in MCB scheduling is to remove ambiguous store/load dependences. For each load, the algorithm searches upward, removing any dependence arcs to store instructions not determined to have a definite dependence. Associated with each load, the algorithm maintains a list of store instructions whose dependence has been removed. The algorithm currently only removes dependences to stores which precede the load, i.e., only removes flow dependences. Although nothing prevents dependences to subsequent stores (anti-dependences) from being removed, experience has shown there is little or no benefit from removing these dependences. To limit over-speculation of loads, the algorithm limits the number of store/load dependences which can be removed for each load. If too many dependence arcs are removed, a greedy scheduling algorithm is likely to move the load far ahead of its initial position, needlessly increasing register pressure and the probability of false conflicts in the MCB. Additionally, the algorithm ensures dependences are formed between the load instruction and any subroutine call in the superblock, preventing loads from bypassing subroutine calls. Thus, no MCB information is valid across subroutine calls.

Next, the superblock is scheduled. Each time a load instruction is scheduled, the list of stores associated with the load is examined. If all stores on the list have already been scheduled, the load did not bypass any stores during scheduling, and the associated check instruction can be deleted. The flow dependence between the load and the check ensures the check cannot be scheduled prior to the load; thus deletion of the check (and removal of its dependences) does not impact instructions already scheduled. If it is determined the load has bypassed a store during scheduling, the load is converted to its preload form. In our current implementation, one check instruction is required for each preload instruction. However, multiple check instructions could potentially be coalesced to reduce the execution overhead and code expansion incurred by

the potentially large number of checks. Because the check is a single-operand instruction, extra bits should be available to accommodate a mask field to specify a set of registers which are to be checked by this instruction. For example, if a register bank with 64 registers is partitioned into eight sets of eight registers each, the check instruction would use three bits to specify which bank was being checked and eight bits to specify the register mask. The coalesced check would branch to correction code, which would have to provide correct execution regardless of which preload instruction experienced a conflict. Further research is required to assess the usefulness of coalescing check instructions.

4.2.2 Inserting correction code

The compiler provides correction code for each preload instruction. When a check instruction determines that a conflict has occurred, it branches to the correction code. The correction code re-executes the preload instruction and all dependent instructions up to the point of the check. (In the infrequent case that the load has bypassed a single store, the correction code can replace the re-execution of the preload with a simple move from the store's source register. In fact, the move itself may become unnecessary via forward copy propagation.) The original load instruction will not be a preload within correction code (because its check has already occurred), but any dependent instructions which are preloads must be re-executed as preloads. During insertion of correction code, the compiler must check for any anti-dependences which would overwrite source operands, such that these operands would not be available for execution within the correction code. If anti-dependences are detected, they are removed by virtual register renaming.

Because scheduling is performed on superblocks that do not allow side entrances, the correction code cannot jump back into the superblock after re-executing the required instructions. Instead, the correction code jumps to *tail duplication* code, which is simply a duplicate copy of all superblock instructions subsequent to the check instruction. This tail duplication code (Figure 4.5(c)) ensures all dependences and register live ranges are calculated correctly during register allocation and post-pass scheduling. Following post-pass scheduling, however, the superblock structure is no longer necessary to the compiler and the code can be restructured to allow jumps back into the superblock. At this point, all jumps within the correction code are redirected to jump back into the superblock immediately following the check instruction, and all tail duplication code can be deleted. Thus, the tail duplication code is only a temporary tool used by the compiler to maintain correct dependences and live ranges during register allocation and post-pass scheduling, and is removed prior to final code generation (Figure 4.5(d)).

4.3 Experimental Evaluation

To evaluate the MCB approach, experiments were conducted on a set of twenty-nine benchmark programs, including nine common Unix utility programs, six programs from SPEC-CINT92, and fourteen programs from SPEC-CFP92. Experimental results were obtained using the detailed emulation-driven simulation described in Section 2.4.

Table 4.1 outlines the architecture modeled for these experiments (the *target* architecture) and Table 4.2 shows the instruction latencies used. The instruction latencies used were those of the HP PA-RISC^{TM} 7100. The IMPACT simulator models in detail the architecture's prefetch and issue unit, instruction and data caches, branch target buffer (BTB), and hardware interlocks. This allows the simulator to accurately measure the number of cycles required to

Table 4.1 Simulated Architecture.

Architectural Features
8-issue in-order execution superscalar processor
Extended version of HP PA-RISC instruction set
- Extensions for MCB
- Silent versions of all trapping instructions
64 integer, 64 floating-point registers
Dcache: 64k, direct mapped, non-blocking, 64 byte blocks,
8 cycle miss penalty, write-thru, no write allocate
Icache: 64k, direct mapped, non-blocking, 64 byte blocks,
8 cycle miss penalty
BTB: 1k entries, direct mapped, 2-bit counter,
2 cycle misprediction penalty
MCB support

Table 4.2 Instruction Latencies.

Function	Lat	Function	Lat
Int ALU	1	FP ALU	2
(pre)load	2	FP multiply	2
store	1	FP div(SGL)	8
branch (check)	1	FP div(DBL)	15

execute a program, as well as to provide detailed analysis such as cache hit rates, BTB prediction accuracy, and total MCB true/false conflicts.

4.3.1 MCB emulation

To create an executable file to drive the simulation, the functionality of the MCB must be emulated to allow the code to execute on the *host* architecture, an HP PA-RISC 7100 workstation. Following code scheduling, the code contains preload and check instructions, which are not executable by the host architecture. Thus, the code must be transformed to accurately emulate the MCB code. To accomplish this, the MCB code is modified with explicit

address comparisons similar to that for Nicolau's run-time memory disambiguation. Figure 4.6 illustrates the code changes required to emulate the MCB. Figure 4.6(a) shows the target architecture code which would be simulated, and Figure 4.6(b) shows the code after emulation code has been added. In the emulation code, register $R30$ holds the address of the preload, and $R40$ and $R50$ hold the addresses of the stores. Registers $R45$ and $R55$ are set by explicit comparisons of the load address to the two store addresses. Because the preload instruction has bypassed numerous store instructions, $R35$ is used to record whether any of the stores caused a conflict. Thus, $R35$ is initially zeroed and is subsequently ORed with the results of the address comparisons. The check instruction is emulated with a conditional branch instruction, whose direction is based upon the value of $R35$.

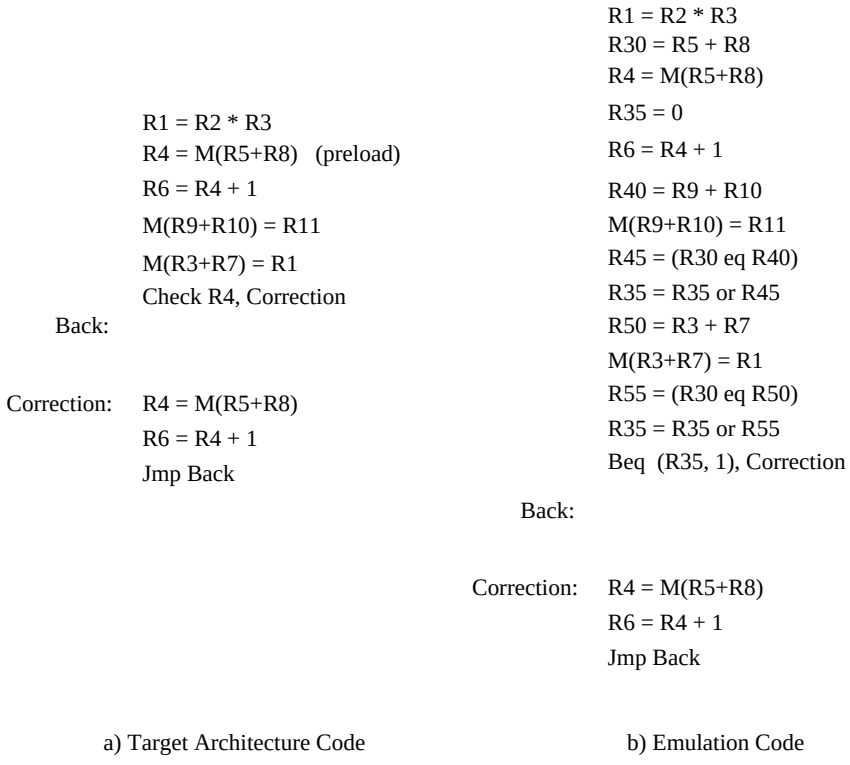


Figure 4.6 MCB Emulation Code.

Note the code growth required in this trivial example to correctly emulate the MCB scheme. In real benchmark code where numerous loads have bypassed numerous stores, overall code growth of 1000% was not uncommon. This clearly indicates the problems that schemes such as run-time disambiguation will have in the presence of aggressive code scheduling and optimization.

4.3.2 MCB size and associativity

The first MCB experiment was to measure MCB performance for various size MCB preload arrays. For this experiment, set associativity and signature field size were held constant (8-way set associativity and five signature bits) while the MCB size was varied from 16 to 128 entries, i.e., 2 to 16 sets. Additionally, performance for the perfect MCB case (i.e., false conflicts never occur) was measured to show asymptotic performance. Figure 4.7 shows the results from the six benchmarks evaluated. These six benchmarks were selected for this experiment because ambiguous memory dependences were shown to be major performance impediments for them in Figure 3.1. Speedup is shown for the MCB 8-issue architecture, relative to a baseline 8-issue architecture with no MCB. For several benchmarks, an MCB size of 32 or 64 entries was sufficient to approach perfect performance. The performance for *056.ear* dropped significantly for sizes below 64 entries, and the performance for *052.alvinn* and *cmp* did not reach asymptotic performance even for a size 128 MCB. This was the result of excessive load-load conflicts caused by multiple variables hashing to the same MCB location.

The results of MCB associativity testing are somewhat compiler-specific and are not shown. For most benchmarks, 8-way set associativity is required to achieve best MCB performance. Two factors heavily influence this need: 1) the IMPACT compiler often unrolls loops up to 8

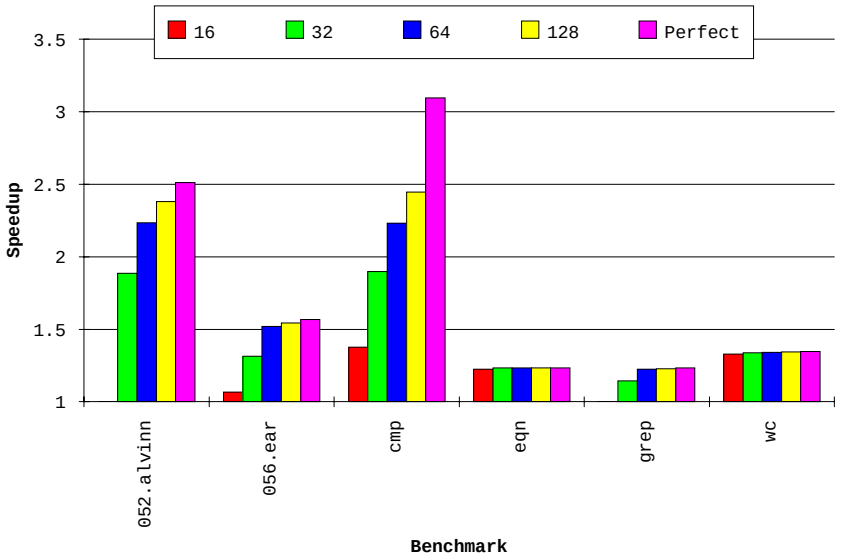


Figure 4.7 MCB Size Evaluation. Speedup of an 8-issue architecture for various size MCBs vs. an 8-issue architecture without MCB (8-way set associative, 5 signature bits).

times; and 2) because the 3 LSBs of the load address are not used during hashing, up to 8 sequential single-byte loads will hash to the same MCB location. Thus, 8-way set associativity is necessary to reduce the number of false load-load conflicts. Even at this associativity, the performance of *cmp* was impacted as a result of load-load conflicts caused by sequential loads and by independent variables hashing to the same location.

4.3.3 Signature field size

To reduce the number of false load-store conflicts, the MCB contains a hashed signature field. The required width of this signature field was evaluated, holding MCB size constant at 64 entries, 8-way set associative. Performance was measured for field sizes of 0, 3, 5, and 7 bits, and performance for a full 32-bit signature is shown for comparison. MCB 8-issue speedup is again shown relative to the baseline architecture. Figure 4.8 shows the results; a signature size of 5 bits approached asymptotic performance of the full signature for all benchmarks. The

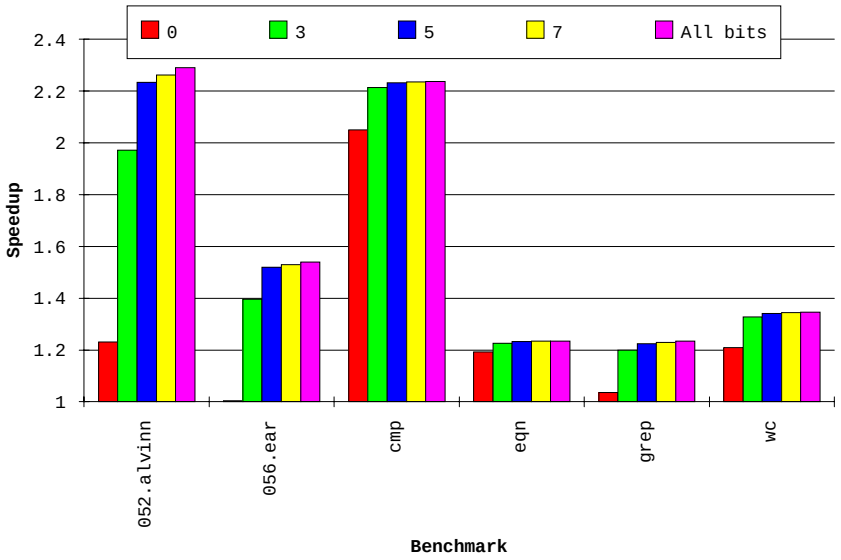


Figure 4.8 MCB Signature Size. Speedup of an 8-issue architecture with various size address signature fields vs. an 8-issue architecture without MCB (8-way set associative, 5 signature bits).

performance for several benchmarks suffered for signature sizes below 5 bits as a result of false load-store conflicts.

4.3.4 MCB performance

In the previous two sections, MCB parameters were varied to determine the best physical configuration. Experiments in this section are measured using a 64 entry, 8-way set associative MCB with 5 signature bits. Results are shown for the suite of 29 integer and floating-point benchmarks.

Integer performance

Figures 4.9 and 4.10 show results for the integer benchmarks. These figures reflect the performance for an 8-issue architecture without MCB (using the existing IMPACT low-level static disambiguation) and for an 8-issue MCB architecture, both relative to a baseline single-

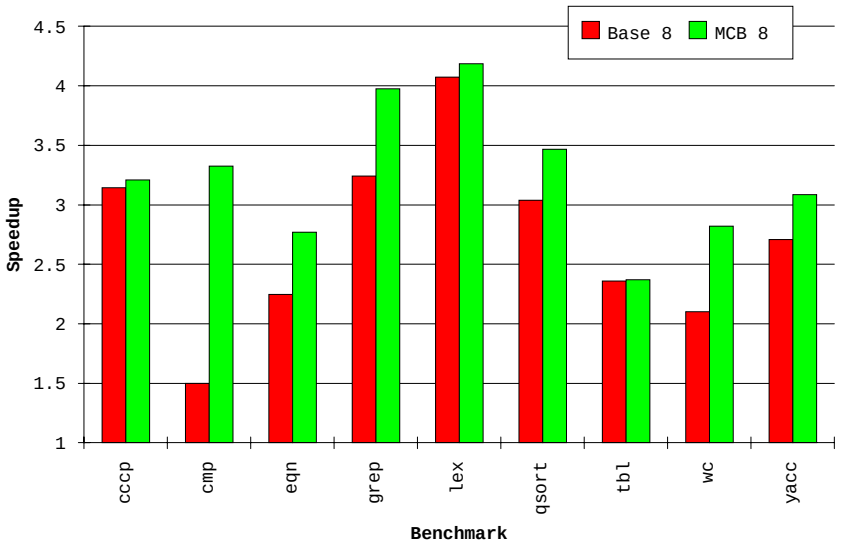


Figure 4.9 Unix MCB 8-Issue Results. Speedup of code compiled with and without MCB over a baseline single-issue architecture.

issue architecture. Figures 4.11 and 4.12 show the performance of the 8-issue MCB architecture relative to the 8-issue architecture without MCB. The MCB architecture showed good ILP for most benchmarks; note that although an 8-issue architecture is being modeled, only 4 integer ALUs are available per cycle. Speedups of more than 2.5 times over the single-issue processor are achieved for 8 of 9 Unix benchmarks and 4 of 6 SPEC-CINT92 benchmarks.

In comparison to the 8-issue architecture without MCB, the MCB architecture provides modest speedup for many of the benchmarks. Note that not all benchmarks are limited in performance by memory dependences, and thus not all benefit from improved memory disambiguation. However, there is a direct correspondence between the benchmarks improved by MCB and those from Figure 3.1; MCB achieved speedup for all benchmarks for which memory disambiguation was a significant impediment to ILP. The benchmark *cmp* achieved the most significant speedup. This is the result of being able to overlap iterations of the unrolled inner loop because of the dependences removed by the MCB approach. Benchmarks such as

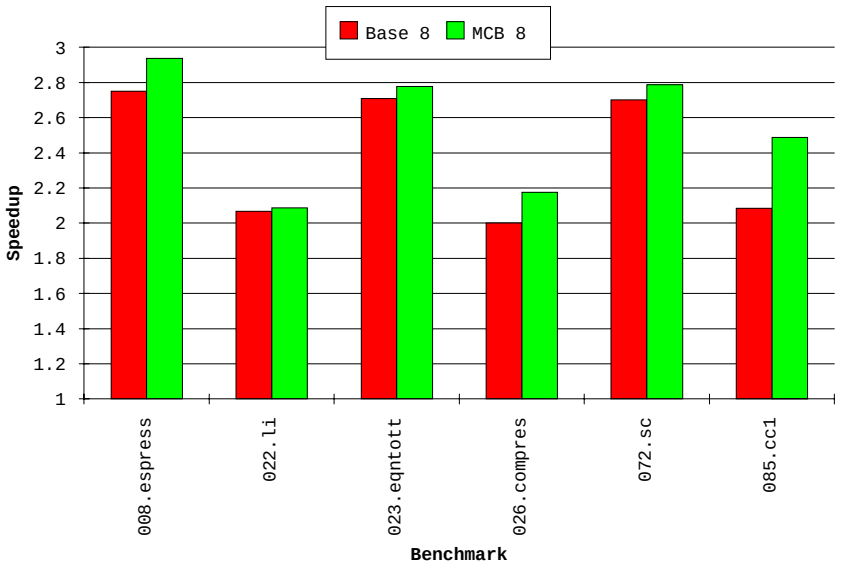


Figure 4.10 SPEC-CINT92 MCB 8-Issue Results. Speedup of code compiled with and without MCB over a baseline single-issue architecture.

---

tbl, *022.li*, *026.eqntott*, and *072.sc* essentially achieved no speedup because the important inner loops contain no store operations. Other benchmarks, such as *cccp* and *lex*, achieved no speedup over the baseline architecture because the existing low-level memory disambiguation already provided good disambiguation. Note that *cccp* and *lex* showed good ILP in Figure 4.9 even for the non-MCB architecture. For several other benchmarks, including *026.compress* and *008.espresso*, MCB performance gains were somewhat masked by cache effects. MCB code suffers slightly more from cache effects because it experiences a greater overall number of cache misses. This increase in cache misses results because MCB's greater scheduling freedom allows more speculative execution of loads above branches; load misses from these speculative loads would not be experienced in less aggressively scheduled code.

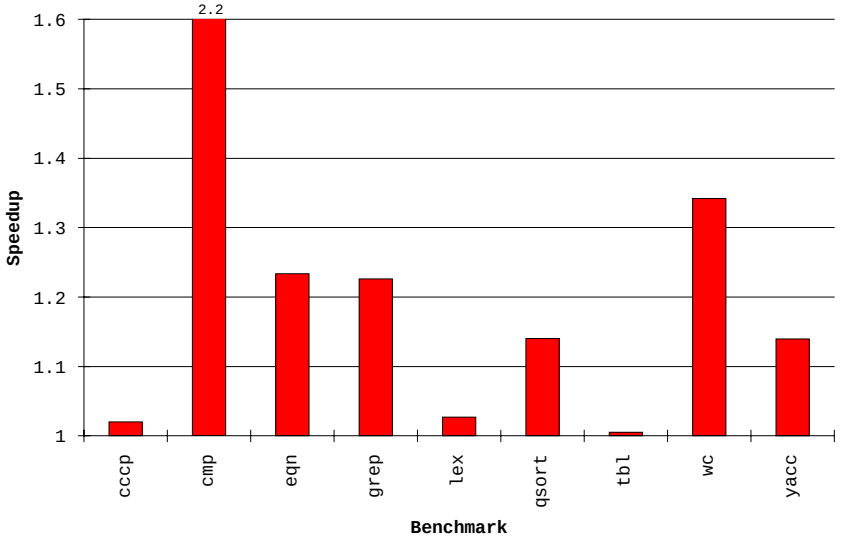


Figure 4.11 Unix MCB 8-Issue Results. Speedup of code compiled with MCB over an 8-issue architecture without MCB.

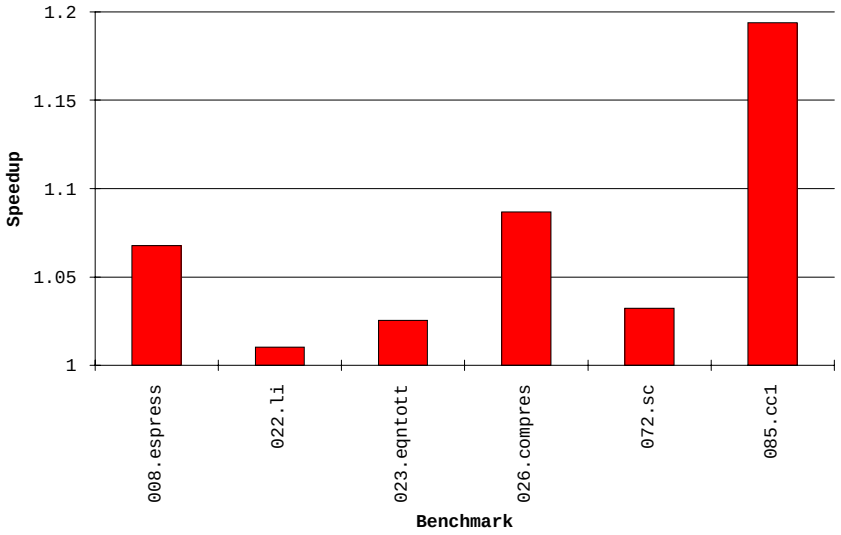


Figure 4.12 SPEC-CINT92 MCB 8-Issue Results. Speedup of code compiled with MCB over an 8-issue architecture without MCB.

Floating-point performance

Figures 4.13 and 4.14 show results for the floating-point benchmarks. Figure 4.13 shows the performance of 8-issue architectures with and without MCB compared to a single-issue architecture. Figure 4.14 presents the speedup of the 8-issue MCB architecture over the 8-issue architecture without MCB. There are several interesting points to note about these floating-point results. First, the performance of the 8-issue architecture without MCB is in general poor. This result is not surprising since floating-point benchmarks are usually dominated by array accesses that are relatively difficult to disambiguate using only information available within the low-level code (i.e., without interprocedural analysis or source-level information). Second, note that the performance with MCB is significantly improved compared to that for the 8-issue without MCB. The results indicate that memory disambiguation is a more severe impediment to ILP for floating-point code, which tends to have larger basic blocks and highly predictable branches. For most benchmarks, the MCB technique provides significant speedup. Exceptions are *015.doduc* and *089.su2cor*. The heavily executed blocks of *015.doduc* are not significantly hindered by ambiguous memory dependences; either they do not contain store operations or the existing stores do not significantly limit ILP. The performance of *089.su2cor* was severely degraded due to false load-load conflicts.

