EXPLOITING INSTRUCTION LEVEL PARALLELISM IN THE PRESENCE OF CONDITIONAL BRANCHES

BY

SCOTT ALAN MAHLKE

B.S., University of Illinois, 1988
M.S., University of Illinois, 1991

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, 1996

Urbana, Illinois

EXPLOITING INSTRUCTION LEVEL PARALLELISM
IN THE PRESENCE OF CONDITIONAL BRANCHES

Scott Alan Mahlke, Ph.D.
Department of Electrical and Computer Engineering
University of Illinois at Urbana-Champaign, 1996
Wen-mei W. Hwu, Advisor

Wide issue superscalar and VLIW processors utilize instruction-level parallelism (ILP) to achieve high performance. However, if insufficient ILP is found, the performance potential of these processors suffers dramatically. Branch instructions, which are one of the major limitations to exploiting ILP, enforce strict ordering conditions in programs to ensure correct execution. Therefore, it is difficult to achieve the desired overlap of instruction execution with branches in the instruction stream. To effectively exploit ILP in the presence of branches requires efficient handling of branches and the dependences they impose.

This dissertation investigates two techniques for exposing and enhancing ILP in the presence of branches, *speculative execution* and *predicated execution*. Speculative execution enables an ILP compiler to remove dependences between instructions and prior branches. In this manner, the execution of instructions and predicted future instructions may be overlapped. Compiler-controlled speculative execution is employed using an efficient structure called the *superblock*. The formation and optimization of superblocks increase ILP along important execution paths by systematically removing constraints due to unimportant paths. In conjunction with superblock optimizations, speculative execution is utilized to remove control dependences in the superblock to aggressively reorder instructions across branches to achieve a high degree of execution overlap.

For many applications, speculative execution alone is not sufficient to achieve high performance. The fundamental limitation is that speculation only removes dependences between branches and other instructions. The branches themselves remain in the code, which causes difficult problems. This motivates the second technique investigated in this dissertation, predicated execution, which is an architectural capability that enables the conditional execution of instructions based on the value of a Boolean source operand. Predicated execution allows a compiler to eliminate branch instructions using this conditional execution support. Additionally, predicated execution provides an efficient interface for the compiler to overlap the execution of multiple paths of control. Predicated execution is exploited in the compiler via a generalized form of a superblock, called the *hyperblock*. Hyperblocks provide the framework for the compiler to selectively eliminate branches using predicated execution as well as apply speculative execution to exploit ILP.

DEDICATION

Dedicated to the fond memory of my grandfather, Richard Bannon.

ACKNOWLEDGMENTS

First and foremost, I would like to thank my advisor, Professor Wen-mei Hwu, for his guidance throughout my graduate studies. Most importantly, I would like to thank him for his patience. I think I truly tested the limits of this patience on several occasions. But through everything, he provided me with continued encouragement and support.

Next, I would like to extend my gratitude to the members of my dissertation committee, Professor Janak Patel, Dr. Bob Rau, and Professor Pen-Chung Yew. Their numerous comments, questions, and suggestions improved the quality of this work immensely. Also, I would like to thank Vinod Kathail, Bob Rau, and Mike Schlansker at Hewlett-Packard Laboratories. Their teaching and suggestions had a strong influence on the directions of this work.

This research truly would not have been possible without the support, hard work, and friendship of the members of the IMPACT research group. Members of the group were always there to discuss ideas, debate solutions, practice talks, and develop software. I feel extremely fortunate to have been a part of this group. I would first like to thank two members of the group, Pohua Chang and William Chen. Pohua put a great deal of time and effort in to educating me in the area of ILP compilation. His energy and endless supply of ideas provided a strong motivation for my work. William was a close friend and colleague throughout graduate school. He was always there to discuss research, brainstorm new ideas, and provide helpful suggestions.