A third item to note is that the overall ILP achieved for the MCB architecture is relatively low, exceeding 2.5 times speedup over that for the single-issue architecture for only 4 of the 14 benchmarks. This is less speedup than was achieved for the integer benchmarks, even though the floating point benchmarks would be expected to be more amenable to ILP transformations due to their larger basic blocks and predictable branches. The conflict statistics presented in the next section demonstrate that this poor speedup is the result of excessive conflicts experienced

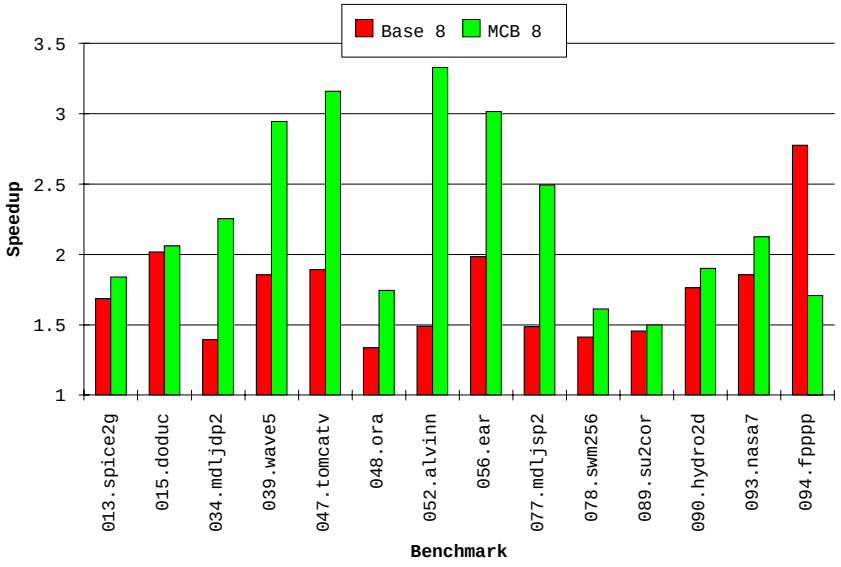


Figure 4.13 SPEC-CFP92 MCB 8-Issue Results. Speedup of code compiled with and without MCB over a baseline single-issue architecture.

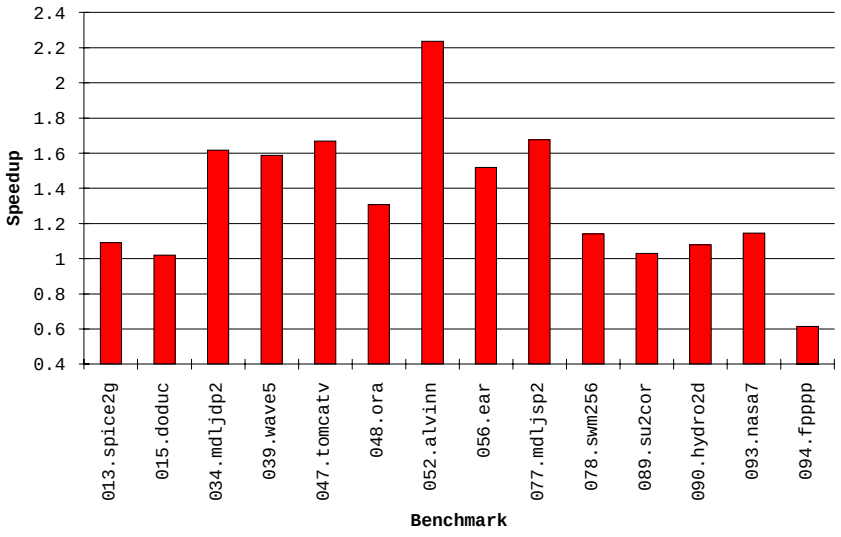


Figure 4.14 SPEC-CFP92 MCB 8-Issue Results. Speedup of code compiled with MCB over an 8-issue architecture without MCB.

by the MCB. (A later section will address reducing these conflicts.) A final item to note is the large slowdown experienced by *094.fpppp* using MCB. This is also the result of conflicts experienced by the MCB code.

MCB statistics

Table 4.3 shows the conflict statistics for the 8-issue MCB architecture, using the MCB configuration from the previous section. The second column shows the total dynamic check instructions executed, followed by the number of true conflicts, false load-load conflicts, and false load-store conflicts. The final column shows the percentage of dynamic check instructions which branched to correction code. For most integer benchmarks, the percentage of time the correction code is executed is very low; the exceptions were *cmp yacc*, *008.espresso*, and *085.cc1*. With the exception of *cmp*, performance degradation due to the execution of correction code was only about 2%-3% for these benchmarks, compared to a perfect MCB without false conflicts. Note that for all benchmarks except *008.espresso* and *085.cc1*, false conflicts were the primary cause of taken checks. These false conflicts were primarily false load-load conflicts. False load-store conflicts posed no significant problem for the integer benchmarks, indicating that the 5-bit signature field used was successful at limiting these conflicts.

The conflict statistics for floating-point benchmarks were significantly worse, with several of the benchmarks experiencing over 10% taken check instructions. This high level of taken checks corresponds to significant performance degradation. For many of the benchmarks (*013.spice2g6*, *015.doduc*, *034.mdljdp2*, *039.wave5*, *077.mdljsp2*, and *093.nasa7*) true conflicts were a primary factor in the high number of taken checks. This points to the need for better static disambiguation to detect store/load pairs which are likely to conflict, so the compiler can avoid reordering

Table 4.3 MCB Conflict Statistics (8-issue architecture, 64 entries, 8-way set associative, 5 signature bits).

Benchmark	Total Checks	True Confs	False Ld-Ld Confs	False Ld-St Confs	% Checks Taken
cccp	23.5K	0	0	42	0.18
cmp	1087K	0	55.1K	605	5.12
eqn	793K	0	0	667	0.08
grep	96.3K	0	0	395	0.41
lex	47.3K	0	0	16	0.03
qsort	785K	0	0	1586	0.20
tbl	7203	0	0	9	0.12
wc	306K	0	0	406	0.13
yacc	211K	56	15.8K	403	7.52
008.espresso	324K	5262	8779	735	4.56
022.li	224K	0	0	422	0.19
023.eqntott	32.1K	0	7	33	0.12
026.compress	160K	0	0	817	0.51
072.sc	216K	0	0	364	0.17
085.cc1	705K	24.4K	14.9K	4182	6.16
013.spice2g6	4312K	128K	187	24.3K	3.53
015.doduc	336K	8166	1182	10693	5.96
034.mdljdp2	3230K	74.6K	3218	43.6K	3.76
039.wave5	6178K	212K	167K	43.0K	6.14
047.tomcatv	639K	0	0	24.7K	3.87
048.ora	3534K	0	0	133K	3.76
052.alvinn	10.8M	0	178K	44.0K	2.05
056.ear	23.1M	0	119.8K	74.6K	0.84
077.mdljsp2	3080K	73.5K	5367	31.1K	3.57
078.swm256	39.8M	0	6210K	497K	16.85
089.su2cor	13.2M	0	2086K	187K	17.22
090.hydro2d	9334K	0	727K	209K	10.03
093.nasa7	21.3M	1130K	717K	629K	11.63
094.fpppp	4516K	1104	726K	18.5K	16.51

them. Both load-load and load-store false conflicts also contributed significantly to the high number of total conflicts. This high number of false conflicts indicates the size of the MCB preload array may have to be reconsidered for floating-point benchmarks; this will be explored further in the next section.

Table 4.4 shows the effect of the MCB compiler techniques on the static and dynamic code size, again using an 8-issue architecture with a 64-entry, 8-way set associative, 5 signature-bit configuration. The addition of MCB code increased the static code size an average of 8.6% across the integer benchmarks. The integer benchmarks that showed the worst static code expansion were the very small benchmarks (*cmp* and *wc*), in which the addition of a small number of check instructions and correction code to the most-frequently executed blocks made a significant change in the static code size. For the floating-point benchmarks, average static code growth was 12.7%. This indicates the MCB transformation was applied more frequently to the floating-point benchmarks than to the integer benchmarks, due to larger superblocks and the greater percentage of loads in floating-point code (integer code typically has more scalar variables which reside in register and do not require load instructions).

Note that the MCB code transformations resulted in a significant increase in the dynamic number of instructions executed for most benchmarks, particularly the floating-point benchmarks. This increase in dynamic instructions is primarily the result of greater speculation freedom afforded by the improved memory disambiguation and by the additional code which must be executed when conflicts occur. However, the greater scheduling freedom allowed by MCB was in general able to pack this increased number of instructions into a tighter schedule and achieve speedup for many of the benchmarks.

Table 4.4 MCB Static and Dynamic Code Size (8-issue architecture, 64 entries, 8-way set associative, 5 signature bits).

Benchmark	% Static Instruction Increase	% Dynamic Instruction Increase
cccp	0.7	0.0
cmp	47.2	40.5
eqn	2.2	5.7
grep	4.3	9.9
lex	0.7	0.4
qsort	14.7	13.3
tbl	0.3	0.2
wc	23.5	21.5
yacc	5.3	2.3
008.espresso	3.8	8.3
022.li	1.3	3.6
023.eqntott	8.6	0.2
026.compress	13.1	11.3
072.sc	1.5	1.4
085.cc1	1.8	7.4
013.spice2g6	3.0	3.7
015.doduc	6.9	1.6
034.mdljdp2	11.1	20.1
039.wave5	10.4	21.5
047.tomcatv	5.8	8.5
048.ora	3.9	16.7
052.alvinn	22.0	30.5
056.ear	11.4	16.5
077.mdljsp2	9.5	18.8
078.swm256	21.2	35.9
089.su2cor	15.5	31.8
090.hydro2d	12.2	21.4
093.nasa7	22.3	34.6
094.fpppp	22.0	9.7

4.3.5 Reducing MCB conflicts

In the previous section, conflict statistics indicate the set associative MCB design results in a relatively higher number of both false and true conflicts for floating-point code than for integer code. This higher number of conflicts can result in significant performance degradation. To reduce the number of true conflicts, better static memory disambiguation is needed to detect store/load pairs which are likely to conflict. Another possible technique to reduce the number of true conflicts is to use memory profiling to dynamically determine instructions which are likely to conflict.

The high number of false conflicts indicates the size of the MCB preload array chosen for the earlier experiments, although good for integer code, may not have been optimal for floating-point code. Although a 64-entry MCB appears to work well for integer code, a larger MCB may be required to reduce false conflicts for floating-point code. [1] The MCB size experiment (shown in Figure 4.7) was repeated using four of the floating-point benchmarks which experienced a high number of false conflicts in the previous experiment. Figure 4.15 shows the results of this experiment. For each benchmark, the different bars on the graph reflect the performance improvement for various size MCB architectures over a baseline 8-issue architecture without MCB.

For two of the benchmarks (*047.tomcatv* and *093.nasa7*), varying the size of the MCB resulted in only minor variations in performance. For these benchmarks, the 64-entry MCB used for earlier experiments provides nearly the same performance as for the perfect MCB.

[1] Increasing the size of the preload array will reduce the frequency of both false load-load and load-store conflicts. Although the number of false load-store conflicts is primarily affected by the size of the signature field, increasing the size of the preload array also reduces these conflicts because fewer signature comparisons are required.

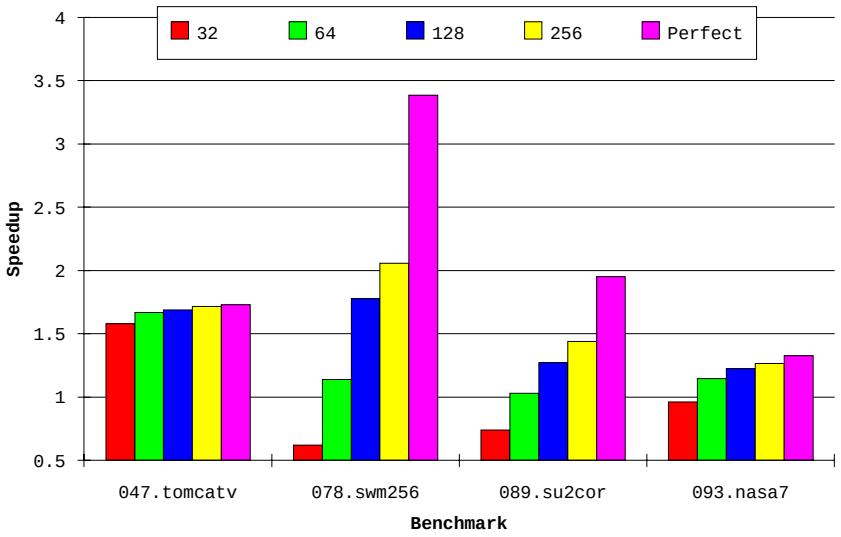


Figure 4.15 Floating-Point MCB Size Evaluation. Speedup of an 8-issue architecture for various size MCBs vs. an 8-issue architecture without MCB (8-way set associative, 5 signature bits).

However, for the other two benchmarks (*078.swm256* and *089.su2cor*), varying the size of the MCB resulted in dramatic changes in performance. For both of these benchmarks, even a 256-entry MCB provided significantly less performance than for the perfect MCB.

An experiment was performed to quantify the effect on performance of varying MCB size, using all 14 floating-point benchmarks. Figure 4.16 shows the results of this experiment. The leftmost bar for each benchmark represents the results for the 64-entry set associative design presented previously. The middle bar shows the results for a 256-entry set associative MCB design. The rightmost bar represents a perfect MCB, which experiences no false conflicts.

For several benchmarks, the 256-entry MCB performed significantly better than the 64-entry MCB. Additionally, it provided comparable performance to that for the perfect MCB for the majority of benchmarks. However, as seen in Figure 4.15, even a 256-entry MCB was not large enough to provide performance comparable to that for the perfect MCB for

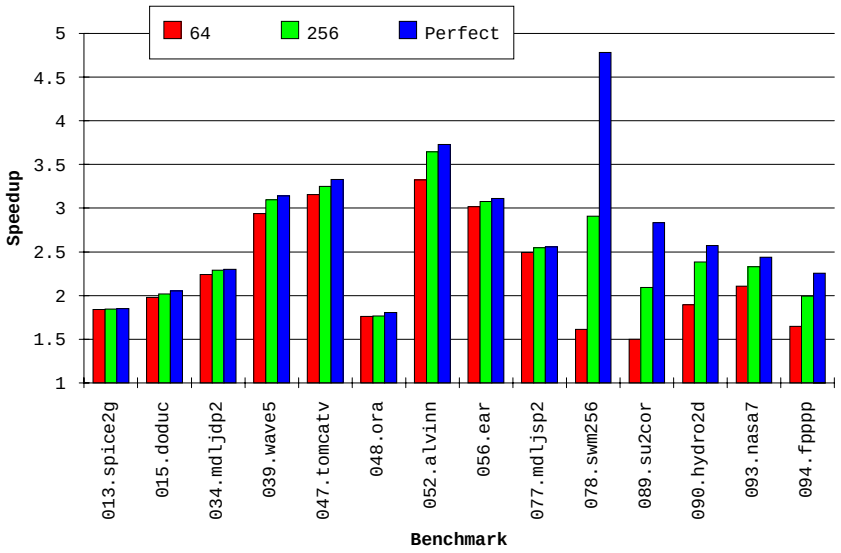


Figure 4.16 8-Issue Results for Different MCB Models.

benchmarks such as *078.swm256* and *089.su2cor*. Rather than reflecting a problem in the set associative design, this performance degradation is the result of limitations in the current compiler technology being used to support MCB. In particular, the list scheduling algorithm being used is "greedy," and tends to overschedule the code. When the scheduler is used to produce MCB code, no cost is assigned to moving a load above an ambiguous store; rather, the dependence is simply removed and the scheduler is free to schedule the memory operations in any order. Because the scheduler is greedy, load operations tend to move unnecessarily early in the schedule and unnecessary preloads/checks are generated. Because floating-point code has a high percentage of load operations, the number of preloads generated tends to overwhelm MCB hardware resources, causing a high number of false conflicts. Scheduling techniques are being investigated to reduce this overscheduling of load operations. It is believed that improved scheduling with a 256-entry set associative MCB will provide comparable performance to the perfect MCB.

4.4 MCB Summary

In this chapter, a combined hardware and compiler approach for dynamic memory disambiguation has been evaluated. Using detailed simulation, MCB is shown to obtain substantial speedup for many of the integer benchmarks evaluated. Results for the floating-point benchmarks demonstrated significantly greater performance benefit, despite limitations in the current compiler technology.

The MCB, or any other memory disambiguation approach, is not a panacea that will provide speedup for all programs. For some programs, control transfer instructions remain the primary bottleneck, and ambiguous dependences are not a significant problem. However, test results demonstrate that MCB provides substantial speedup for those programs whose ILP is limited by ambiguous memory dependences.

CHAPTER 5

STATIC MEMORY DISAMBIGUATION USING SYNC ARCS

In the previous chapter, a dynamic memory disambiguation approach was investigated and found to provide good performance improvement. In this chapter, a new technique to provide improved static memory disambiguation for low-level optimization and scheduling is explored.

In most compilers, static memory disambiguation for low-level code uses only information available within the low-level code (i.e., no source-level information), and is able to achieve limited success. Although it may be clear within the source code that two memory accesses cannot access the same location, this information is often lost when the code is converted to low-level form. Figure 5.1 shows an example of this problem, from the inner loop of the benchmark *wc*. In Figure 5.1(a), the code to get a character from the file buffer and increment a global variable is shown. Using typing information, it can be determined that the various memory references are to different fields of the *fp* structure, and to a buffer pointed to by the *ptr* field of the structure. (The variables *charct* and *c* are scalar and will reside in register.) Each of these memory references are clearly disambiguous with other references. However, the dependence relationship between these references is less clear in the low-level code in Figure 5.1(b). In particular, the pointer dereference in *op*69 (corresponding to $(fp) \rightarrow ptr$) cannot easily be disambiguated from the stores in the block. As a result, important optimizations are prevented. Thus, improved static memory disambiguation for low-level code is much easier to accomplish with some visibility to source-level information.

```
for ( ; ; )
  {
    c = (--(fp)->cnt  <  0    ?  filbuf(fp)  :
          (int) *(fp)->__ptr++);
    if ( c == -1)
       break;
    charct++;
  }
```

(a) Original source code segment

```
cb 12
  op59 ld_i  r101, r3 + 0
  op60  add  r102, r101, 1
  op61  st_i  r3 + 0, r102
  op64  blt  r101, i,  cb48
  op65  ld_i  r103, r3 + 4
  op67 add_u  r104, r103, 1
  op68  st_i  r3 + 4, r104
  op69  ld_uc  r4, r103 + 0
  op71  beq  r4, -1, cb24
  op72 ld_i  r105, r74 + 0
  op74  add  r106, r105, 1
  op75  st_i  r74 + 0, r106
```

(b) Lcode segment

Figure 5.1 Difficulty of Memory Disambiguation for Low-Level Code.

In the next section, the relative merits of different approaches for providing source-level information to the low-level code are discussed. This is followed by a detailed explanation of the proposed technique. Experimental results for the proposed technique are provided in Chapter 7.

5.1 Providing Source Information to the Intermediate Code

In general, source-level information can be passed to the low-level code in two ways: (1) maintain some source code information within the low-level IR, and perform dependence analysis on the low-level code; or (2) perform dependence analysis at the source code level, and pass explicit dependence information to the low-level code. Each of these approaches has advantages and disadvantages.

5.1.1 Performing static analysis on low-level code

Passing raw source information to the low-level code, either embedded within the low-level code or through an external file, would allow accurate dependence analysis to be performed on the low-level code. The primary advantage of this approach is that dependence analysis can be re-accomplished at any point during compilation. If code transformations invalidate the dependence analysis, the availability of the source information allows the dependence analysis to be re-performed after the transformations for use by later stages of compilation. If desired, the results of dependence analysis can be discarded after use, and re-generated when needed again.

However, this approach has several drawbacks. First, the magnitude of the raw source information which must be maintained is unclear. To perform dependence analysis for scientific code which relies heavily on arrays, perhaps simply maintaining source-level array index information would be sufficient to provide good static disambiguation. It is likely, however, that accurate dependence analysis for pointers would require much more source information. To allow an in-depth interprocedural alias analysis would essentially require visibility to the entire source code. Maintaining the complete source information within the low-level IR would be extremely expensive in terms of memory requirements.

A second drawback of maintaining source information within the low-level form is the difficulty of maintaining that information through code transformations. For example, if a loop is unrolled to expose ILP, the source code for this loop would also have to be transformed to maintain an accurate representation for performing dependence analysis.

A third argument against performing dependence analysis on the low-level code is the expense. Although having the source information within the low-level code makes re-analysis

possible, the time required for this analysis probably makes repeated analyses impractical. Although interprocedural techniques for pointer analysis are becoming more and more powerful, they require an extremely high compile-time investment. Finally, performing the analysis on the low-level IR does not remove the need to perform an incremental update of the dependence information. Because it is impractical to re-perform dependence analysis after each transformation, the dependence information will have to be maintained through most transformations, requiring at least the same level of effort as the alternate approach discussed in the next section.

5.1.2 Performing static analysis on source-level code

The second approach for providing source-level help to support low-level memory disambiguation is to perform an in-depth analysis once at the source level and then to maintain this dependence information throughout subsequent compilation. The primary advantage of this approach is that the analysis is performed when the required information is most available. The analysis must only be done once, and its results are available for use by later stages of compilation. There is no requirement to maintain any source-level information within the low-level IR.

This approach also has several potential difficulties. First, the amount of dependence information which must be maintained can be extremely large. Techniques would be needed to limit the number of explicit dependence arcs being maintained. Second, because the analysis cannot be re-accomplished, the dependence information must be maintained through all code transformations. It has not previously been shown that this can be done without significant loss of accuracy. It is important to note, however, that although maintaining the dependence information may prove quite difficult, the approach described in Section 5.1.1 faces the same

problem. Regardless of the approach used, some incremental analysis will have to be performed due to the expense of dependence analysis.

5.2 Sync Arcs

After extensive discussions within the IMPACT research group and with industry, the second approach described above was chosen for this thesis. The proposed approach is to perform dependence analysis once at the source code level, to accurately maintain this information throughout subsequent compilation, and then to use this information to facilitate low-level optimization and scheduling. The explicit dependence information being passed down is called synchronization arcs or *sync arcs*. Although currently only memory dependence information is being propagated to the low-level code, sync arcs could also be used to represent any required ordering or "synchronization" between operations. For the sync arc approach to be successfully applied, the following issues must be addressed:

- The necessary dependence information must be identified.
- The dependence information must be extracted from the source level and propagated to the low-level code.
- The dependence information must be maintained through low-level code transformations.
- The amount of dependence data being maintained must be controlled
- A mechanism for using the dependence information within the low-level code must be developed

These issues are discussed in subsequent sections.

5.2.1 Desired dependence information

Static memory disambiguation has most often been applied to source-level loop transformations. These transformations typically are inhibited by any type of memory dependence

within the loop. Thus, very accurate dependence analysis is required; even a single ambiguous dependence can prevent an important transformation. However, when applying static memory dependence to low-level code optimization and scheduling, the uses of the disambiguation information are different, and may require a different degree of accuracy. It is enlightening to understand how improved memory disambiguation would be used during the low-level stages of ILP compilation.

Because the IMPACT compiler employs aggressive ILP compilation techniques, its use of memory disambiguation for low-level code should be representative. In general, IMPACT employs memory disambiguation to support three areas of compilation: acyclic scheduling, cyclic scheduling (software pipelining), and optimization.

During acyclic scheduling, the ability to reorder two memory operations is based upon whether a dependence exists between the two operations during a single execution of the block being scheduled. For example, if the superblock being scheduled consists of a single iteration of a loop, then two memory operations can be reordered if they never reference the same memory location during any single iteration of the loop. The array references

```
A[i] = x

   y = A[i+1]
```

can be freely reordered because they never reference the same element of the array during any single iteration. Thus, during acyclic scheduling, operations can be reordered unless a *non-loop carried* or intra-iteration dependence exists between them.

For cyclic scheduling, more accurate dependence information is required. In addition to knowing whether a non-loop carried dependence exists between memory operations, cyclic scheduling has to know whether a *loop carried* or inter-iteration dependence exists. In the

example above, the read of $A[i+1]$ will reference the same location as the store to $A[i]$ which will occur during the next iteration (assuming i is the loop induction variable). Further, cyclic scheduling has to know the *dependence distance* between two memory operations, which is the number of iterations from when one memory operation references a particular location until when the other operation references the same location. Therefore, in the above example, an anti dependence of distance 1 exists from the load to the store. Non-loop carried dependences are sometimes referred to as distance 0 dependences.

Thus, from looking at the requirements of acyclic and cyclic scheduling, it can be seen that sync arcs should represent whether the dependence is non-loop carried and/or loop carried, and what the dependence distance is. However, as discussed in Chapter 3, in some cases array dependence analysis may fail to clearly establish the dependence relationship between memory operations. When this occurs, the sync arc must be able to specify that the dependence distance is unknown.

Memory disambiguation is also critical for supporting ILP optimizations. Perhaps the most important optimizations requiring disambiguation are loop unrolling, loop invariant code removal, and redundant load/store removal.

The loop unrolling optimization requires similar dependence information to cyclic scheduling. Figure 5.2 illustrates the dependence information required for loop unrolling. The source code and low-level code for a simple pair of array references are shown in Figures 5.2(a) and (b). Assuming this code is in a loop whose induction variable is i, there is a distance 2 flow dependence from the store to the load. To allow unrolling, no special dependence information is required. However, to accurately preserve the correct dependences (as shown in Figure 5.2(c)) the optimizer has to properly understand the dependence distance. Without understanding of

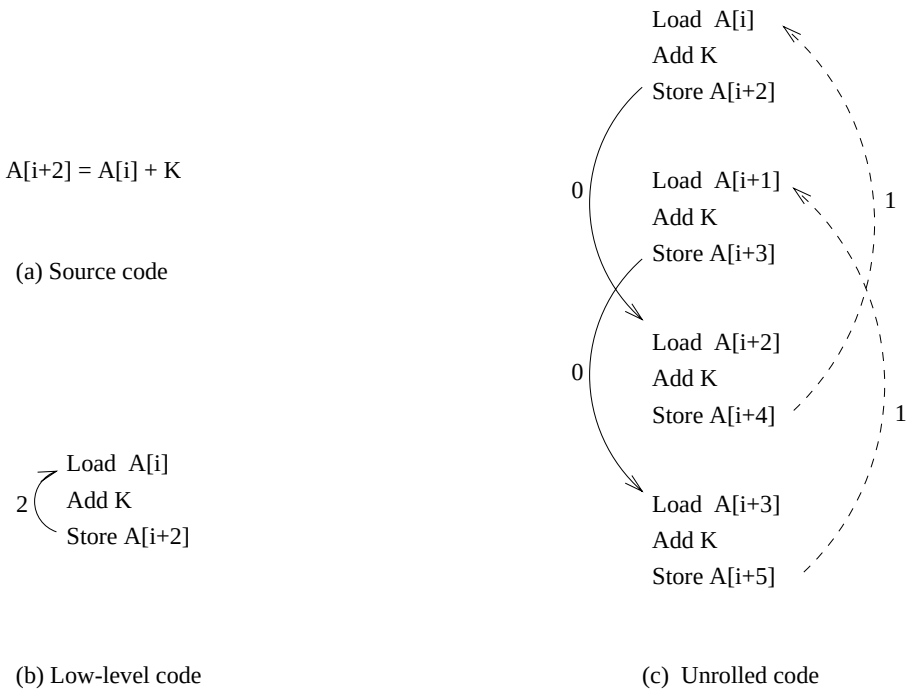
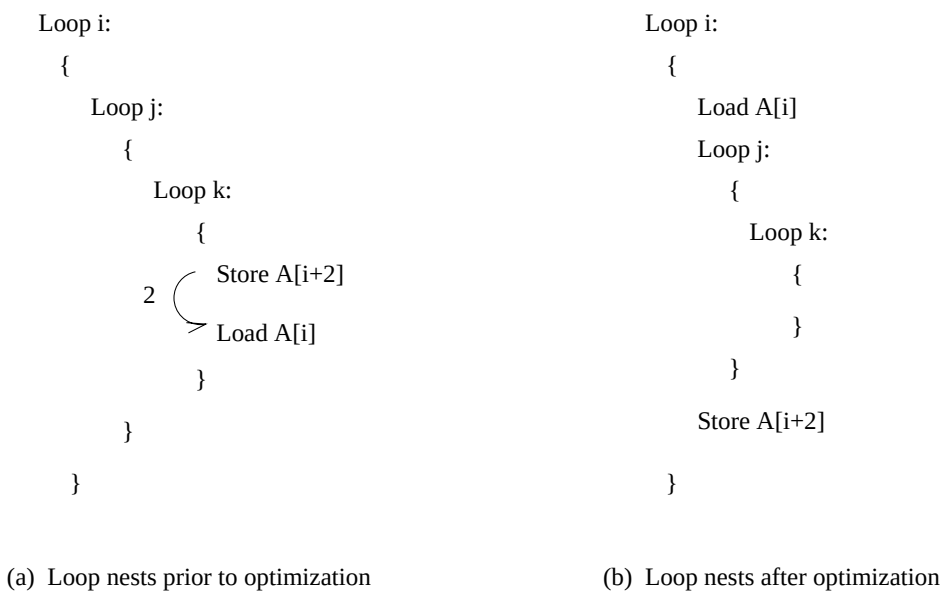


Figure 5.2 Dependence Information to Support Loop Unrolling.

dependence distance, the optimizer would have to assume dependences between all load/store pairs in the loop, severely restricting subsequent code scheduling.

Another characteristic of the dependence relationship between two memory operations which must be understood is which loop carries the dependence. Figure 5.3 illustrates this characteristic, as applied to the loop invariant code removal optimization. There is a distance 2 flow dependence from the store to the load; however, note that the inner loop induction variable is k and that the array references are based upon the outer loop variable i. Thus, for all iterations of both the j and k loops, the load and store operations reference invariant locations and can legally be moved outside these loops as shown in Figure 5.3(b). However, this optimization can only be performed if the dependence information contains information indicating which loop carries the dependence.



(a) Loop nests prior to optimization

(b) Loop nests after optimization

Figure 5.3 Dependence Information to Support Loop Invariant Code Removal.

Two other characteristics of dependence that a static analysis might have to determine are frequency and certainty. Figure 5.4 illustrates the need for these characteristics, using the example of redundant load elimination. The redundant load elimination optimization will attempt to delete the second load, if it is truly redundant. For the optimization to be valid, the second load must always reference the same memory locations as the first load every time this section of code is executed. Thus, the certainty of the dependence between the loads must be definite (i.e., the static analysis did not conservatively add this dependence because it could not ascertain the true dependence relationship) and the frequency must be always. Additionally, for the optimization to be valid, there can be no intervening store operations which might possibly reference the same address as the loads. Even a possible dependence between the store and either load, which holds for even one execution, is sufficient to prevent the optimization. Sync arcs, then, need some mechanism for recording how sure the dependence is, and how frequently during execution it might occur. Another possible use of the dependence frequency information

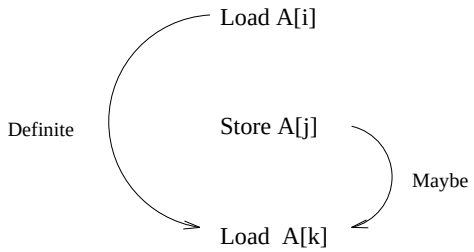


Figure 5.4 Dependence Information to Support Redundant Load Elimination.

would be the MCB technique examined in the previous chapter. Although it is a dynamic technique, it relies on some basic level of static analysis to prevent reordering operations which are frequently dependent. If the static analysis could determine that a possible dependence occurs between two memory operations, but that this dependence occurs only infrequently, a dynamic technique such as MCB might be able to exploit this to reorder instructions which are infrequently dependent.

One other item to note from this redundant load elimination example is that the dependence between the two loads is an input dependence. Although it is obvious that sync arcs would want to carry dependence information for flow, anti, and output dependences, this demonstrates a need to maintain input dependences as well.

A final type of dependence information which might be useful (although not exploited in the current implementation) is the concept of how dependence varies from iteration to iteration. In a previous example, the references $A[i]$ and $A[i+1]$ were said to have a distance 1 dependence, which holds for all iterations of the loop. For other code, however, the dependence distance may vary between different loop iterations. For example, consider the code example in Figure 5.5. The references $A[i]$ and $A[N-i]$ have a dependence distance which varies between loop iterations. However, note that for this reference pair there will be a distance 0 (non-loop

```
for (i=o; i<N; i++)  {
    A[i] = A[N - i];
}
```

Figure 5.5 Single Iteration Dependence Example.

```
for (i=o; i<SIZE; i++)  {
   if (flags[i]) {
      prime = i + i + 3;
      for (k=i +prime; k<=SIZE; k+=prime)
         flags[k] = FALSE;
      count++;
   }
}
```

Figure 5.6 Threshold Dependence Distance Example.

carried) dependence for at most one iteration of the loop. If this iteration can be identified to the low-level code, then potentially the low-level stages of the compiler could take advantage of this knowledge to perform code transformations such as loop splitting to further expose ILP.

Another situation in which information on how the dependence varies might be useful is if the dependence distance varies, but is always greater than some threshold. Figure 5.6 shows an example of this from the *siev* benchmark. The references to $flags[i]$ and $flags[k]$ have a definite dependence between them; during the first iteration of the outer loop the dependence distance is 3, and then is greater than 3 for all subsequent iterations. The capability to indicate a dependence threshold such as this within the sync arc structure would allow subsequent optimizations such as unrolling to more accurately represent the true dependence.

Table 5.1 summarizes the dependence information which would be useful for the sync arc data structure to maintain.

Table 5.1 Desired Dependence Information.

Dependence type (flow, anti, output, input)
Dependence distance (known/unknown)
Whether dependence is non-loop, inner loop, or outer loop carried
Certainty of dependence (definite, maybe)
Frequency of dependence (always, sometimes, rarely)
How dependence varies (constant, single iteration, threshold)

5.2.2 Extracting sync arcs

The IMPACT compiler performs its source-level analysis within the Pcode module. To support the high-level analyses, transformations, and optimizations performed within the Pcode intermediate representation, detailed data dependence analysis is done. Prior to the work in this thesis, the data dependence analysis within Pcode was limited to analysis of Fortran programs which have been translated from Fortran to C using the *f2c* tool. Because the programs being analyzed were originally Fortran code, the data dependence analysis was able to make numerous simplifying assumptions, particularly in regard to aliasing and the use of pointers. Extensions to the existing dependence analysis allowing it to analyze C programs are presented in Chapter 6.

With the availability of accurate source-level dependence arcs, extracting sync arcs is a relatively simple task. The dependence information must be associated with the corresponding expression within the IR which will eventually be translated into a load or store operation. When the Pcode module produces output code in the next IR representation (Hcode), it incorporates the dependence information into the output file associated with the appropriate expression. In the IMPACT compiler, the next lower IR is Hcode, which also represents expressions hierarchically. To pass the dependence information through Hcode, it is attached to the expressions using *expression pragmas*, a comment-like structure employed by the IMPACT

Pointer to dependent oper - 32 bits
Dependence distance - 16 bits
Dependence info - 16 bits
- Certainty of dependence - 1 bit
- Frequency of dependence - 2 bits
- Flags - 13 bits
-- Outer, inner, non-loop carried
-- distance know/unknown
-- threshold dependence
-- single iteration dependence

Figure 5.7 Sync Arc Format.

compiler. When Hcode is translated into the low-level (Lcode) data structure, the expression pragmas are converted into Lcode's sync arc representation.

Within Lcode, sync arcs are maintained as fields within the internal structure of individual instructions. Each instruction has pointers both to the sync arcs for which it is the source (head) of the dependence and to the arcs for which it is the destination (tail). In the current implementation, each sync arc requires two words of data. Figure 5.7 illustrates the internal Lcode format for sync arcs. One word contains a pointer to the dependent instruction, i.e., the instruction at the other end of the sync arc. The second word is divided into bit fields storing the dependence distance and various characteristics of the arc.

Note that the current implementation uses only three single-bit flags to indicate whether the dependence is outer, inner, or non-loop carried. This simple representation can result in some minor loss of accuracy when memory operations are moved into an outer loop nest. Using the current implementation, when a memory operation whose dependence is outer loop carried is moved into an outer loop nest, the dependence must conservatively be considered inner loop carried. Loss of accuracy results because the dependence information does not record which loop nest actually carries the dependence, but rather merely records whether it is an inner

or outer loop. Although this loss of accuracy is considered minor because the two dependent operations probably are now in different blocks of the code (and thus an overly conservative dependence between them is not likely to result in performance loss), this accuracy loss could be avoided by maintaining information in the sync arc structure that records which loop carries the dependence. This could be done by maintaining a *distance vector* such as is used in source-level dependence analysis, or by simply recording the nesting level of the loop which actually carries the dependence.

5.2.3 Maintaining sync arcs

Two concerns with passing explicit dependence arcs from the source level to the low-level code are: 1) can the information be maintained as the code is transformed by optimizations; and 2) can the amount of data being passed down to the low-level code be limited so that the approach is feasible. In this section, the first issue is discussed; the second issue is presented in the following section.

One limitation of the sync arc approach is that dependence analysis cannot be re-accomplished if the dependence information is lost or becomes less accurate as the result of code transformations. For the approach to be viable, then, sync arcs must be able to be accurately maintained through transformations. As part of this thesis, changes were made throughout the backend of the IMPACT compiler to ensure that all ILP transformations properly maintain the sync arcs. With the exception of inlining, discussed below, none of IMPACT's current suite of optimizations posed a significant problem for maintaining the sync arcs, and the results published in Chapter 7 reflect compilation using all optimizations. In the remainder of this section, some of the interesting issues in maintaining sync arcs are discussed.

The first code optimization which poses difficulty for sync arcs is inlining, which is currently performed on the Hcode IR. Because the Hcode stage of compilation is subsequent to Pcode, sync arcs are already in the IR when inlining would normally be performed. Figure 5.8 illustrates the problem presented by inlining. The pseudo-code on the left side of the figure represents a function which contains a call to Function A. The arcs between loads and stores represent the memory dependence information. If we wish to inline Function A into this function, we are not able to determine the dependence relationship between memory operations from the original function and operations from Function A. The sync arcs accurately represent the dependences within each of the individual functions, but do not provide information between operations that were originally in different functions.

One solution to this problem is to assume that all inlined memory operations are implicitly dependent on all memory operations in the calling function. However, this solution to a great degree defeats the purpose of the inlining. These conservative dependences would restrict much of the ability to execute instructions from the inlined function in parallel with instructions from the calling function. Another potential solution would be to perform some limited dependence analysis as part of the inlining, to reduce the unnecessary dependences which must be added. However, this is counter to the purpose of sync arcs, which was to make source-level analysis available to low-level code; this solution would attempt analysis on critical sections of code without the benefit of the source information. The chosen solution was to alter the phase ordering of IMPACT compilation, so that inlining is accomplished prior to Pcode dependence analysis. This solution entails a complete re-implementation of IMPACT's inliner at the Pcode level, and was beyond the scope of this thesis. Results reported in this thesis were obtained without the benefit of inlining.

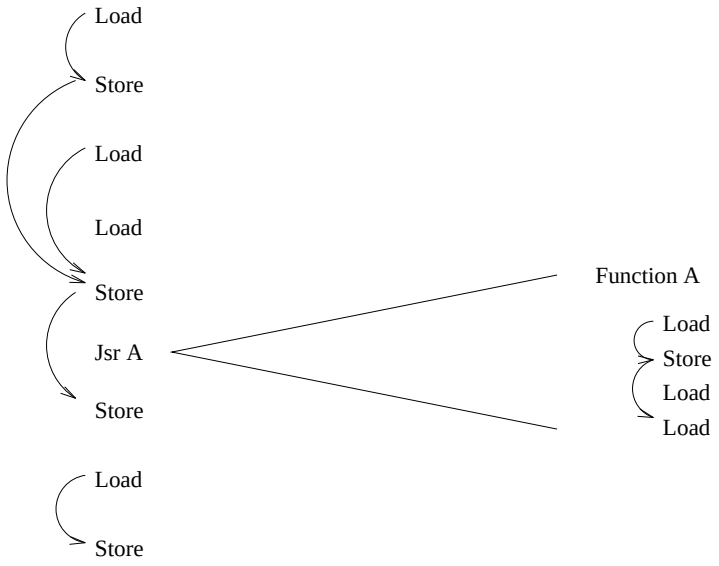


Figure 5.8 Inlining Code with Sync Arcs.

Another important issue regarding sync arcs is whether the dependences can be maintained when the memory operations are moved around, either within the same loop nest or outside the loop nest. Maintaining dependence information for code motion within a loop is trivial. If two memory operations are legally reordered by a code transformation such as acyclic scheduling, no characteristic of the dependence is altered. For example, if a loop-carried flow dependence exists from a store operation to a subsequent load, and scheduling reorders the operations, the dependence will still be a flow dependence with the same dependence distance.

Code motion across loop boundaries is a slightly more complicated transformation for sync arcs. Figure 5.9 shows an example of code motion across loop boundaries as the result of the loop invariant code removal optimization. In this example, the dependence from the load to $Store2$ is assumed to be carried by an outer loop, allowing the load to be moved outside of $LoopC$. The sync arcs must then be updated based upon this code motion. The basic rule employed for determining whether the sync arc must be updated is whether the nearest common enclosing

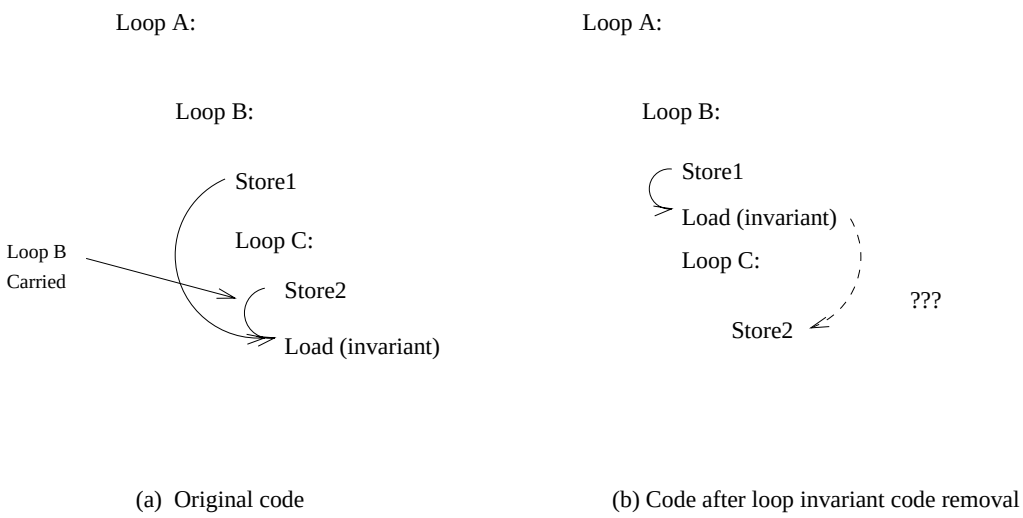
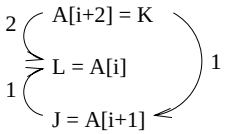


Figure 5.9 Updating Sync Arcs for Code Motion.

loop for the two operations has changed. In the case of the dependence between the load and $Store1$, the nearest common enclosing loop before and after the code motion is $LoopB$. Because the enclosing loop has not changed, the characteristics of the dependence haven't changed, and the sync arc requires no update. However, for the dependence between the load and $Store2$, the nearest common enclosing loop has changed from $LoopC$ to $LoopB$. In this case, a dependence that was previously outer loop carried is now inner loop carried and the sync arc data structure must be updated to reflect this. Note that the loop which carries the dependence has not changed, but this loop is now the inner loop rather than the outer loop with respect to the dependence pair. The loop which carries the dependence is not changed by any IMPACT low-level optimizations.

Perhaps the most interesting optimization requiring sync arcs to be updated is loop unrolling. As discussed above, one of the strengths of sync arcs is that the dependence distance information allows loops to be unrolled so as to accurately maintain only necessary dependences. If this can be done without loss of accuracy, it can significantly increase available ILP.



(a) Loop body

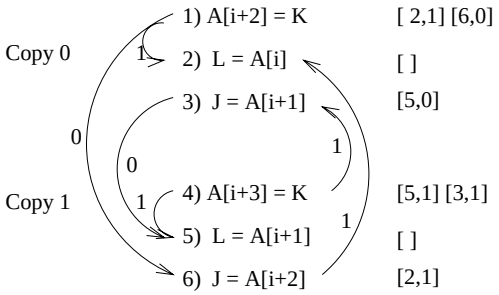
1) A[i+2] = K [2,2] [3,1]
2) L = A[i] [ ]
3) J = A[i+1] [2,1]

(b) Loop body w/ sync arcs

New dist = (Source_copy # + dist) / unrolls

Dest_copy # = (Source_copy # + dist) % unrolls

(c) Unrolling formulas



(d) Unrolled loop body

Figure 5.10 Updating Sync Arcs for Loop Unrolling.

A simple method of updating sync arcs for unrolling has been developed which provides full accuracy. Figure 5.10 demonstrates this technique.

In Figure 5.10(a), a loop body is shown, with arcs which represent the memory dependences and dependence distances between the operations. Figure 5.10(b) shows the same body and a numeric representation of the sync arc information for this loop for the dependences. This numeric representation reflects the arcs which go *from* the operation; the destination operation number and the distance are shown. For example, the first operation has a sync arc *to* operation 2, of distance 2, and a sync arc to operation 3 of distance 1.

To update sync arcs for unrolling, the two formulas shown in Figure 5.10(c) are used. To calculate what operation should be the destination of the updated arc, we simply use modulo

arithmetic based upon which copy of the loop contains the head of the arc and the original dependence distance. The new distance is calculated using integer division on the same two values.

Figure 5.10(d) shows the loop body after being unrolled. The updated dependence arcs and sync arcs are also shown. To illustrate how the formulas are applied, consider the original sync arcs for the first operation, which is now in *Copy*0 of the body. The original arc went to Operation 2, with distance two. Applying the formulas results in

```
New_dist =  ( 0 + 2 ) / 2 = 1

Dest_copy = ( 0 + 2 ) % 2 = 0.
```

Thus, the arc is now a distance 1 arc, going to the copy of Operation 2 located in *Copy*0 of the loop. Likewise, applying the formulas to the arc from Operation 1 to Operation 3, of distance 1, results in a new arc to *Copy*1 of distance 0. Because the copy of Operation 3 located in *Copy*1 is Operation 6, an arc is placed from Operation 1 to Operation 6 of distance 0. Using the formulas to update all the arcs in the unrolled loop body results in an accurate update of the sync arcs, adding only the required arcs.

5.2.4 Limiting the number of sync arcs

The second issue which affects sync arc viability is whether the number of explicit dependences which must be represented by sync arcs restricts application of the technique. Certain ILP optimizations such as superblock formation and loop unrolling, which create multiple versions of the same operation, exacerbate the potential problem. If the number of sync arcs is a problem, what techniques can be developed to reduce the number of sync arcs down to a "reasonable" level?

A suite of 29 benchmarks was compiled using sync arcs, and subjected to IMPACT's most aggressive ILP optimizations. Although the size of the files which store the intermediate form between compilation stages grew significantly and the compilation was slowed to a limited extent due to the presence of sync arcs, all benchmarks compiled without extreme difficulty. Results indicate, however, that techniques to limit the number of explicit sync arcs being represented would be desirable. In the remainder of this section, several techniques for limiting the number of sync arcs are proposed.

The first technique for limiting sync arcs is to eliminate unnecessary dependences. The current sync arc implementation is address based, such that two memory references will have a dependence placed between them regardless of the flow of control. This approach requires a simple dependence analysis to generate the sync arcs and produces sync arcs in a format to easily interface to low-level dependence routines. However, it results in unnecessary transitive dependences. The alternate approach is to perform a flow-based analysis, so that an arc is only placed between two memory references if the value accessed in one could be the same value accessed by the other. This approach increases the complexity of dependence analysis, but reduces the number of arcs which must be maintained. Figure 5.11 illustrates the two approaches. In Figure 5.11(a), the dependences for simple scalar variables are shown. The address-based approach would require a transitive sync arc from the first store to the load to be present in the code, while the flow-based approach would remove this arc. The flow-based approach would also remove non-loop based dependences between memory references on opposite paths of control, such as in an *if-then-else* statement. However, in the case of array dependences, which are not inherently transitive, the flow-based approach may be no more effective than the address-based approach. In Figure 5.11(b), the transitive arc cannot

Figure 5.11 Address-Based Versus Flow-Based Analysis.

be removed without loss of accuracy. For non-scientific code not dominated by array variables, the number of sync arcs could be significantly reduced by eliminating the scalar transitive dependences.