The group members for which I owe many thanks to are those who worked on hyperblocks and predicated execution. The research really began with David Lin, who helped formulate most of the original ideas for hyperblocks and predicate compilation. Rick Hank, John Gyllenhaal, and Roger Bringmann contributed to almost every aspect of the research with their ideas, insight, and suggestions. Rick's work on code generation, dataflow analysis, register allocation, and emulation was central to this research. John's work on emulation, profiling, and simulation was equally important. Roger's research provided the scheduling framework used in this dissertation. Dave Gallagher was a willing sounding board for all of my ideas and was always there to debate any issue. He also provided important work on the predicate analysis modules. Jim McCormick contributed his thoughts and effort in the area of partial predication. Finally, David August provided valuable contributions with his ideas and work on loop peeling and hyperblock optimization. I would also like to thank him for all the invaluable comments and suggestions he provided on this dissertation.

There are several other group members that I wish to thank. My officemates, Sadun Anik, Tom Conte, Dave Gallagher, and Nancy Warter made office life very enjoyable with their thoughts and discussions. Grant Haab, Sabrina Hwu, Tokuzo Kiyohara, and Dan Lavery provided invaluable feedback on ideas, papers, and talks. Dan Connors, Brian Deitrich, Cheng-Hsueh Hsieh, and Teresa Johnson provided helpful views and software tools.

Next, I would like to extend special thanks to my friends on Copper, Black Knight, Sojourn, and Shayol-Ghul Dikumuds. Although at times mud bordered on an addiction, it provided a much needed escape from the harsh realities of life. Many thanks to Mookie, Allenbri, Old, Tryth, Tang, Orcus, Namu, Ima, Cucumber, Dragnar, Proteus, Cython, and Miax.

Last, I would like to acknowledge the support of some friends and family. Brian Upper, Brad Gilbert, Tom Begnel, Jim Falling, and all the Groundhogs provided many good times through my days in Illinois. My parents, Jeanne and Monte, and my sister, Laura, gave me the encouragement and consistent support that I needed to make it through graduate school.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

Superscalar and very long instruction word (VLIW) processors can potentially provide large performance improvements over their scalar predecessors by providing multiple data paths and functional units. The parallel resources are exploited by concurrently executing independent instructions from the instruction stream. However, conditional branch instructions pose difficult problems for all types of processors that exploit instruction-level parallelism (ILP). Recent studies have shown that by using conventional code optimization and scheduling methods, superscalar and VLIW processors cannot produce a sustained speedup of more than two for nonnumeric programs [1],[2],[3]. For such programs, conventional architectural and compilation methods do not provide enough support to utilize these processors.

Branch instructions are the major impediment to exploiting ILP in nonnumeric applications. There are several reasons this occurs. First, the amount of ILP within basic blocks is extremely limited due to the small number of instructions in each block. For nonnumeric benchmarks, researchers report that approximately 20% to 30% of the dynamic instructions are branches. This results in an average basic block size for these programs of three to five instructions. As a result, compiler and/or hardware ILP techniques must look beyond the basic block boundaries to find sufficient parallelism.

Branch prediction is the most widely applied technique to extract ILP across basic blocks. There are two basic classes of branch prediction strategies: *static* and *dynamic*. Static branch prediction utilizes information available at compile time to make predictions. Several example static prediction schemes are branch direction (backward taken, forward not taken) [4]; heuristics based on the program structure [5],[6]; and profile information [7],[8],[9]. For compilers employing scheduling techniques such as trace scheduling [10], static branch prediction is used to identify likely sequences of basic blocks that can be scheduled as single units. Dynamic branch prediction utilizes run-time behavior to make predictions. Example dynamic branch prediction schemes are the branch target buffer (BTB) with a 2-bit saturating counter [11] and two-level adaptive training [12],[13],[14]. For processors employing hardware scheduling, dynamic branch prediction is used to identify a continuous window of instructions from which instructions are selected for execution.

Regardless of the prediction strategy employed, the software or hardware scheduler is presented with a larger block of instructions, enabling it to expose a greater amount of ILP. While correct branch prediction can increase ILP, incorrect prediction often results in large performance penalties. These large performance penalties or misprediction penalties are the second reason that branches are the major impediment to exploiting ILP. Recent studies comparing perfect branch prediction with a highly accurate dynamic predictor show that a small fraction of mispredictions can reduce performance by a factor of two to more than ten [15],[16].