A second possibility for limiting the number of sync arcs is to mark certain references as having implicit dependences. References for which the dependence analysis is unable to provide any reasonable disambiguation could be marked *sync_all* to indicate that implicit dependences exist to all other memory operations in the function. However, it is unlikely that this approach can be used widely without significant loss of accuracy. Even very difficult references to analyze (e.g., pointers to dynamically allocated memory) can usually be disambiguated to some extent (e.g., the pointer could be disambiguated from a statically declared variable). As a result, this technique for limiting sync arcs may not be appropriate for general use, but may be useful in conjunction with one of the following approaches.

Another approach to limiting the number of required sync arcs is to mark explicit regions within the code, indicating a barrier across which all memory references are assumed to have an implicit dependence. Most low-level code transformations which use memory dependence information operate on some limited region of the code (e.g., a loop nest). While it is critical to have accurate disambiguation between operations within the region being optimized, depen-

dences across region boundaries may be of little importance. Thus, by inserting explicit barriers around important regions and assuming implicit dependences across these barriers, many sync arcs could be eliminated. During low-level code transformations, any code motion across a barrier would require either additional barriers to be inserted to protect the moved operations or the moved operations to be marked as *sync_all*.

A final approach to limiting the number of arcs is to assume barriers across which all dependences are implicit. Rather than inserting explicit barriers, the code structure would imply the barriers. For example, loop nests could be assumed as an implicit dependence region. Again, code motion across these implicit barriers would require the moved operations to be marked as *sync_all*.

For the IMPACT compiler, the most promising of these approaches is to specify explicit dependence regions. A new region-based compilation paradigm is being explored [63], in which strongly connected sections of the code (e.g., loops) are grouped into compilation regions. Each region is then processed through various stages of compilation as a separate compilation unit. Because code transformations are localized within each region, it is of low importance to have accurate disambiguation for memory reference pairs which are located in separate regions. Therefore, assuming implicit dependences across region boundaries would have little effect on performance, but would significantly reduce the number of sync arcs.

5.2.5 Using sync arcs

Because sync arcs represent memory dependences explicitly, applying them to optimization and scheduling is straightforward. Lcode modules use sync arcs in one of two ways. The first way to use sync arcs is to answer explicit queries regarding the dependence relationship between two

operations. The query specifies what type of dependence relationship is of interest to the caller (i.e., non-loop carried, inner-loop carried, etc.). For example, the loop invariant code removal optimization, which requires no inner-loop or non-loop carried dependences, provides a mask field specifying these particular dependence characteristics as part of the dependence query. The second way sync arcs are used is for building Lcode's internal dependence graph, which is used primarily by code scheduling. The dependence graph maintains arcs corresponding to register, control, and memory dependences. To obtain the memory dependences for this graph, the sync arc data structures are queried, using a mask specifying only non-loop carried dependences.

5.3 Sync Arc Summary

In this chapter, a method for providing improved static memory disambiguation to support low-level code optimization and scheduling has been proposed. After an analysis of possible methods for providing improved memory disambiguation, the sync arc approach was chosen. This approach performs static dependence analysis on the source-level intermediate representation and then preserves the results of this analysis in the form of explicit dependence arcs which are passed to the low-level code. The data requirements for sync arcs were discussed, and two issues regarding the viability of sync arcs were explored. Successful implementation of sync arcs demonstrates that sync arcs can indeed be accurately preserved through aggressive code transformations. Several methods were proposed for limiting the number of sync arcs which must be maintained. The sync arc approach was tested and shown to be viable across a diverse benchmark suite. The results of this testing, provided in Chapter 7, demonstrate that

the sync arc approach is successful in significantly improving the existing low-level memory disambiguation.

CHAPTER 6

C DEPENDENCE ANALYSIS TO GENERATE SYNC ARCS

In the previous chapter, the sync arc approach was proposed for providing explicit dependence information to support low-level optimization and scheduling. For Fortran programs, strong source-level dependence analysis was available prior to the work in this thesis within the IMPACT compiler to provide the dependence information required to support sync arcs. However, dependence analysis for C programs was not supported.

The C language provides interesting dependence analysis challenges not present in Fortran. The biggest challenge is posed by the availability of pointers within C, which exacerbate the *aliasing* problem. Aliasing occurs as the result of pointer assignments, when the same memory location can be accessed using different access names. To provide reasonable accuracy for aliasing, interprocedural alias and side effect analysis is required. In this chapter, the support added to the IMPACT compiler to allow accurate source-level C dependence analysis to support sync arcs is discussed. First, the changes to the existing dependence analysis required to support C semantics are presented. This is followed by a description of the interprocedural analysis proposed and implemented for this thesis.

6.1 Dependence Analysis for C Programs

Although the IMPACT compiler is primarily a C compiler, it is also able to handle programs originally written in Fortran by using the *f2c* translation tool. Because most of the scientific

benchmarks used for testing source-level transformations are written in Fortran, IMPACT's source-level dependence analysis was developed to analyze only Fortran benchmarks. Although the existing analysis understood most C syntax, it assumed that the code being compiled contained only Fortran semantics. For example, the analysis did not consider the possibility of aliases between global variables. In this section, the modifications to the existing analysis required to handle C semantics are described. An overview of the interesting semantic differences between C and Fortran are presented, followed by a discussion of some of the implementation details.

6.1.1 Semantic differences

Aliasing

The most obvious difference between Fortran and C semantics is the availability of pointers in C. Pointers pose numerous challenges to static dependence analysis. Certainly the biggest challenge is to accurately and efficiently deal with the aliasing problem.

In C, the pointer assignment, $ptr = \&A$, forms an alias between ptr and the variable A. Additionally, dereferences of the two variables are also aliased. For example, if A is a structure, the two references:

```
ptr->field = y;

A.field = z;
```

store to the same location. To ensure correctness, the dependence analysis must be able to accurately handle these aliases.

In the case of global pointer variables, it is possible for an alias to be created in one function, and to hold (i.e., remain valid) in another function. In the example above, if both ptr and A are

global variables, the alias created by the assignment $ptr = \&A$ would hold in other functions executed subsequent to the assignment. Thus, without interprocedural analysis, any global pointer referenced in a function must be assumed to be aliased to all other global variables within the function.

Another difficulty of pointers is that they facilitate the use of dynamically allocated data structures, such as linked lists. Dependence analysis for statically allocated data structures, sometimes referred to as *named* objects, is somewhat easier than for dynamically allocated structures (*unnamed* objects), because when aliases to static objects are formed, the name of the object is often visible. In contrast, dynamically allocated objects have no name, and are only accessed through dereferences of a pointer. Researchers have found disambiguation of recursively defined structures (e.g., linked lists and binary trees) particularly difficult due to the problem of determining whether these structures contain cycles [56], [57], [58]. For example, in the following code segment:

```
while (ptr != NULL) {
    ptr->count = ptr->next->count + 4;
    ptr = ptr->next;
}
```

it is difficult to disambiguate the references to $ptr \rightarrow count$ and $ptr \rightarrow next \rightarrow count$ without knowledge of whether the data structure being accessed contains cycles.

Pointer arithmetic

Another challenge resulting from pointers is dealing with pointer arithmetic, and its duality with array accesses. In Fortran semantics, it is relatively straightforward to determine the relationship of two array accesses such as

```
A[i] = y;
A[i+1] = z; .
```

Because the location of the array is static, the dependence relationship of these two accesses is fixed unless the array index i is changed between the accesses. However, using C pointers the dependence relationship is also dependent upon the pointer not being modified. Consider the following C version of the above array accesses.

```
p = &A[0];
p[i] = y;
p[i+1] = z;
```

In this code, the functionality is the same as for the previous code segment, but the dependence analysis is more complex. Not only must the analysis determine if the index variable i has changed between the accesses, but it must also detect if the pointer itself has been modified. The dependence analysis must also be able to deal with the duality between pointer arithmetic and array accesses. For example, the following three accesses refer to the same location in memory.

```
*(p+i+1) = x;
p[i+1] = y;
++p[i] = z;
```

Function call semantics

In Fortran, function calls pass arguments *by-reference*. Rather than passing the value of a variable to the called function, the address of the variable is passed. Thus, procedure binding

creates aliases between the function call arguments and the called function formal parameters. In contrast, C passes scalar variables *by-value*, and no aliases are formed for scalars. However, for non-scalars such as arrays and structures, C passes by-reference, and binding creates aliases. Dependence analysis for C must be able to distinguish between these cases and correctly create procedure binding aliases.

Additionally, the semantics of standard Fortran do not allow aliases between function arguments. Thus, the dependence analysis can make the assumption that no aliases exist between the formal parameters to a function. For C, however, language semantics enforce no such restriction, and interprocedural analysis is required to determine the dependence relationship between formal parameters.

Unions

Another source of aliasing within C is the *union* data structure. Unions allow different names and types to be assigned to the same memory location. Unlike structures, in which different fields are independent, aliases exist between all fields within a union. Unions can be particularly problematic to low-level dependence analyses, which often assume that memory references of different types or sizes are independent.

6.1.2 Required modifications to existing dependence analysis

In the previous section, some of the important semantic differences between C and Fortran which impact dependence analysis were discussed. In this section, the modifications to IMPACT's existing dependence analysis (which previously assumed Fortran semantics) required for C dependence analysis are overviewed. The intent is to examine a few of the interesting

issues involved in the modifications rather than to provide full implementation details. Those interested in details of the implementation should refer to [3].

Access table structure

Pcode dependence analysis is performed intraprocedurally on each function. The first step in the analysis is to build an *access table*, containing all of the variable references within the function, indexed by the variable name. Within each entry in the table, a linked list of the individual references based upon the variable name is maintained. For programs compiled from Fortran, this approach worked well for two reasons. First, references based on a particular variable in general referred to the same data structure. Whether an array was referenced as the entire array (e.g., A) or as a particular element of the array (e.g., $A[i]$), the same data structure is being referenced. Therefore, it was convenient to group both references under the same access table entry. A second reason that indexing the access table by variable worked well is that Fortran-semantic access expressions contain only a single access. Whether the expression is A, $A[i][j]$, or $A.B.C$, only one memory reference is involved.

In contrast, C dependence analysis requires analysis of arbitrarily complex access expressions. For example, the following is a single access expression from the benchmark *085.gcc*.

```
(((((insn)->fld[3].rtx))->fld[0].rtvec->elem[i].rtx))->fld[0].rtx
```

The expression contains numerous separate accesses (i.e., will generate numerous memory references) to several different data structures. To efficiently represent complex expressions such as this, the access table format required modification.

Consider a simpler expression containing several accesses, such as $**ptr$. If ptr is assumed to be a global variable (i.e., its value does not reside in a register), this expression contains

three separate accesses: ptr, $*ptr$, and $**ptr$, each of which references a different location. Grouping these three accesses into the same entry in the access table is inefficient, because they reference totally separate structures. Instead, the Pcode access table was modified so that separate entries exist for expressions which access different locations. Therefore, ptr, $*ptr$, and $**ptr$ are each placed in separate entries in the access table.

Entries in the access table are indexed by a unique name derived from the access expression. In general, access expressions are connected by either the star (*), arrow ($\rightarrow$), dot (.), or index ([ ]) operators. Because the pointer deference such as $ptr \rightarrow field$ is equivalent to $(*ptr).field$, the arrow operator can be represented as a combined star and dot. Also, because of the duality of pointer and array references, the index expression can also be represented with the star.

Table 6.1 shows a list of some C expressions and the corresponding access table entry name by which the expression is indexed. Note that operators are placed in the entry name according to the order in which the expression is evaluated, e.g., the expression $**p$ requires p to be accessed prior to dereferencing it. This also corresponds to the bottom-up order in which the operators would appear in a parse-tree representation of the corresponding expression. This ordering ensures expressions such as $A[i].B$ and $A.B[i]$, which reference different structures, are placed in separate access table entries. Note also that the access $A[i][j]$ could receive different entry names, based upon how it was declared. If declared as a 2-dimensional array, then the expression corresponds to a single dereference of the variable A ($A*$). However, if the variable were declared as a pointer or array of pointers, the expression indicates two dereferences of A ($A**$).

This naming convention is also useful for identifying aliases between entries in the access table. Consider the statement

Table 6.1 Access Table Names.

C Expressions	Access Name
p	p
*p	p*
**p	p**
p→field	p*.field
A[i]	A*
A[i][j]	A* or A**
A.B	A.B
A[i].B	A*.B
A.B[i]	A.B*

```
p = &A;
```

which forms an alias between p and A. An alias is also created for other expression pairs such $p{\rightarrow}field$ and $A.field$. The access table naming makes the addition of these aliases straightforward. Additional aliases can be added by looking for names with common suffixes appended to the originally aliased pair. Therefore, aliasing the access table names $p*$ and A implies that names $p*.field$ and $A.field$ also alias.

Duality of pointer and array references

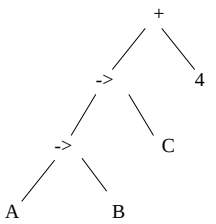
As discussed above, elements of an array can accessed using array notation or pointer arithmetic. To facilitate symbolic analysis of the access expressions, the two forms are treated uniformly by dependence analysis. Each access in the access table maintains an index expression field. The expression $A[i]$ is maintained in the access table under the name $A*$, with the index expression i. Likewise, the expression $*(p+i)$ is stored under the name $p*$ with index expression i. A simple pointer expression $*p$ is assumed to have an index expression of 0. This uniform treatment of pointers and arrays allows existing array disambiguation techniques to be applied

to C code. The Omega test used for Fortran dependence analysis can therefore also be applied to C dependence analysis.

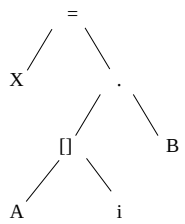
Identifying memory references

The complex access expression from *085.gcc* presented earlier contains several accesses, and will correspond to several load operations in the low-level IR. Dependence analysis has to identify each of the individual accesses in the expression which will correspond to an actual load or store operation in the low-level IR, and place this access in the access table. Once the access table correctly reflects all memory accesses in the program, existing tools from Fortran dependence analysis (e.g., Omega test and scalar dependence analysis) can be used to determine the dependence relationship between the individual accesses. Identifying individual accesses is performed by evaluating the hierarchical (parse-tree) representation of the expression from the bottom up, starting with the variable expression. A trivial example of this is shown in Figure 6.1(a). The expression tree is evaluated, starting with the variable A. To decide if A represents an access, both A and its parent expression (arrow) must be examined. Because the parent expression represents a dereference, A is determined to be an access and is added to the access table. Analysis continues by walking up the tree to the arrow expression. It too is determined to be an access. Finally, the top-level arrow expression is evaluated. Because its parent expression is not one of the access operators, it is also an access. Thus, the expression $A \rightarrow B \rightarrow C$ contains three accesses.

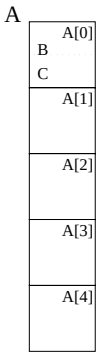
However, identifying the memory references is not always as straightforward as shown in the previous example. Consider the expression $A[i].B$ whose parse tree is shown in Figure 6.1(b). Again, analysis starts with the variable expression A. Because its parent is an index expression



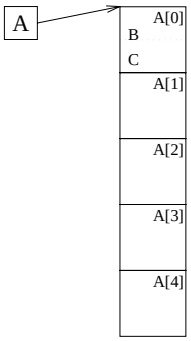
(a) Parse tree of A->B->C + 4



(b) Parse tree of X = A[i].B



(c) Physical memory layout



(d) Alternate memory layout

Figure 6.1 Finding Memory References.

(and the data structure is declared as an array), no explicit load of the variable A is required and it is therefore not an access. Analysis continues on the index expression; because its parent is a dot expression, the index also does not generate an access. Finally, the dot expression is determined to be an access. Thus, the entire expression only generates a single access in the access table. Figure 6.1(c) illustrates why this occurs. The data structure implied by the access expression, again assuming A was declared as an array, is an array of structures, laid out in consecutive memory locations. To access a particular element of one of the arrays, in this case $A[i].B$, the compiler will make the address calculation:

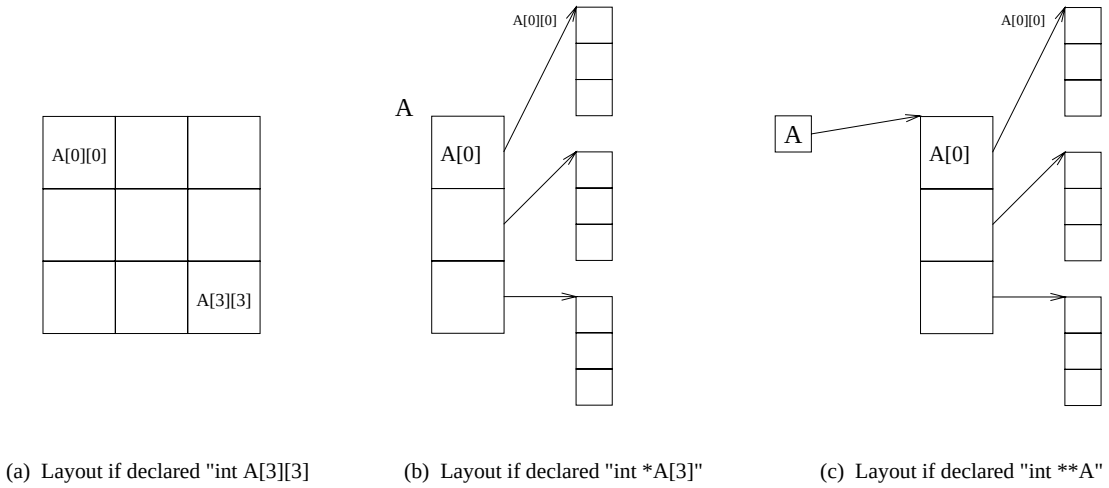
```
address offset = i * sizeof(struct) + (offset of B in struct)
```

A single load will be generated for this expression, from the address of A plus the calculated address offset.

A complication in determining what expressions correspond to memory accesses is that how the variables are declared determines whether the expression corresponds to an access. In the previous example, it was assumed that the variable A was declared as an array. Instead, let's assume the variable was declared as "$int\ *A$." In this case, the expression shown in Figure 6.1(b) corresponds to the physical memory layout depicted in Figure 6.1(d). The variable A is now a pointer, containing the address of the array of structures. To access a field in one element of the array, first A must be loaded to get the address of the array. Then the calculated offset is added to this address to determine the address for the second access. Thus, the entire expression requires two accesses instead of one because variable A was declared differently.

Another example of this complication is for the expression $A[i][j]$, which can correspond to either one, two, or three memory references depending on how the variable was declared. Figure 6.2(a) illustrates the physical data structure for the access expression, assuming A is declared as a two-dimensional array. Simple arithmetic operations calculate the offset of the required element, and a single memory reference is required to fetch the desired value. Figure 6.2(b) and (c) show the alternate data structures for different variable declarations, requiring 2 or 3 memory references. Thus, identifying those expressions which correspond to accesses requires knowledge of the variable declarations.

Another complication in determining those expressions which correspond to memory accesses is the address operator (&). This operator essentially acts to cancel out what would otherwise be an access. Using the example from Figure 6.2, consider now the expression $\&A[i][j]$.



(a) Layout if declared "int A[3][3] (b) Layout if declared "int *A[3]" (c) Layout if declared "int **A"

Figure 6.2 Different Structures Based Upon Data Declaration.

Instead of corresponding to either 1, 2, or 3 accesses as before, now the expression only corresponds to 0, 1, or 2 accesses. For example if the variable was declared as "$int\ *A[3]$" as shown in Figure 6.2(b), the first access to $A[i]$ would still be required, but from the value stored there the address of $A[i][j]$ can be calculated using arithmetic expressions and no further loads are required. Thus, when determining whether an expression is an access, the operators above the current operator in the expression tree must be checked for the address operator.

Table 6.2 summarizes the rules for determining whether an operator corresponds to an access. For each operator in the left column, the entry in the table indicates whether the combination of the operator and a particular parent operator will result in a memory reference. The entries in the table with question marks indicate situations in which the variable declaration determines whether the expression corresponds to a memory reference. One entry which perhaps is non-intuitive is a star operator whose parent is an index operator. Normally, one would expect this type of expression to always correspond to an access. However, it is possible to declare a two-dimension array, and access it using an expression such as $(*A)[j]$, which would

Table 6.2 Rules for Determining if an Operator Corresponds to an Access.

	Parent Operator					
Operator	Addr	Index	Dot	Arrow	Star	Other
Index	N	?	N	Y	?	Y
Dot	N	N	N	Y	Y	Y
Arrow	N	N	N	Y	Y	Y
Star	N	?	N	Y	?	Y
Var	N	?	N	Y	?	Y

be equivalent to $A[0][j]$. (Although legal code, one might question this type of coding style.) In this case, the star operator would not generate an access.

Intraprocedural alias analysis

Whether or not interprocedural alias analysis is performed, C dependence analysis must determine what aliases are formed within each function. Aliases created interprocedurally within other functions or by procedure binding will be discussed in the next section. Here, two types of aliases which are created strictly within the individual function are discussed: aliases which are implicit to the data types of the accesses and aliases which are explicitly formed by assignments within the function body.

Aliases are implicit in the data type for the case of *structures* and *unions*. Because accesses to different fields of these types will be placed in different entries in the access table, explicit aliases are assigned between the entries to aid dependence analysis. In the case of structures, an alias is created between references to the entire structure and to a single field, but not between two fields. In the code segment,

```
A = B;

B.x = x;
```

```
B.y = y;
```

dependences are required between the first and second statements, and between the first and third statements; no dependence is required between the second and third statements. In the case of unions, aliases are assigned between references to the entire union and and its fields, and also between field references.

The other source of aliases with the function is explicit pointer assignments, which can occur in two formats. The first format is the assignment $x = y$, in which the value of one pointer is assigned to another pointer. In this case, the variables x and y are aliased. The second format is the form $z = \&A$, in which an alias is formed between $*z$ and A.

After assignment aliases are formed between entries in the access table, other names in the access table must be searched to perform a closure on the new alias. There are two general types of closure required: common suffix closure and transitive closure. Both will be examined, assuming a new alias between the names $x*$ and y has just been formed.

Common suffix aliases are those which must hold, given the newly created alias holds. They can be derived by symbolic manipulation of the access table names for the newly aliased accesses, looking for pairs of access table entries with names which have a common suffix appended to the original pair. For example, given the alias above, a common suffix alias would be added between access table entries with the names $x**$ and $y*$, if they exist.

Simple transitive closure must also be performed after a new alias is formed. It looks for other aliases of $x*$ and y and forms the transitive alias. For example, if y were previously aliased to z, a new alias between $x*$ and z must be created. A more complex transitive closure is required for the new alias if an alias between w and x existed when the new alias involving

$x*$ is created. In this case, an alias between the names $w*$ and y must be added. Note that the creation of new aliases through closure requires, in turn, closure analysis on these aliases.

6.2 Interprocedural Analysis for C Programs

The existence of pointers in C provides great flexibility to the programmer, but results in a greater challenge to the dependence analysis. Through the use of pointer assignment, the programmer can reference a particular location in memory using different names, thereby creating an alias between the two names. The problem is exacerbated by the fact that an alias formed between two global variables in one function may remain valid within other functions executed after the alias is formed. When performing only intraprocedural dependence analysis, it must be assumed that any global pointer variable could have been aliased to any other global variable by a previously executed function. Additionally, if an alias is formed from a global pointer to a local pointer within the function, the local pointer must also be assumed to alias to all other global variables. These conservative assumptions required to ensure correctness are likely to seriously inhibit the effectiveness of dependence analysis.

An additional aliasing problem is created by procedure binding. If a pointer or array variable is passed as an argument to a function, C passes the variable by reference, and an alias is created between the function's formal parameter and the incoming variable. When doing strictly intraprocedural analysis within a function, the arguments which are used to call the function are not visible to the function. Again, to ensure correctness, the dependence analysis must assume that any global variable could be passed as an argument. Thus, all formal parameters to the function must be conservatively assumed to be aliased to all global variables. Likewise, it is possible that the calling function could pass the same pointer as arguments to

two different formal parameters. Without interprocedural analysis, it must be assumed that any formal parameters used as pointers are aliased.

Though not unique to C, a final dependence analysis problem which occurs when interprocedural information is not available is the presence of function calls within the function being analyzed. Without visibility to other functions, the dependence analysis is unable to determine what global variables the called function may reference or modify. Therefore, the dependence analysis must conservatively assume that all global variables are modified by the called function and must add appropriate dependences between the function call and these variables. Additionally, any arguments passed by reference to the other function could potentially be modified. These arguments, as well as any other variables aliased to the arguments, must also be dependent upon the function call.

Each of these problems requires the intraprocedural dependence analysis to make very conservative assumptions. As a result, the dependence analysis is likely to generate numerous false dependences between independent variables. When these false dependences are passed to the low-level code, important optimization and scheduling opportunities may be missed. Thus, limiting the dependence analysis to a single function at a time, with no visibility to other functions, can result in significantly degraded performance. Considerable compile-time overhead can also result from the maintenance of these false dependence arcs.

Interprocedural analysis can be employed to overcome these problems. By providing visibility to surrounding functions during dependence analysis, overly conservative assumptions can be avoided. To provide strong dependence analysis to support sync arcs, interprocedural analysis was implemented in the Pcode environment as part of this thesis effort. This interprocedural

analysis gathers alias and side effect information and identifies targets of indirect function calls. This information is merged back into the code to support later stages of compilation.

In the remainder of this section, the implementation of this interprocedural analysis is presented. First, the required granularity of the analysis is discussed. This is followed by a discussion of how the program call graph is built, a task complicated by indirect function calls. Finally, some details of the implementation are examined.

6.2.1 Granularity of analysis

Early work in interprocedural analysis [51], [53], [64] employed a rather coarse-grain analysis. Most of the proposed methods examined each function individually, deriving summary information from the function. The program call graph was then built, and a data-flow analysis was performed by iterating over the call graph. This approach produces fairly good results, but is not fully accurate due to the granularity used. Figure 6.3(a) illustrates why accuracy is lost using this approach. Within function *main*, an alias is formed between the variables x and y. Note that this alias is not formed until after the call to $func1$ and does not hold within that function. However, each function is treated as a single node by the interprocedural analysis, the flow information within *main* is lost and the alias would have to be considered valid within $func1$. As a result, a false dependence would be created between the references to $*x$ and $*y$ in $func1$.

More recent work in interprocedural analysis of pointer aliasing [54], [65] performs the analysis at a significantly finer granularity. Landi and Ryder perform interprocedural analysis on a *interprocedural control flow graph* (ICFG), which is essentially the union of the program call graph and the individual functions' control flow graphs, augmented by *entry*, *exit*, and

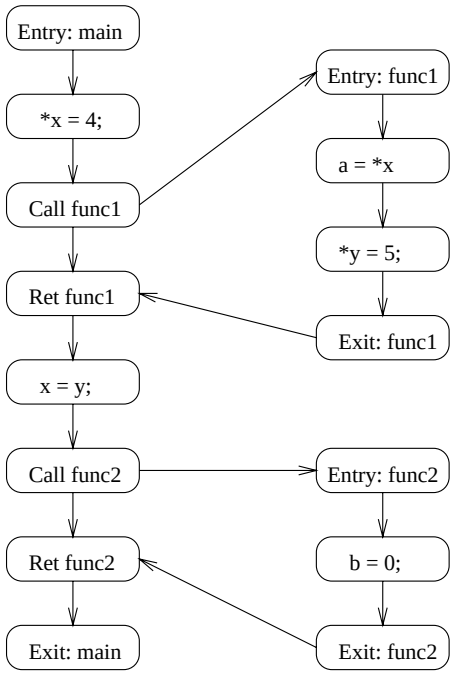
```
func1 ( )
{
   int a;

   a = *x;
   *y = 5;
}


func2 ( )
{
   int b;
   b = 0;
}


main ( )
{
   int  *x, *y;
   *x = 4;
   func1();
   x = y;
   func2();
}
```

(a) Sample program



(b) Interprocedural control flow graph

Figure 6.3 Accuracy Loss of Low-Granularity Interprocedural Analysis.

return nodes. Figure 6.3(b) illustrates the ICFG for the program from Figure 6.3(a). Just as traditional data-flow analysis iterates over a function's control flow graph, interprocedural analysis iterates over this ICFG. Due to this finer granularity, the accuracy of the analysis is increased.

Although the finer-granularity approach provides an increase in the precision of the analysis, it pays for this precision with added complexity. Note that with the ICFG, the entire program must essentially be loaded into memory simultaneously. In contrast, the coarser granularity

approach examines one function at a time, and must only process the summary information for the entire program. Much of the research using this finer granularity has dealt with relatively small programs. It remains to be seen whether this approach is practical on large C programs in terms of both the time required for analysis and the memory requirements.

In determining the granularity of analysis to be used for the interprocedural analysis to support sync arcs, therefore, the tradeoffs between precision and complexity must be considered. The essential question is whether the added precision of the finer-granularity approach will make a significant difference in the performance of low-level optimization and scheduling. Traditionally, source-level dependence analysis has been applied to source-to-source transformations, requiring very precise analysis. For many loop transformations, a single ambiguous dependence is sufficient to prevent the transformation. In contrast, low-level transformations may be much less sensitive to slight imprecisions in the analysis. For example, if the majority of the ambiguous dependences are resolved during code scheduling, a single unnecessary dependence may not result in significant performance degradation. Therefore, dependence analysis to support low-level transformations, although requiring good precision, may be more tolerant of slight imprecisions than dependence analysis for source-level transformations.

The interprocedural analysis chosen to support sync arcs for this thesis uses the coarser-grain approach of iterating over the program call graph rather than the ICFG. It is believed this analysis will provide sufficient precision to allow aggressive low-level code optimization and scheduling. The results presented in Chapter 7 support this supposition.

6.2.2 Building the program call graph

For languages which do not support indirect function calls, a single pass through each of the functions is sufficient to resolve the program call graph. However, indirect function calls (sometimes referred to as function or procedure variables) significantly complicate the problem of building the call graph. Because the actual function being called by an indirect function call may be stored in a global variable or passed into the function via the formal parameters, a simple intraprocedural analysis of the function may be insufficient to determine the target, or targets, of an indirect function call.

Figure 6.4 illustrates this problem, using code segments from the SPEC-CINT92 benchmark *023.eqntott*. Portions of the two functions which perform quicksort are shown. The top-level function, *qsort*, receives the function variable *compar* as a formal parameter. The function assigns the variable to a global variable *qcmp*, and then calls the low-level routine *qst*, which actually performs the recursive sort. The compare function pointer is passed to *qst* via the global variable *qcmp*. Thus, the indirect function call within *qst* is performed using a value which was first passed through a procedure binding, and then through a global variable. An intraprocedural analysis of these functions would be unable to resolve the indirect function call and the program call graph could not accurately be built.

One approach to this problem is to simply search the program for all function names which are ever assigned to function variables. All indirect function calls could then be assumed to potentially call all of these functions. A possible refinement of this approach would be to examine the function arguments of the indirect function call and to assume an indirect function call only calls those functions which match its parameter list. Unfortunately, this simple approach results in many incorrect edges in the call graph. For interprocedural data-

```
qsort (base, n, size, compar)
   char *base;
   int n;
   int size;
   int (*compar)();
{
   qcmp= compar;
          ⋮
   qst (base, max);
          ⋮
   if ((*qcmp)(j, lo) > 0)
      j = lo;
}


qst (base, max)
   char *base, *max;
{
          ⋮
   ((*qcmp)(jj=base,i) > 0)
          ⋮
}
```

Figure 6.4 Interprocedural Function Pointers.

flow analysis, these extra edges will lengthen the analysis and reduce its precision. Additionally, the imprecision may inhibit important transformations which rely upon the call graph (e.g., inlining).

An adequate solution to this problem requires interprocedural data-flow analysis. As seen in Figure 6.4, the possible values of the function variable must be propagated through several functions from where they are defined to where they are used. Unfortunately, interprocedural data-flow analysis requires the presence of a program call graph. Thus, the program call graph and the interprocedural analysis are mutually dependent. One solution is to do both simulta-

```
signals ( )
{
   signal (SIGQUIT, quit);
   signal (SIGPIPE, quit);
   signal (SIGTERM, quit);
   signal (SIGALRM, time_out);
   signal (SIGFPE, quit);
   signal (SIGBUS, quit);
}
```

Figure 6.5 Indirect Function Calls Through Library Functions.

neously, iteratively building the call graph as data-flow is being performed. Wiehl discusses an approach similar to this [53]; however, actual details to his approach are sketchy.

The approach used for this thesis builds the control flow graph while performing data-flow analysis. The data-flow analysis is performed iteratively, initially starting with only the function *main*. As call targets are resolved during the iterative analysis, new functions are added to both the flow analysis and to the call graph. Further details of the iterative analysis are provided in the next section.

This approach has been tested across the fifteen C benchmarks from the benchmark suite, of which six had indirect function calls. All function calls for all benchmarks were successfully resolved and accurate call graphs were generated, with one exception. The technique has difficulty with indirect calls made within library functions. In particular, this occurs with the library function *signal*. Figure 6.5 illustrates this problem, using the signal handling routine from the benchmark *072.sc*. This routine defines user handlers (*quit* and *time_out*) to be invoked in the case of errors. These two routines are never directly called within the user code, but are only provided as function pointers to the signal library. Because they are not called by user code, they cannot accurately be attached to the call graph.

6.2.3 Implementation

The first step in implementing interprocedural analysis is to understand the desired output. To support dependence analysis, the output of the interprocedural analysis should produce accurate alias and side effect information. For each function, the analysis should identify the aliases between pairs of global variables, the aliases between global variables and the function's formal parameters, and the aliases between the formal parameters. Aliases involving local variables can then be resolved for each function using intraprocedural analysis. The side effect information required for each function is simply a list of the global variables defined and used by the function, or by any functions reachable through calls made by this function. Also, side effect information regarding passed arguments is necessary and is used to accurately define the memory dependence between memory operations and subroutine calls.

To provide this desired information, interprocedural analysis is performed in three phases (Figure 6.6). During the first phase, data required to support the interprocedural analysis are extracted from each function in the program. The second phase of interprocedural analysis then performs an iterative data-flow analysis on this summary information from each function. Finally, the results of the interprocedural analysis are used to perform accurate dependence analysis for each function.

Extracting summary data

Rather than attempting to bring the entire program into the compiler at one time and performing data-flow over the entire program, this thesis employs an approach which makes a pass over each function individually in the program. During this pass, summary information for each function is extracted and stored in an interprocedural data file. Thus, the interproce-

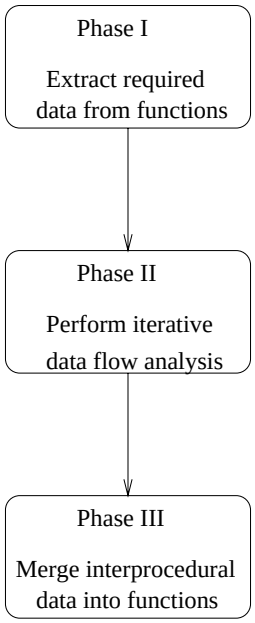


Figure 6.6 Phases of Interprocedural Analysis.

dural analysis must only perform data-flow using the summary information rather than holding the entire program in memory. During the first phase of interprocedural analysis, the summary information is extracted. The following data are included in each function's summary information:

(1) list of formal parameters

(2) list of function calls, including parameters

(3) global variables defined

(4) global variables referenced

(5) aliases created

(6) formal parameters defined

(7) assignments of function names to pointers

(8) variables returned.

The list of a function's formal parameters is included in the summary information to support procedure binding. When the function is called with a global variable which is passed by reference, an alias is formed between the global variable and the function's corresponding formal parameter. To allow creation of this alias, the names of the formal parameters are included in the summary information. Because aliases can only be created if the argument is passed by reference, only the names of formal parameters which are used as pointers will be listed.

The summary information also maintains a list of the function calls made from within the function. Information on function calls is used to build the call graph and to aid in procedure binding during data-flow analysis. For each function call, several pieces of information are recorded in the summary information. First, the call mode, whether direct or indirect, is specified. For direct calls, the name of the called function is specified. For indirect calls, the function variable is specified. Additionally, all variables which alias (intraprocedurally) with the function variable are listed as variables which could contain the function name. The second type of information maintained for each call is the call arguments. As with the function variables, any aliases of the argument variables are also listed. The final information extracted for each call is the return variable for the call. This is maintained to allow binding of the return variable, which can also create aliases.

Because the aim of the interprocedural analysis is to provide accurate aliasing and side effect information, this type of information must obviously be included in each function's summary information. Therefore, the summary contains a list of the global variables defined and referenced by the function. Through data-flow analysis, this list of global variables can be updated to reflect the side effects of the function and any functions reachable through calls from this function. Central to the interprocedural alias analysis is the list of intraprocedural aliases cre-

ated by each function. The summary information stores only aliases created within the function involving global variables or formal parameters. No aliases involving local variables are stored here, because they do not contribute to the interprocedural data-flow analysis. The only local aliases which might affect the analysis are those which alias either formal parameters or call arguments. These aliases have already been recorded in the call section of the summary, and are not necessary here. Another item included in the summary information is a list of the formal parameters which are defined within the function. This is provided to allow the interprocedural analysis to determine whether the function causes side effects on its incoming arguments.

To support building the call graph in the presence of function pointers, the summary information also contains assignment of function names to pointer variables within the function. For function names which appear directly as a call argument, the name is stored in the call section. The final piece of information recorded for each function is a list of the variables returned by the function. During binding for a function return, these variables can be aliased to the variable in the calling function which stores the return value.

Iterative analysis

The second phase of interprocedural analysis performs an iterative data flow analysis of the summary information collected during the previous phase. This phase does not examine the individual functions again; instead only the summary information is used as input. The output of this phase of the analysis is a call graph of the program and updated summary information for each of the functions. This updated summary information provides interprocedural side effect and alias information and lists the actual targets of indirect function calls.

During this phase of the analysis, data-flow analysis is performed on the program call graph, with individual functions as the nodes in the graph. The efficiency of the data-flow algorithm is dependent upon the order in which nodes in the graph are visited. For example, for an acyclic graph in which information flows only from the parent function to the child function, the data-flow requires only a single pass over the graph if performed as a pre-order traversal of the call graph. However, for programs whose call graph contains cycles, the analysis must be performed iteratively until no further changes in the data occur. Therefore, the type and direction of the data flow during the analysis need to be examined.

Building the call graph, which is performed iteratively using the function *main* as a seed, is a top-down type of data-flow. As new functions are added to the graph, they in turn add more children below them in the graph. The direction of data flow for alias information is less clear. Because of the granularity of the analysis, global aliases propagate both up and down through the graph. Local aliases formed through procedure binding are passed downward in the graph; however, the aliases formed by return binding tend to flow upward. Likewise, possible values which can be assigned to function pointers can be propagated up and down through the graph by calls and returns.

Side effect analysis is clearly a bottom-up type of analysis. For each function, the set of global variables modified is simply the union of the variables modified by the function itself, and the variables modified by each of its children. Thus, the global side effect data is propagated upward through the call graph from the leaves, and a bottom-up analysis is most effective. Likewise, argument side effects are most efficiently found through a bottom-up analysis. Because a function's formal parameters may be passed as an argument, determining whether the formal parameter is modified is again a union of whether the function itself modifies the variable and

whether any function to which the variable is passed modifies it. Therefore, side effect data for formal parameters also flow upward through the graph.

For this implementation, the data-flow was divided into two passes, each of which is performed iteratively. First, a top-down data-flow is performed, flowing alias and function pointer information through the graph. The call graph is fully resolved during this pass. Following this pass, a bottom-up data-flow is performed to obtain global variable side effect and function argument side effect data. The top-down and bottom-up passes are detailed in the following sections.

Top-down pass. The top-down data-flow is initiated by adding the function *main* to the call graph. When a new function is added to the call graph, several steps are performed. First, the function is appended to the list of functions being analyzed. The function is then analyzed for direct function calls; any functions called through direct function calls are recursively added to the call graph. Because of the recursive nature of adding arcs to the call graph during function initialization and because most function calls are direct, the majority of the call graph is generally built during function initialization, before any data-flow is actually performed. Finally, the summary information within the function is examined and any information global in scope, such as assignments to global function pointers made within the function, are added to global data structures.

During top-down data-flow analysis, each node is visited in the order it was initialized. This in general equates to a depth-first search of the call graph. During each visit of a node, the following tasks are performed:

(1) Bind incoming arguments.
(2) Bind return variables.

(3) Bind incoming function pointer values.

(4) Bind return function pointer values.

(5) Update side effected function pointer arguments.

(6) Check global function pointer values.

The first two tasks update the alias information flowing through the graph. To bind incoming arguments, the function follows all incoming arcs (to parent nodes) and determines if the aliases for any incoming arguments have changed. If any incoming argument is now aliased to a new global variable, binding requires that the global variable be aliased to the corresponding formal parameter within the function. The creation of this new alias between the global and formal, in turn, may require other data structures within the function to be updated. For example, if the formal is an argument (or alias of an argument) to a function call, then the global variable must now be added to the possible argument list for that function call.

Binding of return variables is similar to binding incoming arguments. All outgoing arcs in the call graph to child functions are followed to determine if the list of variables possibly returned has changed. If so, the variable within the caller which receives the return value must be aliased to the new return variable.

Figure 6.7 illustrates how aliases propagate due to binding of incoming arguments and return variables. During a visit of $func1$, a search of its parent functions will find that the global variable g is passed as an argument and will create an alias between g and $f1$. Because $f1$ also appears as an argument in the call to $func2$, this alias is recorded at the call site. When $func2$ is visited, its search of incoming arguments will find g as a possible alias to its incoming parameter, and an alias of g and $f2$ is created within $func2$. Thus, the alias due to binding has been propagated through multiple function calls. Note also that $func2$ returns a global variable h. During the visit to $func1$, binding of return variables will form an alias of h to l.

```
int *g, *h;

func1 (int *f1)
{
   int *l;
   l = func2(f1);
}

func2 (int *f2)
{
   int *k;
   *h = *f2 + 1;
   return (h);
}

main ( )
{
   int a;
   a = 4;
   g = &a;
   func1(g);
}
```

Figure 6.7 Aliases Created by Binding.

To facilitate building the call graph when indirect function calls are present, the other four tasks in top-down data-flow deal with updating possible values for function pointers. The first two of these, binding incoming function pointer values and binding return function pointer values, correspond directly to the binding steps discussed above. Rather than creating aliases during the binding, function pointers are assigned possible values. In the same way that aliases flow through the call graph, possible function pointer values also propagate. If a function pointer which has been updated with a new possible value is used within the function in an indirect function call, a new target of the call has been defined and the call graph is updated with this new arc.

```
func1 (int *f1)
{
   int (fptr*)();
    func2(fptr);
}

func2 (int (*f2)() )
{
    f2 = func3;
}

func3 ( )
{
}
```

Figure 6.8 Side effects on Function Pointer Arguments.

Updating side effected function pointer arguments involved the flow of data from the child to the parent function via the function arguments being modified. This is illustrated in Figure 6.8. The function $func1$ passes its local function pointer $fptr$ to $func2$ as an argument. The variable is modified by $func2$, giving the pointer the address of $func3$ as its value. The data-flow analysis must handle value side effects to successfully propagate function pointer values and accurately build the call graph. The final task done when visiting a node is to check for global function pointers with new possible values. If the global function pointer is used in the function, the call graph is updated appropriately.

Bottom-up pass. Following the top-down pass, a bottom-up data-flow analysis is performed to resolve side effects. This analysis is much simpler and tends to iterate less than the top-down analysis. The work performed during this pass also more closely resembles a classic data-flow task. The bottom-up analysis is performed by iteratively visiting each function in the

call graph, in the reverse order to which the functions were added to the graph. Two tasks are performed each time a function node is visited.

The first task performed during the visit of a node is to propagate global defines and uses up the call graph. The set of global variables defined or used by a function is simply the union of the set of variables defined or used by all child nodes, plus the global variables actually defined or used within the function. The second task is to propagate definitions of formal parameters up the call graph. In Figure 6.9, to determine whether a side effect is caused on the variable g by the call to $func1$ inside $main$, the side effect of the formal parameter $f2$ in $func2$ must be propagated up to $func1$. In $func1$, this creates a side effect on $f1$, which is then propagated to $main$ to indicate a side effect on g. For this task, then, when a node is visited, all function calls within the node are examined to determine whether they pass formal parameters as arguments to the callee. If so, the callee is examined to determine whether it modifies that particular argument. If the parameter is modified, then the formal parameter within the node being visited is marked as having been modified.

Merging interprocedural data

The final phase of interprocedural analysis entails merging the results back into the dependence analysis for each function. Each function is re-examined individually, and dependence analysis is performed. This time, however, the results of the interprocedural analysis are available to provide increased accuracy. The interprocedural information is used in various ways.

The interprocedural analysis determined the possible targets of all indirect function calls. This list of possible targets is merged into the Pcode data structure as an expression pragma associated with the indirect function call. The information can then be used during dependence

```
int *g;

func1 (int *f1)
{
   int *l;
   l = func2(f1);
}

func2 (int *f2)
{
   *f2 = 1;
}

main ( )
{
   int a;
   a = 4;
   g = &a;
   func1(g);
}
```

Figure 6.9 Propagation of Side effects on Formal Parameters.

analysis: knowing what functions are potentially called by an indirect function call allows accurate dependences to be added between function calls and memory references, using interprocedural side effect information. During subsequent compilation down to low-level IR, this list of targets is preserved and made available for use by other modules.

The interprocedural data also provide an updated list of aliases which hold within each function. During dependence analysis of the individual functions, these aliases are simply used to create additional aliases between entries in the access table, which will in turn create dependences between appropriate memory references.

The final information provided by interprocedural analysis is the global variable and formal parameter side effect information, which is used to provide accurate memory dependences between function calls and memory operations within the function being analyzed. To determine whether a read of a global variable is memory dependent on a function call, the set of global variables defined by the target function is examined. (Although only the function being compiled is visible to the compiler, the interprocedural summary information for all functions is available.) Similarly, to determine whether a write of a global variable is dependent on a function call, both the set of globals defined and the set of globals used must be examined, and the appropriate dependence added. To determine whether a variable which appears as an argument to a function call should be dependent upon the function call, the callee summary information must be examined to determine if the callee modifies the corresponding formal parameter.

6.3 Dependence Analysis Summary

In this chapter, the C dependence analysis implemented within the IMPACT compiler to support sync arcs has been presented. Prior to this thesis, IMPACT supported dependence analysis only for programs originally compiled from Fortran. The required modifications to this dependence analysis to support C semantics, including support for pointer aliasing, is presented. The need for interprocedural dependence analysis, and its required granularity, is also discussed. A coarse-grain interprocedural analysis, which iterates on the program call graph rather than an interprocedural control flow graph, is proposed and implemented. The difficulty of building the call graph in the presence of indirect function calls is also discussed and a solution provided. The interprocedural analysis was tested on a suite of 15 benchmarks and

shown effective at providing accurate dependence information to support sync arcs. Results of this testing is presented in Chapter 7.

CHAPTER 7

EXPERIMENTAL RESULTS

In previous chapters, the sync arc technique for improving static memory disambiguation for low-level code was proposed, and the implementation of source-level interprocedural dependence analysis to support sync arcs was described. In this chapter, the potential impact of sync arcs on performance is quantitatively studied. A suite of 29 benchmarks, including 15 integer and 14 floating-point programs, is evaluated to better understand the ability of sync arcs to provide improved memory disambiguation. Following the analysis of the sync arc technique, a comparison of the static and dynamic techniques proposed in this thesis is performed.

7.1 Sync Arcs

To measure the performance of the sync arc technique, the suite of benchmarks was compiled, with sync arcs added during the Pcode phase. Each benchmark was compiled in several different ways, varying the modules which used the sync arc information to aid memory disambiguation. The different versions of the low-level code were then simulated using the methodology described in Section 2.4. Simulations were performed for architectures with issue width ranging from 4-issue to 12-issue, using the functional unit configurations shown in Table 7.1. The results obtained for the 15 integer benchmarks are presented, followed by the results for the 14 floating-point benchmarks.