The final reason branches impede ILP so severely is that frequent branches in the instruction stream place an upper limit on the potential ILP. A superscalar or VLIW processor may have to execute multiple branches per cycle to sustain the execution of multiple instructions per cycle. Under the assumption that an instruction stream contains 25% branches, an issue-8 superscalar processor must have the capability to sustain at least two branches per cycle. If the issue-8 processor could only execute a single branch each cycle, its maximal performance would be resource limited to four instructions per cycle. Handling multiple branches per cycle requires additional pipeline complexity, as well as designing multiported branch prediction structures such as the branch target buffer (BTB). In high issue rate processors, it is much easier to duplicate arithmetic function units than to predict and execute multiple branches per cycle. Therefore, most future generation ILP processors will likely have limited branch handling capabilities which may limit performance in nonnumeric applications.

To exploit and extract ILP beyond the basic block boundary requires efficient handling of branches and the control dependences imposed by branches. In this dissertation, two techniques for exploiting and enhancing ILP in the presence of conditional branches are investigated, speculative execution and predicated execution. Speculative execution refers to the execution of an instruction before knowing that its execution is required. Speculative execution enables the compiler or hardware to remove dependences between instructions and prior branches. In this manner, the execution of instructions and predicted future instructions may be overlapped. For the case in which the predicted future instructions were indeed required to execute, a large potential performance gain may be observed. For the contrary case, the speculated instructions are unnecessarily executed and their side effects must be nullified.

Compiler-controlled speculative execution is employed using an efficient structure called the superblock. Superblocks provide an efficient foundation for all phases of ILP compilation, including optimization, scheduling, and register allocation. The formation and optimization of superblocks increase the ILP along the important execution paths by systematically removing constraints due to the unimportant paths. Speculative execution enables the scheduler to remove the control dependences between instructions and branches in a superblock. In conjunction with the data dependences removed with superblock optimization, speculative execution is utilized to efficiently extract ILP across basic block boundaries.

The second technique investigated in this dissertation is predicated execution. Predicated execution support provides an effective means to completely eliminate branches from an instruction stream. Predicated or guarded execution refers to the conditional execution of an instruction based on the value of a Boolean source operand, referred to as the predicate of the instruction [17],[18]. This architectural support allows the compiler to use an *if-conversion* algorithm to convert conditional branches into predicate defining instructions and instructions along alternative paths of each branch into predicated instructions [19],[20],[21]. Predicated instructions are fetched regardless of their predicate value. Instructions whose predicate value is true are executed normally. Conversely, instructions whose predicate is false are nullified, and thus are prevented from modifying the processor state. Predicated execution allows the compiler to trade instruction fetch efficiency for the capability to expose ILP to the hardware along multiple execution paths.

Predicated execution offers the opportunity to improve branch handling in superscalar and VLIW processors. Eliminating frequently mispredicted branches may lead to a substantial reduction in branch prediction misses. As a result, the performance penalties associated with the eliminated branches are removed. Eliminating branches also reduces the need to handle multiple branches per cycle for wide issue processors. As a side effect of reducing the number of branches in the instruction stream, the amount of speculation required to sustain full processor utilization is reduced. Therefore, in the case of a mispredicted branch, fewer speculative instructions must be discarded. Finally, predicated execution provides an efficient interface for the compiler to expose multiple execution paths to the hardware. Without compiler support, the cost of maintaining multiple execution paths in hardware grows rapidly.

Predicated execution support is exploited in the compiler via a structure called the hyperblock. Hyperblocks are a generalized form of superblocks which take advantage of both speculative and predicated execution. The major difference between hyperblocks and superblocks is that an arbitrary number of paths of control may be combined into a single unit with hyperblocks, whereas, with superblocks, a single path of control is identified for ILP compilation. The goal of hyperblock compilation techniques is to intelligently group basic blocks from many different control flow paths into a single manageable structure for compiler optimization and scheduling. Basic blocks are systematically selected for inclusion in hyperblocks to eliminate hard-to-predict branches, maximize ILP optimization opportunities, and avoid over-committing processor resources. Speculative execution is enabled in hyperblocks using a technique called predicate promotion, which removes the predicate from selected instructions to allow execution before its predicate is calculated, thereby speculating the instruction.

1.1 Contributions

The four major contributions of this dissertation are discussed below.