Table 7.1 Number of Functional Units.

Functional Units	4 Issue	6 Issue	8 Issue	12 Issue
Int ALU	2	3	4	6
FP ALU	2	3	4	6
Memory	2	2	4	6
Branch	2	2	4	4

7.1.1 Integer benchmarks

The 15 integer benchmarks were all originally written in C and are compiled using the C interprocedural dependence analysis developed for this thesis. Programs written in C tend to be very control-intensive, resulting in small basic blocks. Although techniques such as the superblock reduce the impact of these branches on scheduling and optimization, processor performance may be highly dependent upon the ability to predict and execute multiple branches per cycle.

The integer benchmarks evaluated include nine common Unix benchmarks and the six SPEC-CINT92 benchmarks. For each experiment, separate graphs showing Unix and SPEC-CINT92 results are provided.

Speedup on 8-issue architecture

Figures 7.1 and 7.2 show the speedup of code compiled with and without sync arcs over a baseline single-issue architecture. The bar on the left for each benchmark reflects the speedup of an 8-issue architecture for code compiled without using sync arcs, relative to the baseline single-issue architecture. The right-hand bar shows the speedup for the same 8-issue architecture using the identical compilation path, except that sync arcs are employed to aid memory

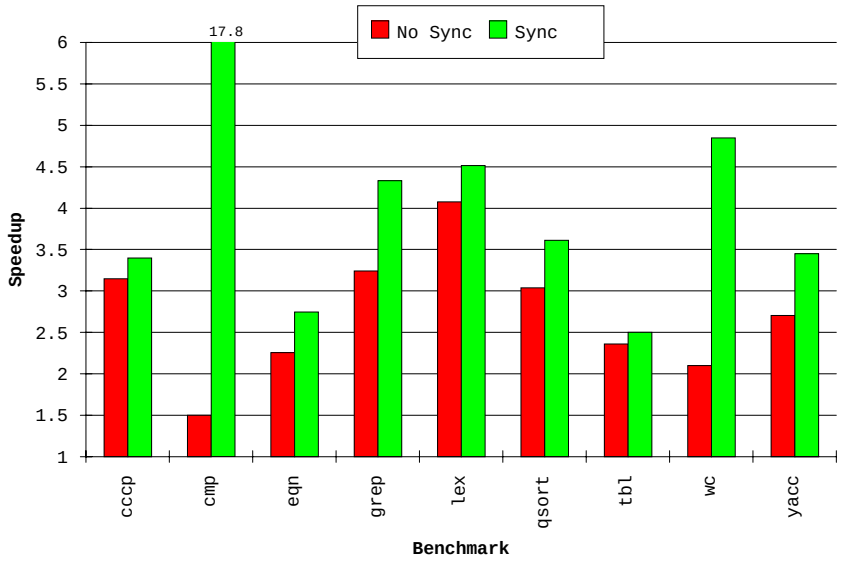


Figure 7.1 Sync Arc 8-Issue Unix Results.

disambiguation. Figures 7.3 and 7.4 show similar information, except that here the ratio of performance between the sync arc and non-sync arc cases is shown. For example, for the benchmark *grep* in Figure 7.1, the non-sync arc code provided a 3.2 times speedup over the base case and the sync arc code provided a 4.3 times speedup. This ratio of these two speedups is reflected as a 1.34 times speedup in Figure 7.3.

The data demonstrate that the sync arc technique is successful at removing ambiguous memory dependences which the existing low-level memory disambiguation was unable to eliminate. Sync arcs resulted in more than 20% speedup over the same architecture without sync arcs for six of the benchmarks, and significantly greater speedup than this in a few cases. The improved memory disambiguation significantly impacted overall ILP; for most benchmarks, the code compiled with sync arcs provided greater than 2.5 times speedup over the baseline single-issue architecture.

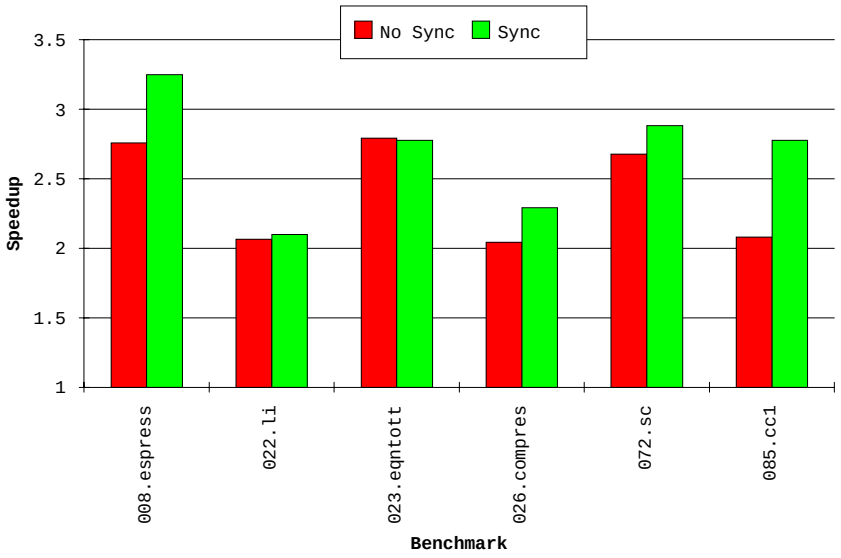


Figure 7.2 Sync Arc 8-Issue SPEC-CINT92 Results.

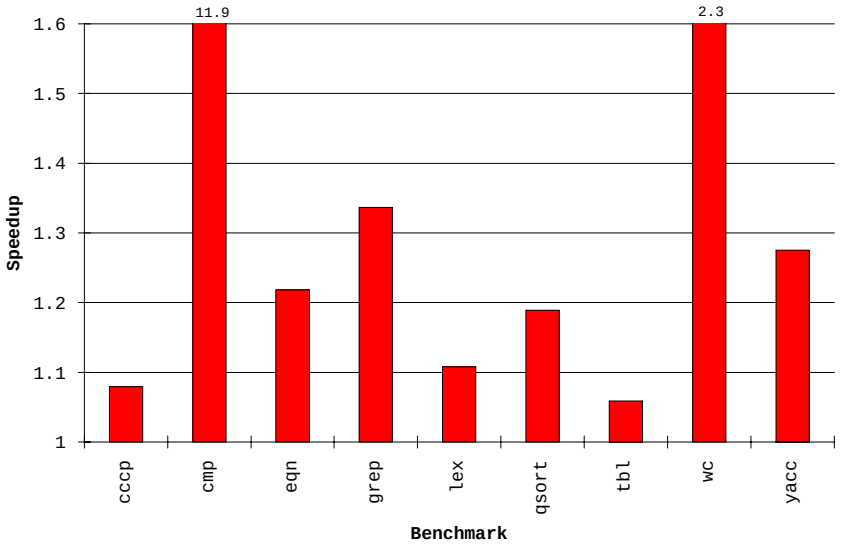


Figure 7.3 Sync Arc 8-Issue Unix Ratios.

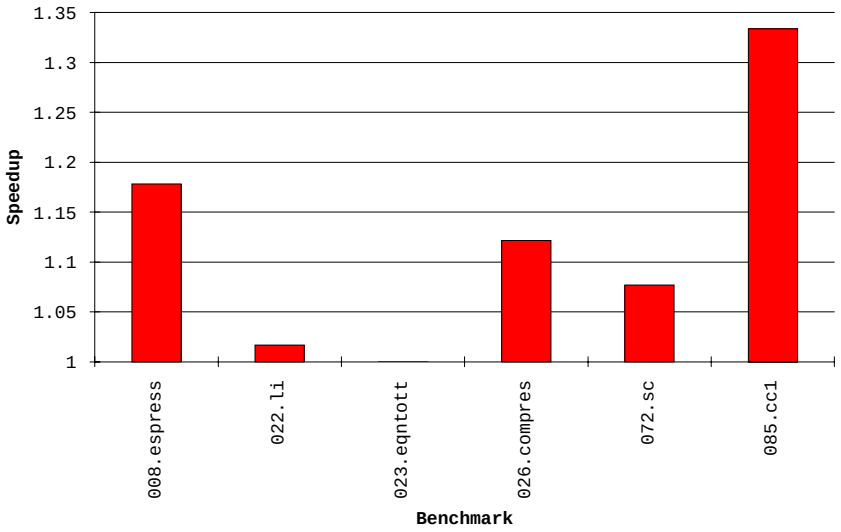


Figure 7.4 Sync Arc 8-Issue SPEC-CINT92 Ratios.

At first glance, an overall speedup of 2.5 may appear somewhat low for an 8-issue processor. However, the potential speedup available as the number of issue positions and functional units is increased is limited by several factors (other than memory and control dependences). First, although an 8-issue architecture is being modeled, only 4 ALUs were available per cycle. Second, because inlining has not been implemented at the Pcode level, it was not performed for this code, limiting the ILP which was exposed. Also, performance improvements are somewhat hidden by the cache and branch prediction effects being simulated. For example, if 30% of the execution cycles are lost in the single-issue architecture due to cache and branch prediction misses, the 8-issue architecture would have to provide much greater than 2.5 times speedup during effective cycles to achieve an overall 2.5 times speedup on the program. This effect is especially evident in the benchmark *026.compress*, whose cache hit rate during simulation was only 77%. Although cycle estimates provided by the IMPACT scheduler indicated a potential

24% speedup due to sync arcs, only about 12% speedup was obtained from simulation results due to the cache effects.

One particularly interesting result was for the benchmark *cmp*, which obtained a 17.8 times speedup over the single-issue processor and an 11.8 times speedup over the 8-issue processor without sync arcs. This extreme effect results because *cmp* is a very small benchmark whose execution is dominated by a single inner loop. Figure 7.5 shows source code for this inner loop, pruned to show the main trace through the loop. Figure 7.6(a) shows the Lcode for one iteration of the loop after optimization, without the benefit of sync arcs. The loop contains 26 instructions, including several stores. Figure 7.6(b) shows the inner loop for the sync arc code. The inner loop contains only 7 instructions, with no stores. Because of the improved memory disambiguation, the remainder of the instructions was able to be removed from the loop. As a result, the non-sync arc code executes more than 3 times the number of dynamic instructions as the sync arc code. Additionally, the store instructions in the non-sync arc code inhibit subsequent code scheduling; the sync arc code is able to obtain an overall instructions per cycle (IPC) value of 4.82 for the program, compared with 1.21 IPC for the non-sync arc code. The combination of these two factors results in an overall 11.8 times speedup for the sync arc code.

Improved memory disambiguation is not able to increase performance for all benchmarks. A notable case is the benchmark *023.eqntott*, from SPEC-CINT92. This benchmark is dominated by a single loop which accounts for over 80% of the execution time. This loop contains no store operations, and memory disambiguation is not a factor in exposing ILP. Note that in Figure 7.2 the non-sync arc code for *023.eqntott* already performs well, showing the best of speedup compared to the single-issue architecture of any of the SPEC-CINT92 benchmarks. Another

```
while(1) {
    chr++;
    c1 = (--(file1)->__cnt < 0 ? __filbuf(file1) : (int) *(file1)->__ptr++);
    c2 = (--(file2)->__cnt < 0 ? __filbuf(file2) : (int) *(file2)->__ptr++);
        if(c1 == c2) {
            if (c1 == '\n') line++;
            continue;
        }
}
```

Figure 7.5 Source Code for Inner Loop of *cmp*.

benchmark which showed little benefit from sync arcs was *022.li*, because *022.li* contains many heavily executed functions which contain a single acyclic basic block. For this benchmark, inlining is required to expose available ILP, and better results for sync arc code are anticipated after Pcode inlining is available.

Relative benefit of optimization and scheduling

For the benchmark *cmp*, large performance benefits were obtained by both optimization, which reduced the number of dynamic instructions, and scheduling, which exposed the available parallelism to the hardware. An experiment was performed to measure the relative benefit of sync arcs to optimization and scheduling. Figures 7.7 and 7.8 show the results of this experiment, which again assumed an 8-issue architecture. For each benchmark, three bars are shown, each reflecting speedup over code compiled without sync arcs. The leftmost bar reflects the speedup if sync arcs are used to aid classic and ILP optimizations, but not code scheduling. The middle bar shows the speedup achieved when sync arcs are used to aid scheduling, but not optimization. The right bar shows the overall speedup when sync arcs are applied to both optimization and scheduling.

```
(op 158 ld_i <LF> [(r 179 i)] [(r 157 i)(i 0)]
(op 160 add [(r 180 i)] [(r 179 i)(i 1)]
(op 161 st_i <LF> [] [(r 157 i)(i 0)(r 180 i)]
(op 162 ld_i <LF> [(r 181 i)] [(r 158 i)(i 0)]
(op 163 ld_i [(r 182 i)] [(r 181 i)(i 0)]
(op 164 add [(r 183 i)] [(r 182 i)(i -1)]
(op 165 st_i [] [(r 181 i)(i 0)(r 183 i)]
(op 168 blt [] [(r 182 i)(i 1)(cb 73)]
(op 169 ld_i <LF> [(r 184 i)] [(r 158 i)(i 0)]
(op 170 ld_i [(r 185 i)] [(r 184 i)(i 4)]
(op 172 add_u [(r 186 i)] [(r 185 i)(i 1)]
(op 173 st_i [] [(r 184 i)(i 4)(r 186 i)]
(op 174 ld_uc [(r 1 i)] [(r 185 i)(i 0)]
(op 176 ld_i <LF> [(r 187 i)] [(r 160 i)(i 0)]
(op 177 ld_i [(r 188 i)] [(r 187 i)(i 0)]
(op 178 add [(r 189 i)] [(r 188 i)(i -1)]
(op 179 st_i [] [(r 187 i)(i 0)(r 189 i)]
(op 182 blt [] [(r 188 i)(i 1)(cb 49)]
(op 183 ld_i <LF> [(r 190 i)] [(r 160 i)(i 0)]
(op 184 ld_i [(r 191 i)] [(r 190 i)(i 4)]
(op 186 add_u [(r 192 i)] [(r 191 i)(i 1)]
(op 187 st_i [] [(r 190 i)(i 4)(r 192 i)]
(op 188 ld_uc [(r 2 i)] [(r 191 i)(i 0)]
(op 190 bne [] [(r 1 i)(r 2 i)(cb 54)]
(op 191 beq [] [(r 1 i)(i 10)(cb 50)]
(op 192 beq [] [(r 1 i)(i -1)(cb 52)]
```

(a) Non-sync arc code

```
(op 168 blt [] [(r 219 i)(i -7)(cb 82)]
(op 174 ld_uc [(r 1 i)] [(r 210 i)(i -8)]
(op 182 blt [] [(r 201 i)(i -7)(cb 83)]
(op 188 ld_uc [(r 2 i)] [(r 192 i)(i -8)]
(op 190 bne [] [(r 1 i)(r 2 i)(cb 84)]
(op 191 beq [] [(r 1 i)(i 10)(cb 85)]
(op 192 beq [] [(r 1 i)(i -1)(cb 86)]
```

(b) Sync arc code

Figure 7.6 Lcode for Inner Loop of *cmp*.

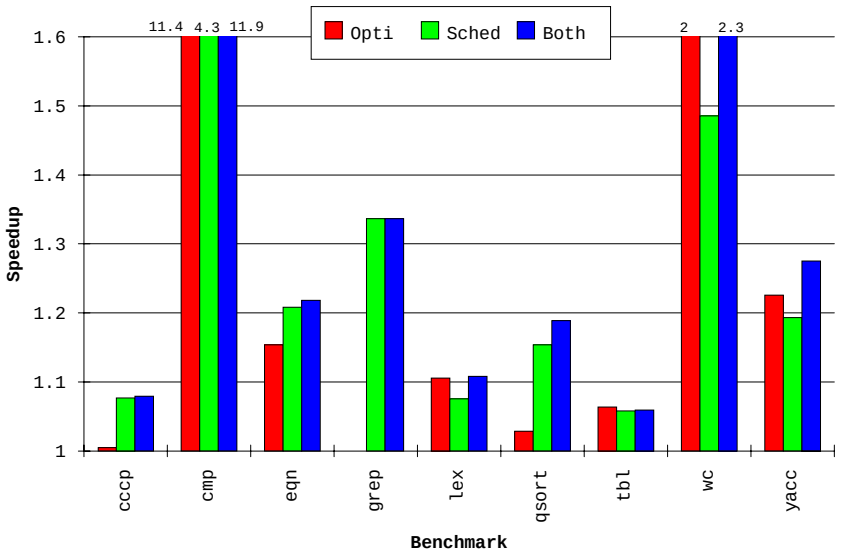


Figure 7.7 Sync Arc 8-Issue Unix Optimization Versus Scheduling.

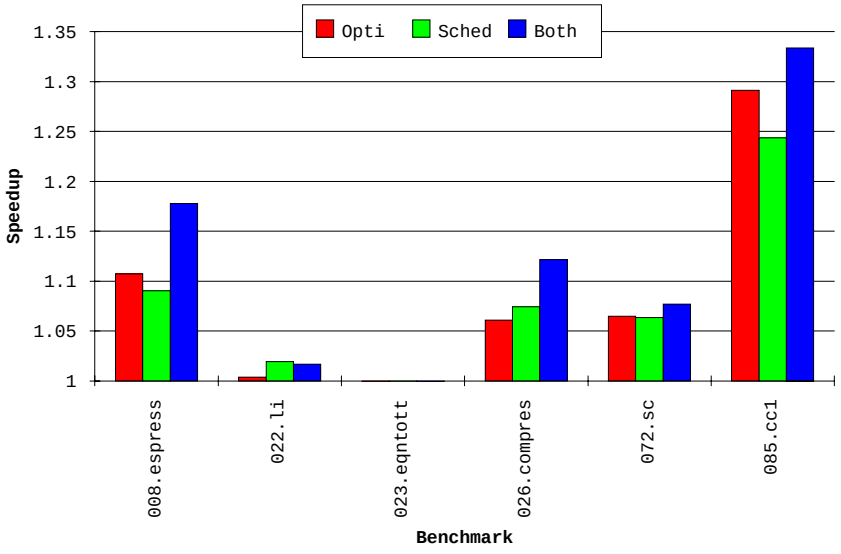


Figure 7.8 Sync Arc 8-Issue SPEC-CINT92 Optimization Versus Scheduling.

Note that the combined effect of using sync arcs for both optimization and scheduling is less than the product of the speedup when using sync arcs individually for either optimization or scheduling, because there is overlap in how the speedup is achieved. For example, if a load can be moved outside a loop during optimization, speedup is achieved because fewer operations must be executed. If the load is not removed by optimization, scheduling may be able to hide the cost of the load by scheduling it in an otherwise empty slot. However, if both optimization and scheduling use sync arcs, the optimization will remove the load and the scheduler will be less able to obtain speedup. Thus, the benefit is achieved by either optimization or scheduling, but not both. However, in some cases, there is a complementary effect, such as the *cmp* example in Figure 7.6. In this example, optimization removed stores from the loop, which resulted in added freedom to the scheduler.

Examining the data in Figures 7.7 and 7.8, no clear pattern emerges as to whether improved disambiguation is more important to optimization or scheduling. Benchmarks such as *cmp*, *lex*, and *wc* achieve most of their speedup due to improved optimization. Others, such as *eqn* and *grep*, achieve speedup primarily as the result of scheduling. In general, results indicate that improved memory disambiguation can significantly enhance the capabilities of both optimization and scheduling.

Varying the number of functional units

For narrow-issue architectures, it may be less critical for the compiler to expose instruction-level parallelism. However, as architectures employ more functional units and issue multiple instructions per cycle, the compiler must attempt to expose sufficient parallelism to make effective use of available resources. An experiment was performed using architectures with

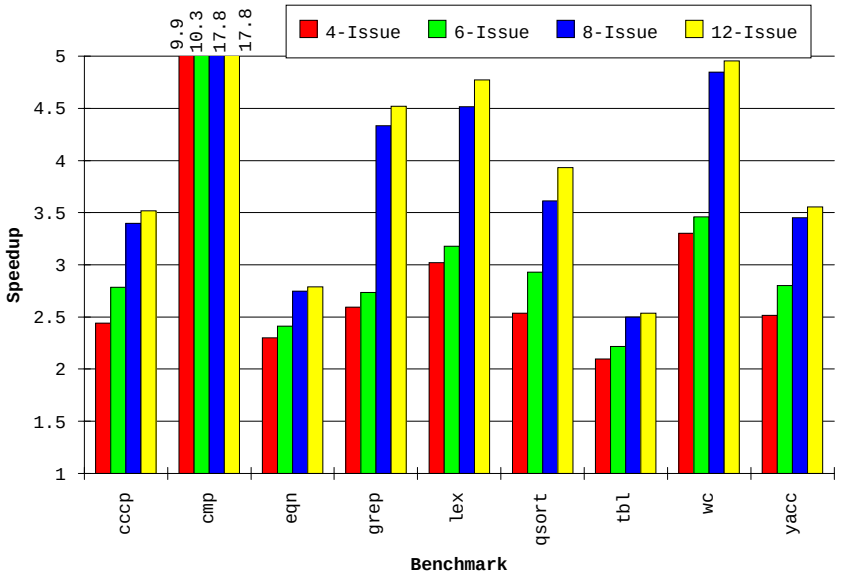


Figure 7.9 Sync Arc Unix Results for Different Issue Rates.

various amounts of resources to determine how effectively IMPACT compilation with sync arcs makes use of increased or decreased resources. Figures 7.9 and 7.10 show the results of this experiment for integer benchmarks. These figures show the speedup which the 4, 6, 8, or 12 issue architecture would achieve using code compiled with sync arcs over the baseline single-issue architecture.

The desired outcome is to see increasing speedup as more function units are made available. For the majority of benchmarks tested, additional processor resources do provide a significant performance increase. This indicates that sufficient ILP is being exposed to the hardware in many important sections of the code. However, for several of the benchmarks, only modest improvements are achieved by the wider issue architectures. For example, although processor resources are increased threefold (from 4- to 12-issue), the benchmarks *tbl* and *022.li* achieve only minor performance benefit, indicating ILP is not being adequately exposed to the hardware.

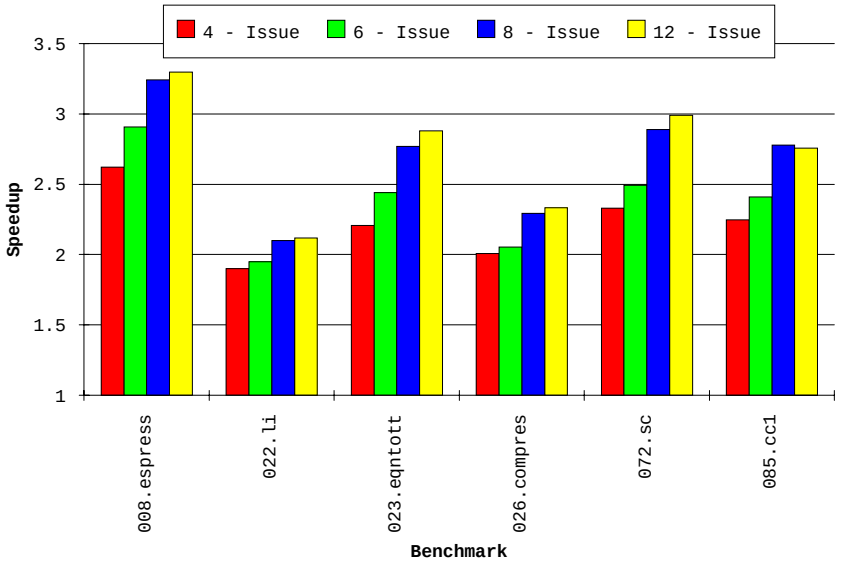


Figure 7.10 Sync Arc SPEC-CINT92 Results for Different Issue Rates.

The lack of inlining during these experiments was a major factor in the inability to expose ILP, particularly for these two benchmarks.

An interesting result seen particularly in the Unix benchmarks is the stairstep effect, in which the results for 8- and 12-issue architectures are similar, but significantly increased over the 4- and 6-issue architectures. Notice in *cmp*, *grep*, *lex*, and *wc* in particular that the results for 4- and 6-issue architectures are nearly equivalent, yet significantly less than the performance for 8- or 12-issue. This stairstep effect is the result of the number of branches the different architectures can issue per cycle. The 4- and 6-issue architectures can execute only 2 branches per cycle, while the 8- and 12-issue architectures can execute 4 branches per cycle. This indicates that the performance for benchmarks which exhibit this stairstep is being directly limited by the ability to execute branches. For example, the code for *cmp* in Figure 7.6(b) shows that the inner loop has been reduced to 8 instructions, 6 of which are branches. Thus, the ability to effectively execute multiple branches per cycle drives processor performance.

Effectiveness of coarse-grain interprocedural analysis

To provide interprocedural aliasing and side-effect information, the dependence analysis implemented for this thesis employs a low-granularity interprocedural analysis. (For further details on this analysis, which iterates on the program call graph, see Section 6.2.) To measure the accuracy of this analysis, the performance of sync arc code generated by the interprocedural analysis was compared to "ideal" sync arc code obtained without interprocedural analysis. This sync arc code without interprocedural analysis makes the unsafe assumption that no aliases hold in a function except those created within the function. Surprisingly, all fifteen Unix and SPEC-CINT92 benchmarks successfully compiled and executed despite this unsafe assumption. Because of this, the effectiveness of the interprocedural analysis at limiting unnecessary aliasing can be measured against an "ideal" analysis which has no interprocedural aliasing.

Figures 7.11 and 7.12 show the results of this analysis. In these figures, the performance of the code using the implemented interprocedural analysis is shown relative to the performance for the "ideal" analysis. A positive performance difference for a benchmark indicates that the interprocedural code performed better, while a negative difference indicates the ideal code compiled without interprocedural analysis performed better. A large negative number (>10%) would indicate that the interprocedural analysis added aliases (possibly correct and necessary) which slowed performance compared to the non-interprocedural code. Note that for all 15 integer benchmarks, no significant performance difference was seen between the interprocedural and non-interprocedural cases. This indicates that the coarse-grain interprocedural analysis employed to support low-level code transformations provided sufficient accuracy. The minor variations seen (±3%) can be attributed to the typical variations obtained from detailed

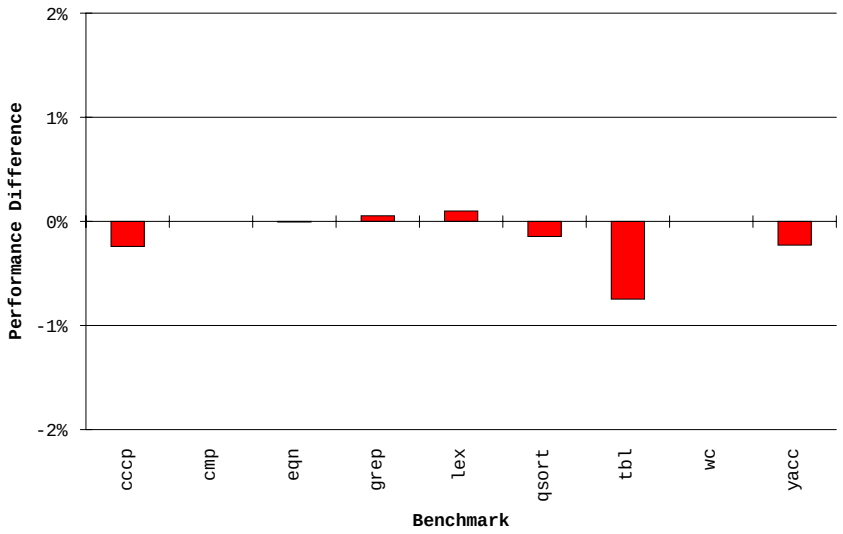


Figure 7.11 Effect of Interprocedural Analysis - Unix.

simulation (e.g., deleting an instruction changes the addresses of branches and may change the effectiveness of the branch prediction scheme).

7.1.2 Floating-point benchmarks

The benchmarks from SPEC-CFP92 were also evaluated to measure sync arc performance. Of the 14 floating-point programs, 12 are Fortran programs which were translated to C using the *f2c* tool. Floating-point programs generally display significantly different characteristics than integer programs. Floating-point programs are usually much less control intensive: they tend to have larger basic blocks than integer C programs. They also tend to have regular loop structures, which perform more average iterations than integer code, and make frequent use of array data structures. These factors would tend to make ILP compilation for floating-point code an easier task, assuming good memory disambiguation is available for array references.

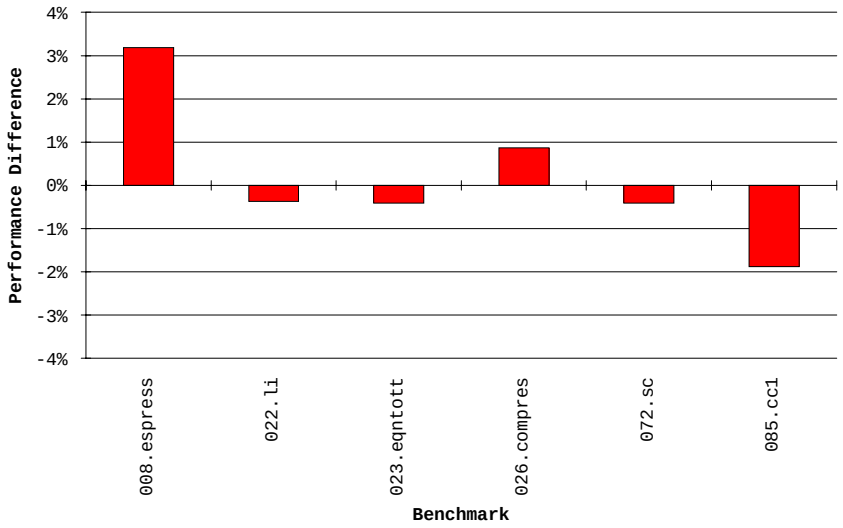


Figure 7.12 Effect of Interprocedural Analysis - SPEC-CINT92.

The following sections present the experimental results for floating-point benchmarks. The methodology and graph format are identical to the results presented in the integer section.

Speedup on 8-issue architecture

Figure 7.13 presents the speedup results for floating-point code compiled with and without sync arcs on an 8-issue processor, baselined to the single-issue processor. Because the existing Lcode memory disambiguation has limited effectiveness for array references, the results for the non-sync arc code are relatively poor. A speedup of less than 2 is achieved for most programs, despite the ability to issue 8 instructions per cycle. With the benefit of improved memory disambiguation, the sync arc code provides substantially better performance for most benchmarks. Figure 7.14, which shows the speedup of sync arc code relative to non-sync arc code, indicates that the benefit of improved memory disambiguation is more pronounced for floating-point code than for integer code (Figures 7.3 and 7.4).

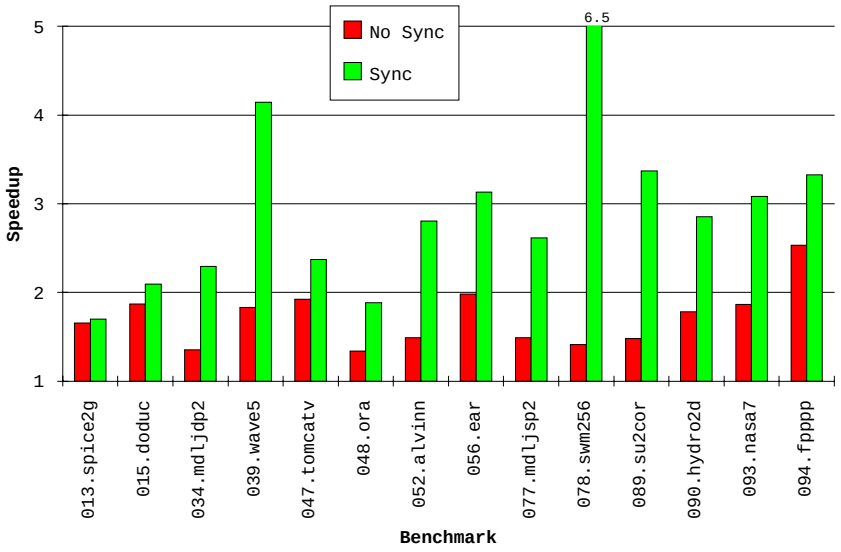


Figure 7.13 Sync Arc 8-Issue SPEC-CFP92 Results.

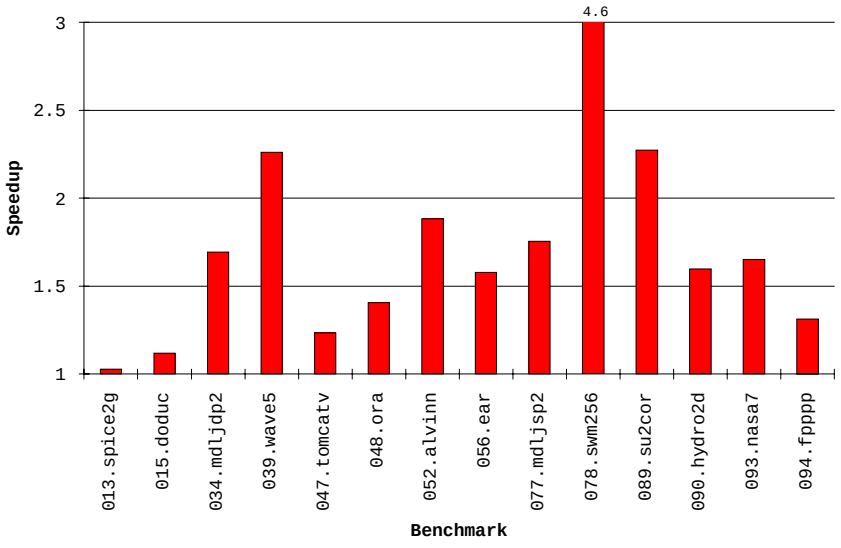


Figure 7.14 Sync Arc 8-Issue SPEC-CFP92 Ratios.

```
for (j = 1; j <= i__1; ++j) {
        i__2 = cons_1.m;
        for (i = 1; i <= i__2; ++i) {
            _BLNK__1.cu[i + 1 + j * 257 - 258] = (_BLNK__1.p[i + 1 + j * 257
                    - 258] + _BLNK__1.p[i + j * 257 - 258]) * (float).5 *
                    _BLNK__1.u[i + 1 + j * 257 - 258];
            _BLNK__1.cv[i + (j + 1) * 257 - 258] = (_BLNK__1.p[i + (j + 1) *
                    257 - 258] + _BLNK__1.p[i + j * 257 - 258]) * (float).5 *
                    _BLNK__1.v[i + (j + 1) * 257 - 258];
            _BLNK__1.z[i + 1 + (j + 1) * 257 - 258] = (fsdx * (_BLNK__1.v[i +
                    1 + (j + 1) * 257 - 258] - _BLNK__1.v[i + (j + 1) * 257 -
                    258]) - fsdy * (_BLNK__1.u[i + 1 + (j + 1) * 257 - 258] -
                    _BLNK__1.u[i + 1 + j * 257 - 258])) / (_BLNK__1.p[i + j *
                    257 - 258] + _BLNK__1.p[i + 1 + j * 257 - 258] +
                    _BLNK__1.p[i + 1 + (j + 1) * 257 - 258] + _BLNK__1.p[i + (
                    j + 1) * 257 - 258]);
            _BLNK__1.h[i + j * 257 - 258] = _BLNK__1.p[i + j * 257 - 258] + (
                    _BLNK__1.u[i + 1 + j * 257 - 258] * _BLNK__1.u[i + 1 + j *
                     257 - 258] + _BLNK__1.u[i + j * 257 - 258] * _BLNK__1.u[
                    i + j * 257 - 258] + _BLNK__1.v[i + (j + 1) * 257 - 258] *
                     _BLNK__1.v[i + (j + 1) * 257 - 258] + _BLNK__1.v[i + j *
                    257 - 258] * _BLNK__1.v[i + j * 257 - 258]) * (float).25;
        }
    }
```

Figure 7.15 Source Code for Inner Loop Nest of *078.swm256*.

The benchmark *078.swm256* obtained the most dramatic speedup of 4.6 times over code compiled without sync arcs. Figure 7.15 shows the inner loop nest of this benchmark, responsible for 99% of the execution time. The inner loop is a *do_all* loop, with no cross-iteration dependences. With the benefit of sync arcs, the low-level scheduler is able to determine that all load operations are independent of all stores. The scheduler is able to overlap operations from different iterations of the unrolled loop and, thus, obtain large speedup. Note that the array-type structure fields would be extremely difficult for low-level memory disambiguation to handle without the use of source-level information.

For most floating-point benchmarks, the sync arc code achieves greater than 2.5 times speedup over the single-issue processor. Although this speedup is much better than that

```
for (i=0;i<n;i++){
        tempin = input[i];
        output[i] = a0[i] * tempin + state1[i];
}
```

Figure 7.16 Source Code from *056.ear*.

achieved by the non-sync arc code, one would expect more parallelism to be available. In fact, the ILP achieved for the floating-point code was not significantly different than that achieved for the control-intensive integer code. This is the result of several factors.

As discussed above, the effects of detailed simulation hide some of the ILP which has been exposed. For the benchmark *093.nasa7*, scheduler cycle estimates indicated a 5 times speedup was achieved for the sync arc code, but due to a 77% cache hit rate, the actual speedup obtained was only about 3.1.

Limitations in the current Pcode dependence analysis inhibits the ILP achieved for several benchmarks. A good example of this is found in the C benchmark *056.ear*. Figure 7.16 shows source code which is representative of code found in several critical functions in *056.ear*. The arrays *input*, *output*, and *state*1 are all formal parameters to the function. Interprocedural analysis correctly aliases the arrays *input* and *output*, because the same array is passed to both variables. The third array, *state*1, is independent of the other two. The problem arises is that Pcode dependence analysis currently does not have a concept of "exact aliasing," such that an alias exists and the arrays begin at the same location. Without this concept, the Pcode dependence analysis must assume loop carried dependences from the store of *output* to loads of *input* in subsequent iterations. After loop unrolling, this dependence severely restricts scheduling in a loop which actually has no loop carried dependences.

```
L150:
    locij = nodplc[tabinf_1.jcpt + locij - 1];
    if (nodplc[tabinf_1.jcolno + locij - 1] == j) {
        goto L155;
    }
    goto L150;
L155:
```

Figure 7.17 Source Code from *013.spice2g6*.

In some cases, parallelism cannot easily be exposed by current ILP techniques. For the benchmark *013.spice2g6*, there is little available parallelism within several of the important inner loops. Figure 7.17 shows an example from its most important inner loop. The variable *locij* is loaded each cycle, and then used in the subsequent iteration to compute the address of its next load. This creates a dependence cycle of length four, which limits the initiation of iterations in the unrolled loop to one every four cycles. Because the low-level code contains only seven operations, initiating a new loop iteration every four cycles allows little overlap of iterations. ILP is potentially available within the outer loop, but exposing it would require extremely sophisticated compilation techniques.

Another factor impacting the ILP that IMPACT achieves for floating-point benchmarks is that IMPACT ILP research in the past has emphasized optimizing C code, and techniques may not be optimized for exposing ILP in floating-point programs. In several benchmarks, overly aggressive scheduling of large superblocks resulted in long register lifetimes and performance was lost due to register spill code. For other benchmarks, the superblock was not the ideal structure for exposing parallelism because of heavy control flow in the inner loops. For architectures which support predication, a technique such as the hyperblock may have been more successful at exposing available ILP.

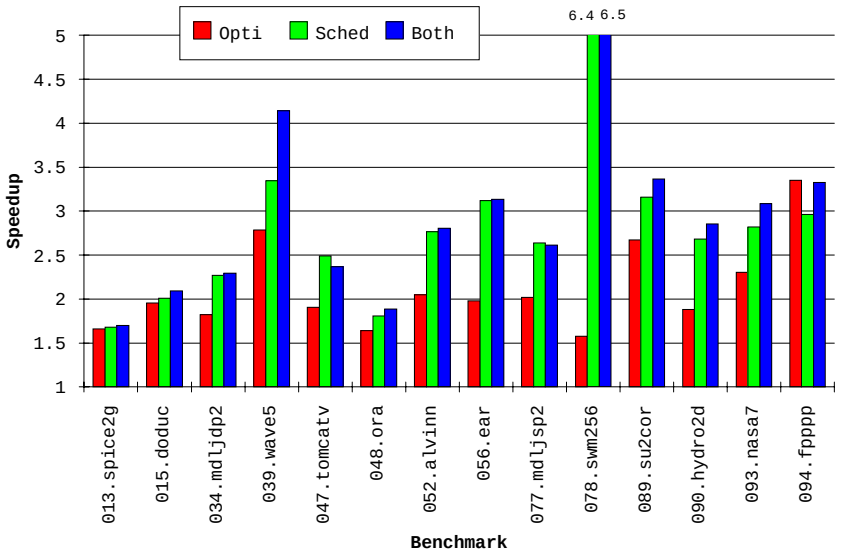


Figure 7.18 Sync Arc 8-Issue SPEC-CFP92 Optimization Versus Scheduling.

Relative benefit of optimization and scheduling

Figure 7.18 shows the relative impact of optimization and scheduling on floating-point code. Similar to the results for integer code, both optimization and scheduling appear to benefit from improved memory disambiguation. For floating-point code, it appears that the benefit of sync arcs may be more important for scheduling than for optimization. Benchmarks such as *047.tomcatv*, *052.alvinn*, *056.ear*, and *078.swm256* achieved the same speedup when sync arcs were used only for scheduling as when they were used for optimization.

Varying the number of functional units

To measure the effectiveness of compilation at exposing ILP for floating-point benchmarks, an experiment was performed using architectures with various amounts of resources. Figure 7.19 shows the result of this experiment. Again, the desired outcome is to see increasing speedup as more function units are made available, as was observed for the benchmarks *039.wave5* and

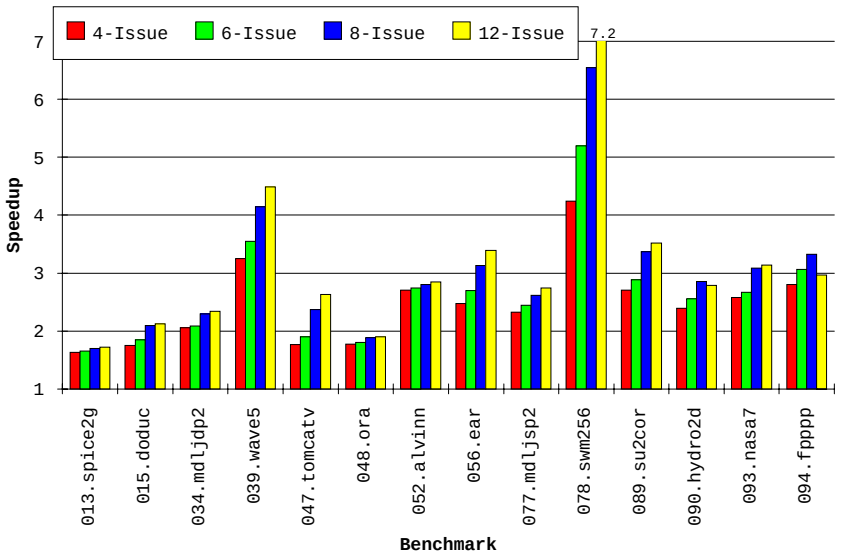


Figure 7.19 Sync Arc SPEC-CFP92 Results for Different Issue Rates.

078.swm256. Unlike the integer benchmarks, however, additional resources had little impact on many of the floating-point benchmarks. Programs such as *013.spice2g6*, *048.ora*, and *052.alvinn* showed little or no performance improvement as the number of resources was increased. This indicates that the compiler, despite improved memory disambiguation, has been unsuccessful at exposing ILP to the hardware.

7.2 Comparison of Static and Dynamic Approaches

Static and dynamic memory disambiguation approaches are best targeted for different applications. Static memory disambiguation is useful for applications for which source code is available, for which dependence analysis techniques have been developed, and which can afford the extra compilation time. Dynamic memory disambiguation is useful for applications for which source code is not available, but which allow modifications to the ISA. However, for applications in which potentially both static and dynamic memory disambiguation could be

applied, it is useful to understand the relative capabilities of the two approaches for improving memory disambiguation. In this section, the MCB results presented in Chapter 4 are compared with the results presented for sync arcs earlier in this chapter.

7.2.1 Performance comparison

To provide a "level playing ground" to compare the two approaches, two minor changes were made to the experiments presented earlier. First, the sync arc results used for comparison here were obtained using sync arc information only to aid code scheduling, and not optimization. This is because the MCB technique has currently been applied only to scheduling and a fair comparison requires comparison of the effect only on scheduling. The second change made for this experiment is to increase the number of floating-point registers in the sync arc model to 128 single-precision and 64 double-precision registers. The MCB results from Chapter 4 were obtained using this number of registers; thus, the sync arc architecture was modified so that both approaches used the same architecture.

Figures 7.20, 7.21, and 7.22 show the comparison between the sync arc approach and the MCB approach for the Unix, SPEC-CINT92, and SPEC-CFP92 benchmarks, respectively. Results are shown for an 8-issue architecture, using the same functional unit mix as in previous experiments. The speedups shown are all relative to a baseline single-issue architecture without improved memory disambiguation. Three bars are presented for each benchmark. The leftmost bar reflects the speedup obtained using the sync arc approach. The center bar reflects the results obtained using the set associative MCB design developed in this thesis. As discussed in Chapter 4, this approach suffered from conflicts for floating-point code, and significant perfor-

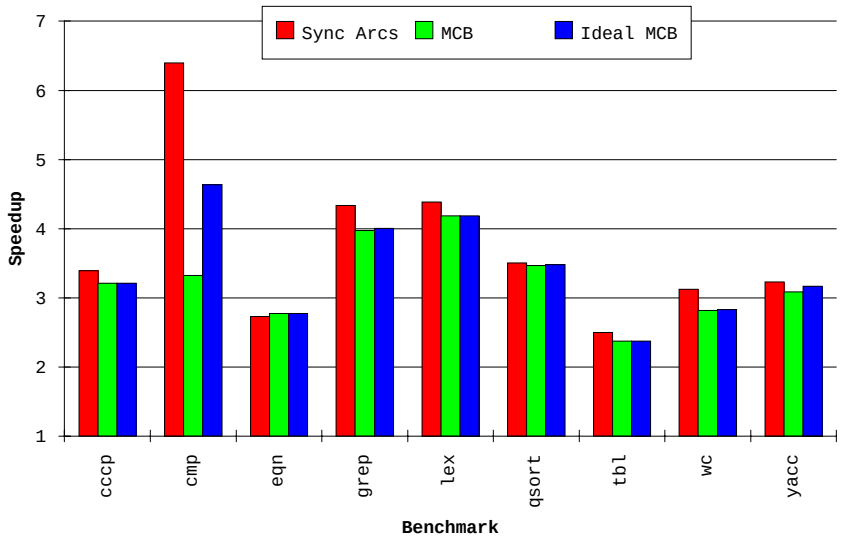


Figure 7.20 Unix Comparison of Sync Arcs to MCB - 8-Issue.

mance was lost. To compare sync arcs with optimal MCB hardware, the rightmost bar reflects the perfect MCB case, in which false conflicts do not occur.

For the 15 integer benchmarks, the sync arc and MCB approaches provided almost identical performance benefit, with the sync arc approach fractionally better. The one exception was *cmp*, which will be discussed later. The set associative and perfect MCB designs in general provided nearly equivalent performance, indicating that conflicts were not a major problem for the integer benchmarks. The significance of sync arcs providing comparable performance to the MCB approach is that this indicates the memory disambiguation provided by sync arcs is very accurate. The perfect MCB case essentially has perfect disambiguation, because it was allowed to reorder all load/store pairs for which a definite dependence could not be proved. For the sync arcs to achieve comparable performance to this perfect disambiguation requires that the sync arc information be very accurate.