- The superblock compilation techniques discussed in this thesis provide an efficient paradigm for ILP compilation. Superblocks differ from most ILP compilation techniques in that they provide an underlying structure for both optimization as well as scheduling and register allocation. Most techniques such as trace scheduling [10] and region scheduling [22] do not extend efficiently to optimization. The major architectural feature exploited to achieve high performance with superblocks is speculative execution. In the area of superblock compilation, the contributions specific to this thesis lie primarily in the areas of superblock classical optimizations, superblock ILP optimizations, and speculative execution using the sentinel scheduling model.

- The hyperblock structure proposed in this thesis provides an effective framework for ILP compilation of nonnumeric programs. Hyperblocks combine the use of predicated and speculative execution to eliminate branches from the instruction stream as well as eliminate dependences for control flow. Hyperblocks offer several distinct advantages over superblocks. First, they allow the compiler to expose parallelism along multiple execution paths to the hardware. Second, they allow the compiler to eliminate a large portion of the dynamic branches from applications, thereby eliminating the need to predict these branches. Finally, they allow the compiler to selectively apply if-conversion to balance the elimination of branch instructions with processor resource availability to ensure that resources are not over-saturated.

- An approach to efficiently extend traditional compiler optimizations, scheduling, and register allocation to operate effectively on predicated code is proposed in this thesis. Predicated code introduces new challenges for compilers since a large portion of the control flow is no longer explicit, but rather represented implicitly by predicates. The fundamental tool utilized by the compiler to understand predicate semantics is called the predicate hierarchy graph. It provides information regarding the relationships among predicates, such as mutual exclusion and covering. Using the predicate information, an instruction-level control flow graph may be constructed to represent the implicit and explicit control flows in predicated code. Traditional compiler transformations use a combination of this control flow graph as well as the predicate relations themselves to effectively handle predicated code. In addition, a set of predicate-specific optimizations has been developed to further improve the performance predicated code. These optimizations utilize the inherent functionality of predicated execution to expose optimization opportunities which are more difficult in traditional architectures. The optimizations include loop peeling, branch combining, instruction merging, and predicate promotion.

- A detailed evaluation of the usefulness of predicated execution in ILP architectures for nonnumeric programs is performed in this thesis. Most research and development have focused on using predicated execution numerical applications in the vector or VLIW domain. In this thesis, it is shown that predicated execution supported by hyperblock compilation techniques is highly effective for nonnumeric applications. The ability of the compiler to eliminate problematic branches and overlap the execution of multiple paths of control with predicated execution is assessed. In addition, a comprehensive evaluation of the relative benefits and limitations of speculative and predicated execution is made in this thesis. The evaluation is unique in the sense that all compiler support for both techniques has been implemented in the compiler, enabling a fair comparison and contrast of the techniques within the scope of a single compiler framework.

1.2 Overview

This dissertation is composed of nine chapters. Chapter 2 presents an overview of the organization and operation of the IMPACT compiler. All compiler techniques discussed in this thesis are implemented within the framework of the IMPACT compiler.

The first technique investigated to deal with branches in ILP processors, speculative execution, is presented in Chapter 3. The superblock is the basic underlying structure utilized by the compiler to extract, enhance, and exploit ILP. Chapter 3 discusses the superblock along with superblock optimization and scheduling techniques. In Chapter 4, a new scheduling model, sentinel speculation, is introduced. Sentinel speculation provides an effective framework for full speculation of instructions, along with an efficient set of mechanisms to detect and recover from all exceptions for speculative instructions.

Chapter 5 provides an experimental evaluation of speculative execution using superblock compilation techniques. The relative benefits of limited versus full speculation are assessed along with the effects of the superblock ILP code transformations. Performance limitations of speculative execution alone are also addressed.

The second technique for dealing with branches in ILP processors, predicated execution, is presented in Chapter 6. Predicated execution enables the compiler to eliminate branches from the instruction stream. Efficient architecture and instruction set architecture extensions for predicated execution are the major focus of this chapter. The proposed compiler support for utilizing predicated execution is given in Chapter 7. Compiler support is based on the hyperblock structure. Hyperblock formation, extensions to superblock optimization and scheduling techniques, and predicate-specific ILP optimizations are discussed in this chapter.