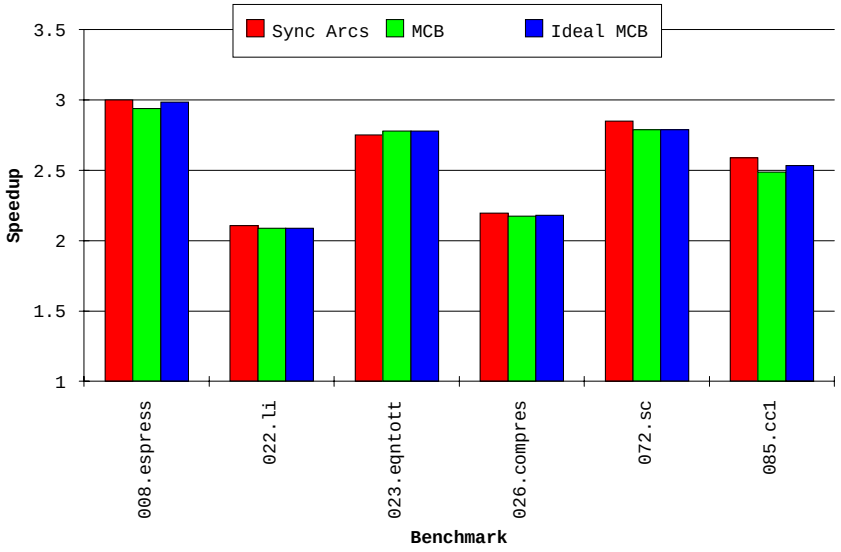


Figure 7.21 SPEC-CINT92 Comparison of Sync Arcs to MCB - 8-Issue.

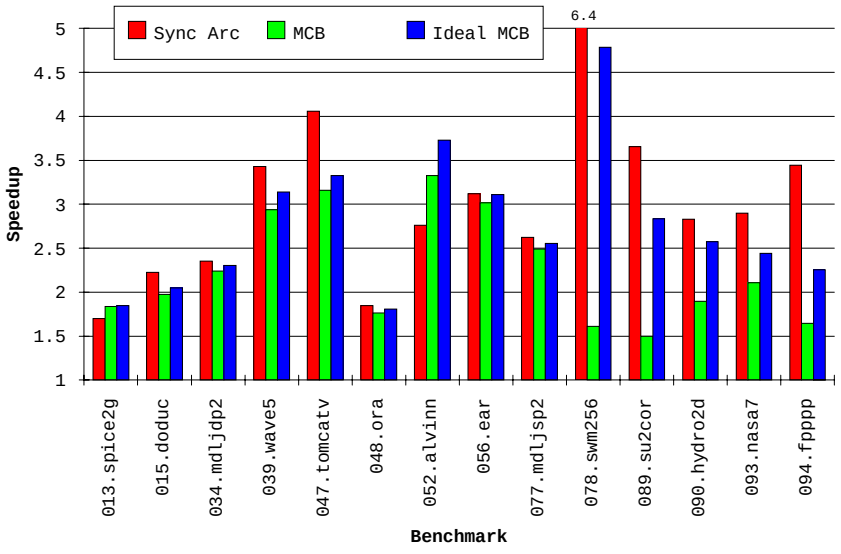


Figure 7.22 SPEC-CFP92 Comparison of Sync Arcs to MCB - 8-Issue.

```
sender = &input_act[0];
end_sender= &input_act[NIU];

for (; sender <= end_sender; )
    (*w_ch++) += (*delta) * (*sender++);
```

Figure 7.23 Source Code from *052.alvinn*.

For the floating-point benchmarks, the sync arc code again compared well with the MCB approach. In many cases, the sync arc code outperformed the ideal MCB by a significant margin. The primary exception to this is *052.alvinn*. For this benchmark, the sync arc performance was hindered by a limitation in the current Pcode C dependence analysis. Figure 7.23 illustrates the reason for this result, showing a loop representative of the two most important inner loops of *052.alvinn*. These loops use a *for-loop* structure in which the loop induction variable and the upper bound are not readily obvious. The loop bodies contain no cross-iteration dependences. Pcode dependence analysis is currently unable to cast this loop into a format which can be interfaced to the Omega Test and is, thus, unable to provide accurate dependences for the array accesses in these loops. As a result, cross-iteration dependences are conservatively assumed, and sync arc performance is degraded. Improved dependence analysis would be expected to improve the relative sync arc performance for other benchmarks as well (e.g., *013.spice2g6* and *056.ear*).

The primary reason the sync arc approach performs better for many benchmarks is that the MCB approach is hindered by the instruction overhead of the additional check instructions. Although in some cases the wide-issue architecture may be able to "hide" the effect of these extra instructions, they often result in some loss of performance. In many cases, this performance loss is minor. The one exception from the integer benchmark suite is *cmp*, which performed

significantly worse for the MCB code. This can be understood by referring to Figure 7.6(a), which shows the non-optimized inner loop of cmp. The loop, which contains 26 instructions (11 of which are loads), is unrolled 8 times prior to scheduling. During scheduling, almost all loads were speculated above stores, requiring the insertion of 83 check instructions, increasing the instruction count of the unrolled loop from 206 instructions to 291. Not only was the overall instruction count increased, but the number of branches was increased from 40 to 123 by the checks. The code scheduler was limited by the available branch resources, and the length of the schedule was increased over the sync arc case. For the floating-point code, the performance loss due to the additional check instructions was much more pronounced. This result points to the need for improved MCB heuristics which trade off the relative cost and potential benefit for allowing loads to bypass ambiguous stores during MCB scheduling.

Another reason the sync arc approach performs better than the MCB approach is the increased register pressure incurred by the MCB approach. This register pressure increase is the result of the lengthening of register live ranges. The destination register of a preload instruction cannot be used again by another preload until after the check instruction for the first preload has been executed. This requirement essentially lengthens the live range of preload instructions to include the associated check, increasing register pressure.

7.2.2 Synergy of the approaches

Although static and dynamic memory disambiguation are targeted for different applications, the approaches should not be viewed as mutually exclusive. Sync arcs provide a means of better disambiguation with no instruction overhead, but may be ineffective at providing accurate disambiguation in some cases (e.g., sparse matrices and complex pointer accesses). Dynamic

approaches such as MCB require instruction overhead and code growth, but provide perfect disambiguation. A combination of the two approaches, which uses static disambiguation information when it is accurate and applies a dynamic technique where the static is limited, might provide an optimal approach. Because the dynamic technique would only be applied in limited areas of the code, the instruction overhead and code growth would be greatly reduced.

To apply the combined approach, the static dependence analysis would have to specify the accuracy of individual sync arcs. Dependences for which the static analysis is accurate would have to be differentiated from those when the analysis failed and the arc was added conservatively. Additionally, if the dependence analysis is able to determine how frequently the dependence holds, this information should be made available within the sync arc. Within the low-level code, sync arcs which reflect low accuracy or infrequent dependence can be removed by applying the dynamic technique.

Although this combined approach is viable, it is questionable how often the inaccurate or infrequent dependences would occur in actual code. The sync arc results presented earlier in this chapter indicate the dependence information is sufficiently accurate to allow aggressive optimization and scheduling for the benchmarks tested.

7.3 Summary of Results

In this chapter, the potential benefit of the sync arc approach to low-level optimization and scheduling has been measured across a suite of 29 integer and floating-point benchmarks. The sync arc technique is shown to significantly improve the compiler's ability to enhance and exploit ILP. Good performance improvements are seen for those benchmarks for which ambiguous memory dependences were a significant impediment. As expected, the sync arc technique

provides little benefit to programs such as *023.eqntott* for which memory dependences are not a problem. Evaluation of the importance of improved memory disambiguation to optimization and scheduling individually indicated that both benefit significantly from sync arcs. Although sync arcs increased ILP for many of the benchmarks tested, the compiler was still limited in its ability to expose sufficient parallelism to the hardware to make efficient use of available resources. Techniques to further reduce the impact of control flow instructions on parallelism, such as predicated execution, are required to further expose ILP.

Sync arcs are a viable technique for improving memory disambiguation and facilitating low-level optimization and scheduling. In Chapter 4, the MCB technique was also shown to be an effective dynamic approach. In this chapter, the two techniques were quantitatively compared and found to provide nearly equivalent performance improvement for most benchmarks. The advantage the sync arc technique had over MCB for several of the floating-point benchmarks was the result of limitations in the MCB scheduling algorithm. Thus, both approaches provide excellent disambiguation, and the most appropriate technique can be applied to a particular application. Because the MCB technique requires special hardware support and incurs instruction overhead, the sync arc technique may be a good choice when both approaches are equally viable. The two approaches could potentially be combined to provide even better memory disambiguation for certain applications.

CHAPTER 8

CONCLUSIONS

8.1 Summary

Ambiguous memory dependences can significantly impact the compiler's ability to expose instruction-level parallelism by preventing important optimization and scheduling opportunities. This dissertation has explored two approaches for overcoming the problems posed by ambiguous memory dependences: dynamic memory disambiguation and static memory disambiguation. Selected techniques from the two approaches were analyzed individually. To perform these analyses, the techniques were fully implemented within the IMPACT compiler environment and detailed simulation was performed. Additionally, a quantitative comparison of the relative merits of the two approaches has been provided.

This dissertation has explored a previously proposed dynamic technique, the *memory conflict buffer*. The MCB technique employs a set of hardware features to perform explicit comparisons between load and store addresses. The compiler takes advantage of this hardware to speculatively reorder load and store operations during code scheduling. In this dissertation, a new hardware design is proposed and a detailed evaluation of the potential performance benefits is performed. Results indicate that MCB is effective at removing ambiguous memory dependences as an impediment to ILP.

This dissertation has proposed a technique for providing improved static memory disambiguation to support ILP compilation. The technique, *sync arcs*, uses detailed source-level

dependence analysis to generate explicit dependence arcs, which are maintained through subsequent compilation. The dependence information required to support ILP compilation is investigated, as well as the difficulty of maintaining sync arcs through code transformations. Several methods of limiting the number of explicit sync arcs which must be maintained are also proposed. The sync arc technique is evaluated using a suite of 29 integer and floating-point benchmarks. Results indicate the approach is highly effective at increasing performance where ambiguous memory dependences previously inhibited ILP.

In order to apply the sync arc technique to C programs, the existing IMPACT dependence analysis has been modified to handle C semantics. The challenges the C language poses to dependence analysis and some of the interesting implementation details for supporting sync arcs are discussed. Because the C language allows pointer aliasing, interprocedural analysis is necessary to avoid overly conservative analysis. A coarse-grain interprocedural analysis is proposed and shown to be effective for supporting ILP compilation. For the suite of 15 integer benchmarks, it is shown that a coarse-grain analysis provides comparable performance to that for an ideal analysis.

Finally, the tradeoffs between dynamic and static memory disambiguation approaches have been examined. The simulation results from the MCB technique and the sync arc technique revealed both techniques provided effective disambiguation. Interestingly, the static sync arc technique provided results which were as good or better than the dynamic technique for most benchmarks. For applications for which source-level information is available during compilation, the sync arc technique may be a good choice for providing improved memory disambiguation to low-level optimization and scheduling. However, dynamic techniques remain a viable alternative

for applications which require extremely fast compilation or when source-level information is unavailable.

8.2 Future Work

The results shown in this dissertation indicate that both static and dynamic approaches warrant further study. The dynamic technique explored in this dissertation, the memory conflict buffer, supports ILP compilation well: it performs well in the presence of aggressive code reordering and allows a load's dependent operations to bypass store operations. However, the current MCB design still suffers from both true and false conflicts, limiting performance gains. Memory dependence profiling could potentially overcome the problem of true conflicts. Further work is needed to develop a hardware design which avoids false conflicts, yet is practical in size and timing. A promising area of research is a fully associative design whose size is independent of the number of architectural registers. The compiler would have to be responsible for limiting the number of preloads which were simultaneously "live" to the size of the MCB array. An additional limitation of the MCB technique as employed in this dissertation is that the technique was only applied to low-level code scheduling. The potential benefit of the MCB technique to low-level optimization also needs to be quantified.

The sync arc technique also shows great promise as a means of improving memory disambiguation for low-level code. This dissertation demonstrates sync arcs can be accurately maintained through aggressive code transformations and that the availability of sync arcs provides significant performance improvement. However, further work is needed to better understand how to control the number of explicit dependence arcs which must be maintained. Several

techniques are proposed, but the relative benefits of these techniques for reducing explicit arcs have not been quantified.

The importance of disambiguating memory dependences to ILP compilation increases as the impact of control instructions is reduced. This dissertation studied ILP compilation in the context of the superblock, which coalesces instructions from a single control flow path. For architectures which support predicated execution, techniques are being studied for coalescing multiple paths of control into a single compilation unit [23], effectively removing many control flow instructions. This reduction in the number of control instructions should make memory disambiguation even more critical. Further research is required to understand the importance of both static and dynamic disambiguation approaches when compiling using predication.

The static memory disambiguation techniques employed in this dissertation rely on the use of source-level information to aid dependence analysis. Although dependence analysis is somewhat easier when source-level information is available, it may be possible to perform relatively accurate dependence analysis on the low-level code without help from the source level. If analysis can be accurately performed on low-level code, then the same disambiguation technique used during compilation of source code could also be applied to applications for which the source is not available, such as binary translation. While it appears that some accuracy may be lost in this type of analysis (as compared with using source-level information), it would be important to better understand and quantify the impact of this accuracy loss for ILP compilation.

REFERENCES

[1] W. Y. Chen, "Data preload for superscalar and VLIW processors," Ph.D. dissertation, Department of Electrical and Computer Engineering, University of Illinois, Urbana, IL, 1993.

[2] S. I. Feldman, D. M. Gray, M. W. Maimore, and N. L. Schryer, "A Fortran-to-C converter," Computing Science Tech. Rep. 149, AT&T Bell Laboratories, Murray Hill, NJ, June 1990.

[3] G. E. Haab, "Data dependence analysis for Fortran programs in the IMPACT compiler," M.S. thesis, Department of Electrical and Computer Engineering, University of Illinois, Urbana, IL, 1995.

[4] K. Subramanian, "Loop transformations for parallel compilers," M.S. thesis, Department of Computer Science, University of Illinois, Urbana, IL, 1993.

[5] S. Anik, "Architectural and software support for executing numerical applications on high performance computers," Ph.D. dissertation, Department of Electrical and Computer Engineering, University of Illinois, Urbana, IL, 1993.

[6] Y. Yamada, "Data relocation and prefetching for programs with large data sets," Ph.D. dissertation, Department of Computer Science, University of Illinois, Urbana, IL, 1995.

[7] W. W. Hwu and P. P. Chang, "Achieving high instruction cache performance with an optimizing compiler," in *Proceedings of the 16th International Symposium on Computer Architecture*, pp. 242–251, May 1989.

[8] P. P. Chang, "Compiler support for multiple instruction issue architectures," Ph.D. dissertation, Department of Electrical and Computer Engineering, University of Illinois, Urbana, IL, 1991.

[9] 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.

[10] J. C. Gyllenhaal, "A machine description language for compilation," M.S. thesis, Department of Electrical and Computer Engineering, University of Illinois, Urbana, IL, 1994.

[11] R. A. Bringmann, "Template for code generation development using the IMPACT-I C compiler," M.S. thesis, Department of Computer Science, University of Illinois, Urbana, IL, 1992.

[12] W. Y. Chen, "An optimizing compiler code generator: A platform for RISC performance analysis," M.S. thesis, Department of Electrical and Computer Engineering, University of Illinois, Urbana, IL, 1991.

[13] R. G. Ouellette, "Compiler support for SPARC architecture processors," M.S. thesis, Department of Electrical and Computer Engineering, University of Illinois, Urbana, IL, 1994.

[14] B. T. Sander, "Performance optimization and evaluation for the impact x86 compiler," M.S. thesis, Department of Electrical and Computer Engineering, University of Illinois, Urbana, IL, 1995.

[15] W. F. Dugal, "Code scheduling and optimization for a superscalar x86 microprocessor," M.S. thesis, Department of Electrical and Computer Engineering, University of Illinois, Urbana, IL, 1995.

[16] 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 94303, February 1994.

[17] D. M. Lavery, "Unrolling-based optimizations for software pipelining," Tech. Rep. IMPACT-95-07, Center for Reliable and High-Performance Computing, University of Illinois, Urbana, IL, June 1995.

[18] N. J. Warter, "Modulo scheduling with isomorphic control transformations," Ph.D. dissertation, Department of Electrical and Computer Engineering, University of Illinois, Urbana, IL, 1993.

[19] A. Aho, R. Sethi, and J. Ullman, *Compilers: Principles, Techniques, and Tools*. Reading, MA: Addison-Wesley, 1986.

[20] W. Pugh, "A practical algorithm for exact array dependence analysis," *Communications of the ACM*, vol. 35, pp. 102–114, Aug. 1992.

[21] S. A. Mahlke, "Design and implementation of a portable global code optimizer," M.S. thesis, Department of Electrical and Computer Engineering, University of Illinois, Urbana, IL, 1991.

[22] R. A. Bringmann, "Compiler-controlled speculation," Ph.D. dissertation, Department of Computer Science, University of Illinois, Urbana, IL, 1995.

[23] S. A. Mahlke, "Exploiting instruction level parallelism in the presence of conditional branches," Ph.D. dissertation, Department of Electrical and Computer Engineering, University of Illinois, Urbana, IL, 1995.

[24] P. P. Chang, D. M. Lavery, and W. W. Hwu, "The importance of prepass code scheduling for superscalar and superpipelined processors," Tech. Rep. CRHC-91-18, Center for Reliable and High-Performance Computing, University of Illinois, Urbana, IL, May 1991.

[25] 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, Urbana, IL, 1993.

[26] E. M. Riseman and C. C. Foster, "The inhibition of potential parallelism by conditional jumps," *IEEE Transactions on Computers*, vol. c-21, pp. 1405–1411, December 1972.

[27] 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*, pp. 290–302, April 1989.

[28] 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*, pp. 272–282, April 1989.

[29] 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.

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

[31] P. P. Chang, S. A. Mahlke, and W. W. Hwu, "Using profile information to assist classic code optimizations," *Software Practice and Experience*, vol. 21, pp. 1301–1321, December 1991.

[32] J. W. C. Fu and J. H. Patel, "How to simulate 100 billion references cheaply," Tech. Rep. CRHC-91-30, Center for Reliable and High-Performance Computing, University of Illinois, Urbana, IL, 1991.

[33] S. A. Mahlke, W. Y. Chen, R. A. Bringmann, R. E. Hank, W. W. Hwu, B. R. Rau, and M. S. Schlansker, "Sentinel scheduling: A model for compiler-controlled speculative execution," *Transactions on Computer Systems*, vol. 11, November 1993.

[34] 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.

[35] 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*, pp. 248–259, October 1992.

[36] A. Nicolau, "Run-time disambiguation: coping with statically unpredictable dependencies," *IEEE Transactions on Computers*, vol. 38, pp. 663–678, May 1989.

[37] W. Y. Chen, S. A. Mahlke, W. W. Hwu, T. Kiyohara, and P. P. Chang, "Tolerating data access latency with register preloading," in *Proceedings of the 6th International Conference on Supercomputing*, July 1992.

[38] H. Company, "Memory processor that permits aggressive execution of load instructions," *UK Patent GB 2 265 481 A*, pending (submitted February 1993).

[39] A. S. Huang, G. Slavenburg, and J. P. Shen, "Speculative disambiguation: A compilation technique for dynamic memory disambiguation," in *Proceedings of the 21st International Symposium on Computer Architecture*, pp. 200–210, April 1994.

[40] G. M. Silberman and K. Ebcioglu, "An architectural framework for supporting heterogeneous instruction-set architectures," *IEEE Computer*, pp. 39–56, June 1993.

[41] R. M. Tomasulo, "An efficient algorithm for exploiting multiple arithmetic units," *IBM Journal of Research and Development*, vol. 11, pp. 25–33, January 1967.

[42] J. E. Thornton, *Design of a Computer - The Control Data 6600.* Glenview, IL: Scott, Foresman, and Co., 1970.

[43] W. W. Hwu, "Exploiting concurrency to achieve high performance in a single-chip microarchitecture," Ph.D. dissertation, Computer Science Division, University of California, Berkeley, CA, 1988.

[44] M. Franklin and G. S. Sohi, "ARB: A hardware mechanism for dynamic memory disambiguation," *IEEE Transactions on Computers*, to be published, 1996.

[45] M. J. Wolfe, "Optimizing compilers for supercomputers," Ph.D. dissertation, Department of Computer Science, University of Illinois, Urbana, IL, 1982.

[46] U. Banerjee, *Dependence Analysis for Supercomputing.* Boston, MA: Kluwer Academic Publishers, 1988.

[47] G. Goff, K. Kennedy, and C.-W. Tseng, "Practical dependence testing," in *Proc. 1991 SIGPLAN Symp. Compiler Construction*, pp. 15–29, June 1991.

[48] D. E. Mayden, J. L. Hennessy, and M. S. Lam, "Efficient and exact data dependence analysis," in *Proc. 1991 SIGPLAN Symp. Compiler Construction*, pp. 1–14, June 1991.

[49] 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.

[50] E. Duesterwald, R. Gupta, and M. L. Soffa, "A practical data flow framework for array reference analysis," in *Proceedings of the ACM SIGPLAN '93 Conference on Programming Language Design and Implementation*, pp. 68–77, June 1993.

[51] J. Banning, "An efficient way to find the side effects of procedure calls and the aliases of variables," in *Proceedings of the 6th ACM Symposium on Principles of Programming Languages*, pp. 29–41, January 1979.

[52] K. Cooper, "Analyzing aliases of reference formal parameters," in *Proceedings of the 12th ACM Symposium on Principles of Programming Languages*, pp. 281–290, January 1985.

[53] W. E. Weihl, "Interprocedural data flow analysis in the presence of pointers, procedure variables and label variables," in *Proceedings of the 7th ACM Symposium on Principles of Programming Languages*, pp. 83–94, January 1980.

[54] W. Landi and B. G. Ryder, "A safe approximate algorithm for interprocedural pointer aliasing," in *Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation*, pp. 235–248, June 1992.

[55] A. Deutsch, "Interprocedural may-alias analysis for pointers: Beyond k-limiting," in *Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation*, pp. 230–241, June 1994.

[56] J. R. Larus and P. N. Hilfinger, "Detecting conflicts between structure accesses," in *Proceedings of the ACM SIGPLAN '88 Conference on Programming Language Design and Implementation*, pp. 21–34, July 1988.

[57] L. J. Hendren and A. Nicolau, "Parallelizing programs with recursive data structures," *IEEE Transactions on Parallel and Distributed Computing*, vol. 1, pp. 35–47, Jan. 1990.

[58] L. Hendren, J. Hummel, and A. Nicolau, "Abstractions for recursive pointer data structures: Improving the analysis and transformation of imperative programs," in *Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation*, pp. 249–260, June 1992.

[59] J. Hummel, L. J. Hendren, and A. Nicolau, "A general data dependence test for dynamic, pointer-based data structures," in *Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation*, pp. 218–229, June 1994.

[60] B. R. Rau, "Pseudo-randomly interleaved memory," in *Proceedings of 18th Internation Symposium on Computer Architecture*, pp. 74–83, May 1991.

[61] 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*, pp. 180–192, April 1987.

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

[63] R. E. Hank, "Region-based compilation to support ILP processing," Tech. Rep. IMPACT-95-06, Center for Reliable and High-Performance Computing, University of Illinois, Urbana, IL, June 1995.

[64] J. Barth, "Interprocedural data flow analysis based on transitive closure," Tech. Rep. UCB-CS-76-44, Computer Science Dept., University of California at Berkeley, September 1976.

[65] W. Landi and B. G. Ryder, "Pointer-induced aliasing: A problem classification," in *Proceedings of the 18th ACM Symposium on Principles of Programming Languages*, pp. 235–248, January 1991.

VITA

David Mark Gallagher was born in Stillwater, Oklahoma, in 1956. He pursued his undergraduate studies at the United States Air Force Academy in Colorado Springs, Colorado where he received the B.S. degree in Electrical Engineering in 1978. After receiving the B.S. degree, he joined the United States Air Force. In the Air Force, he served as a senior pilot, and has advanced to the rank of Lieutenant Colonel. In 1987, he completed the M.S. degree in Electrical Engineering at the Air Force Institute of Technology in Dayton, Ohio. In the fall of 1992, he began his graduate studies in Electrical Engineering at the University of Illinois in Urbana, Illinois. During his tenure at the University of Illinois, he has been a member the IMPACT project directed by Professor Wen-mei W. Hwu. After completing the Ph.D. work, he will continue to serve in the Air Force as an Assistant Professor at the Air Force Institute of Technology.