A second set of experiments is performed and analyzed in Chapter 8. This set of experiments evaluates the effectiveness of predicated execution using hyperblock compilation techniques. A full comparison between speculative execution and predicated execution is also done. Finally, in Chapter 9, conclusions and directions for future research are given.

CHAPTER 2

OVERVIEW OF THE IMPACT COMPILER

All of the compiler techniques to utilize speculative and predicated execution are implemented within the framework of the IMPACT compiler. A block diagram of the IMPACT compiler is presented in Figure 2.1. The compiler is divided into three distinct parts based on the level of intermediate representation (IR) used. The highest level IR, *Pcode*, is a parallel C code representation with loop constructs intact. In Pcode, memory dependence analysis [23],[24], loop-level transformations [25], and memory system optimizations [26],[27] are performed. The middle level IR is referred to as *Hcode*, which is a flattened C representation with simple if-then-else and go-to control flow constructs. In Hcode, statement level profiling is performed. Additionally, profile-guided code layout and function inline expansion are performed at this level [28],[29],[30].

The final level of IR in the IMPACT compiler is referred to as *Lcode*, which is a generalized register transfer language similar in structure to most load/store processor assembly instruction sets. Lcode is logically subdivided into two subcomponents, the machine independent IR, Lcode, and the machine specific IR, *Mcode*. The data structures for both the Lcode and Mcode are identical. The difference is that Mcode is broken down such that there is a one-to-one mapping between Mcode instructions to the target machine's assembly language. Therefore, to convert Lcode to Mcode, the code generator breaks up Lcode instructions into one or more instructions which directly map to the target architecture. Lcode instructions are broken up for a variety

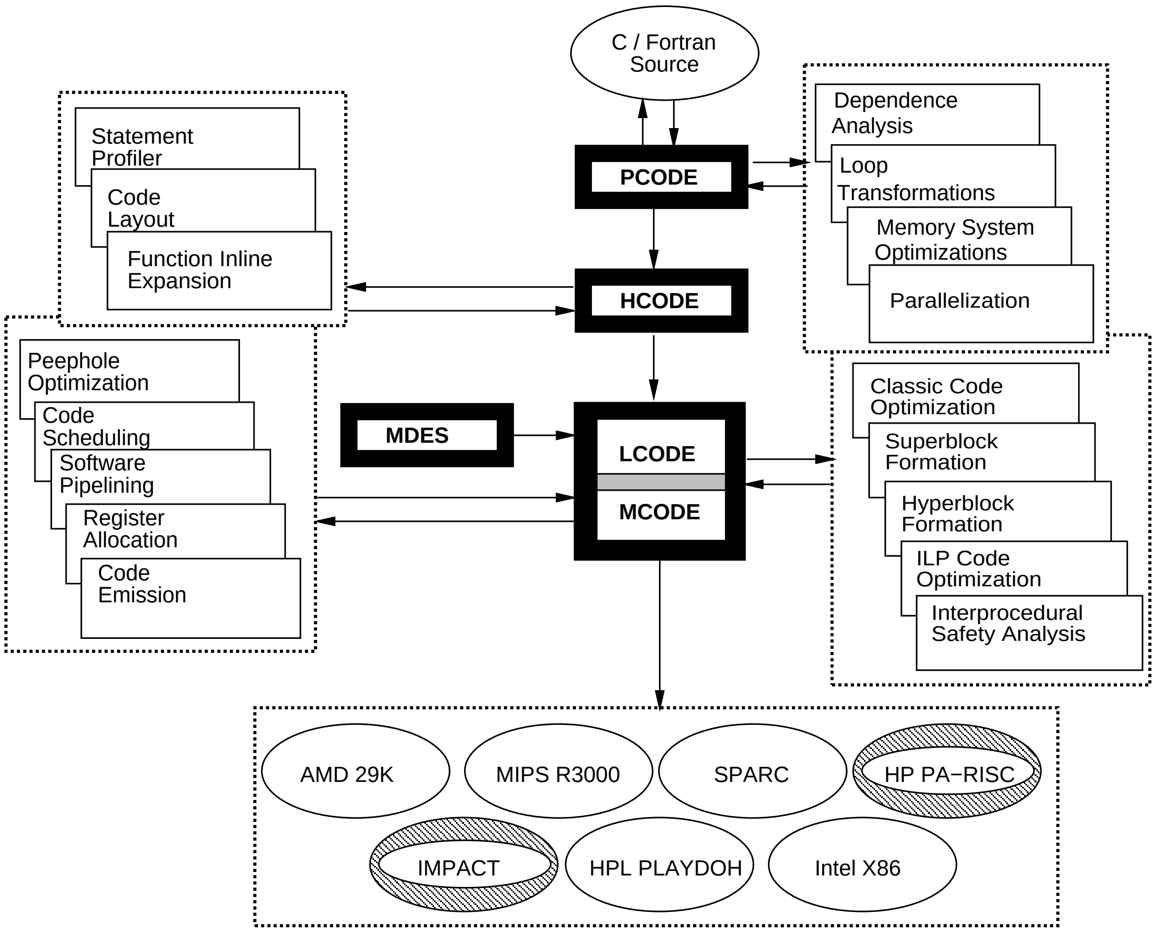


Figure 2.1 The IMPACT compiler.

of reasons including limited addressing modes, limited opcode availability, ability to specify a literal operand, and field width of literal operands.

At the Lcode level, all machine independent classic optimizations are applied [31]. 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 at the Lcode level, interprocedural safety analysis is performed [32]. This includes identifying safe instructions for speculation and function calls that do not modify memory (side-effect free).

Superblock and hyperblock compilation techniques, which are the focus of this thesis, are all performed at the Lcode level. Superblock support includes superblock formation using execution profile information, superblock classical optimization, and superblock ILP optimization. When predicated execution support is available in the target architecture, hyperblocks 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 is available.

All code generation in the IMPACT compiler is performed at the Lcode level. The two largest components of code generation are the instruction scheduler and register allocator. Scheduling is performed via either acyclic global scheduling [32],[33] or software pipelining using modulo scheduling [34]. For the acyclic global scheduling, code scheduling is applied both before register allocation (prepass scheduling) and after register allocation (postpass scheduling) to generate an efficient schedule. For software pipelining, loops targeted for pipelining are identified at the Pcode level and marked for pipelining. These loops are then scheduled using software pipelining, and the remaining code is scheduled using the acyclic global scheduler. In addition to control speculation, both scheduling techniques are capable of exploiting architectural support for data speculation to achieve more aggressive schedules [24],[35],[36].

Graph coloring based register allocation is utilized for all target architectures [37]. The register allocator employs execution profile information if it is available to make more intelligent decisions. For each target architecture, a set of specially tailored peephole optimizations are performed. These peephole optimizations are designed to remove inefficiencies during Lcode to Mcode conversion, to take advantage of specialized opcodes available in the architecture, and to remove inefficient code inserted by the register allocator.

A detailed machine description database, *Mdes*, for the target architecture is also available to all Lcode compilation modules [38]. The Mdes contains a large set of information to assist with optimization, scheduling, register allocation, and code generation. Information such as the number and type of available function units, size and width of register files, instruction latencies, instruction input/output constraints, addressing modes, and pipeline constraints is provided by the Mdes. The Mdes is queried by the optimization phases to make intelligent decisions regarding the applicability of transformations. The scheduler and register allocator rely more heavily on the Mdes to generate efficient as well as correct code.

Seven architectures are actively supported by the IMPACT compiler. These include the AMD 29K [39], MIPS R3000 [40], SPARC [41], HP PA-RISC, and Intel X86. The other two supported architectures, IMPACT and HPL PlayDoh [42], are experimental ILP architectures. These architectures provide an experimental framework for compiler and architecture research.

The IMPACT architecture is a parameterized superscalar processor with an extended version of the HP PA-RISC instruction set. Varying levels of support for speculative execution and predicated execution are available in the IMPACT architecture. For this thesis, all experiments utilize the IMPACT architecture with varying parameters.

CHAPTER 3

SPECULATIVE EXECUTION USING SUPERBLOCKS

For nonnumeric programs, the ILP available within basic blocks is extremely limited [1],[2],[43]. A VLIW or superscalar processor must optimize and schedule instructions across basic block boundaries to achieve higher performance. An effective structure for ILP compilation is the *superblock* [44]. The formation and optimization of superblocks increase 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 extract the available ILP and map it to the processor resources.

The major technique employed to achieve compact superblock schedules is speculative execution, which refers to executing an instruction before knowing that its execution is required. Such an instruction will be referred to as a *speculative instruction*. In the general sense, speculative execution may be engineered at run time or at compile time. Run time speculation is utilized by processors employing dynamic scheduling [45],[46],[47]. Conversely, superblock techniques utilize compile-time engineered speculative execution, or *speculative code motion*.

A compiler may utilize speculative code motion to achieve higher performance in three major ways. First, in regions of the program where insufficient ILP exists to fully utilize the processor resources, useful instructions may be executed. Second, instructions starting long dependence chains may be executed early to reduce the length of critical paths. Finally, long latency instructions may be initiated early to overlap their execution with useful computation. Speculative execution is generally employed by all aggressive scheduling techniques. For example, Tirumalai et al. showed that modulo scheduling of "while" loops depends on speculative support to achieve high performance [48]. Without speculative support, very little execution overlap between loop iterations is achieved.

In this chapter, the major components of the superblock ILP compilation framework are presented. These components include superblock formation, superblock classic optimization, superblock ILP optimization, and superblock scheduling.

3.1 Superblock Formation

The purpose of code optimization and scheduling is to minimize the execution time while preserving the program semantics. When this is done globally, some optimization and scheduling decisions may decrease the execution time for one control path while increasing the time for another path. By making these decisions in favor of the more frequently executed path, an overall performance improvement can be achieved.

Trace scheduling is a technique that was developed to allow scheduling decisions to be made in this manner [10],[49]. In trace scheduling the function is divided into a set of traces that represent the frequently executed paths. There may be conditional branches from the middle of the trace (side exits) and transitions from other traces into the middle of the trace (side entrances). Instructions are scheduled within each trace ignoring these control-flow transitions. After scheduling, bookkeeping is required to ensure the correct execution of off-trace code.

Code motion past side exits can be handled in a fairly straightforward manner. If an instruction J is moved from above to below a side exit, and the destination of J is used before it is redefined when the side exit is taken, then a copy of J must also be placed between the

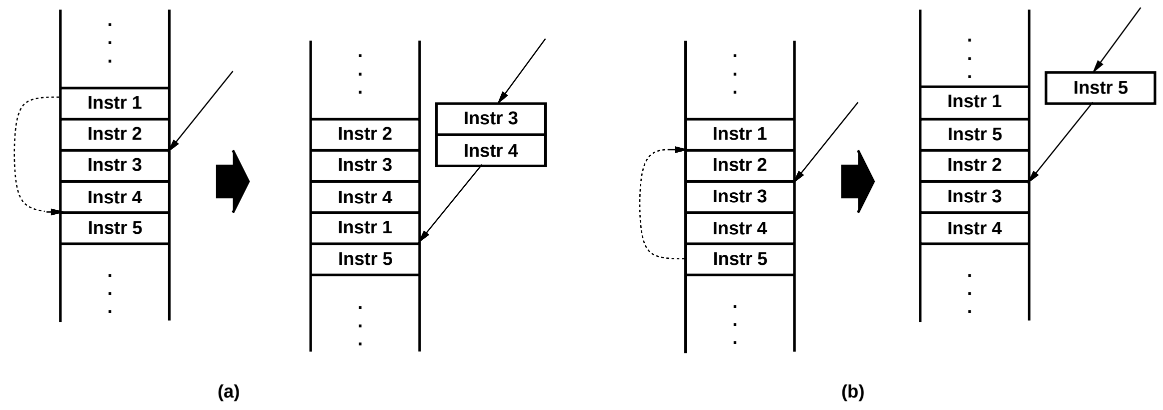


Figure 3.1 Instruction scheduling across trace side entrances, (a) moving an instruction below a side entrance, (b) moving an instruction above a side entrance.

side exit and its target. Movement of an instruction from below to above a branch can also be handled without too much difficulty. The method for doing this is described in Section 3.4.

More complex bookkeeping must be done when code is moved above and below side entrances. Figure 3.1 illustrates this bookkeeping. In Figure 3.1(a), when *Instr 1* is moved below the side entrance (to after *Instr 4*), the side entrance is moved below *Instr 1*. *Instr 3* and *Instr 4* are then copied to the side entrance. Likewise, in Figure 3.1(b), when *Instr 5* is moved above the side entrance, it must also be copied to the side entrance.

Side entrances can also make it more complex to apply optimizations to traces. For example, Figure 3.2 shows how copy propagation can be applied to the trace and the necessary bookkeeping for the off-trace code. In this example, in order to propagate the value of *r1* from *I1* to *I3*, bookkeeping must be performed. Before propagating the value of *r1*, the side entrance is moved to below *I3* and instructions *I2* and *I3* are copied to the side entrance.

The bookkeeping associated with side entrances can be avoided if the side entrances are removed from the trace. A superblock is a trace that has no side entrances. Control may only enter from the top but may leave at one or more exit points. Superblocks are formed in two

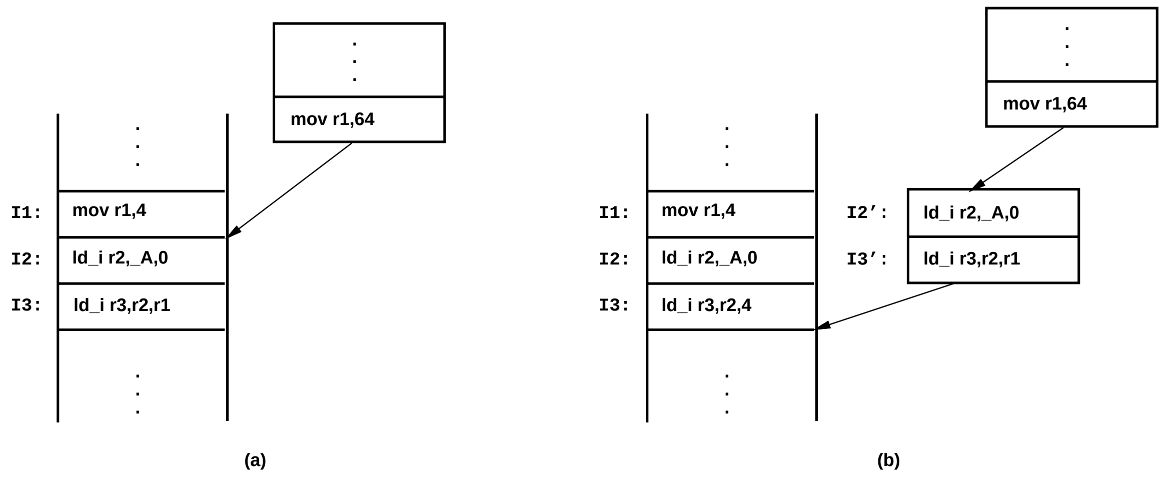


Figure 3.2 Applying copy propagation to an instruction trace, (a) before copy propagation, (b) after copy propagation with bookkeeping code inserted.

steps. First, traces are identified using execution profile information [50]. Second, a process called tail duplication is performed to eliminate any side entrances to the trace [51]. A copy is made of the tail portion of the trace from the first side entrance to the end. All side entrances into the trace are then moved to the corresponding duplicate basic blocks. The basic blocks in a superblock need not be consecutive in the code. However, the implementation restructures the code so that all blocks in a superblock appear in consecutive order for better cache performance.

The formation of superblocks is illustrated in Figure 3.3. Figure 3.3(a) shows a weighted flow graph which represents a loop code segment. The nodes correspond to basic blocks and the arcs correspond to possible control transfers. The count of each basic block indicates the execution frequency of that basic block. The count of each control transfer indicates the frequency of invoking these control transfers. Clearly, the most frequently executed path in this example is the basic block sequence $< A, B, E, F >$. There are three traces: $\{A, B, E, F\}$, $\{C\}$, and $\{D\}$. After trace selection, each trace is converted into a superblock. In Figure 3.3(a), there are two control paths that enter the $\{A, B, E, F\}$ trace at basic block F. Therefore, the tail

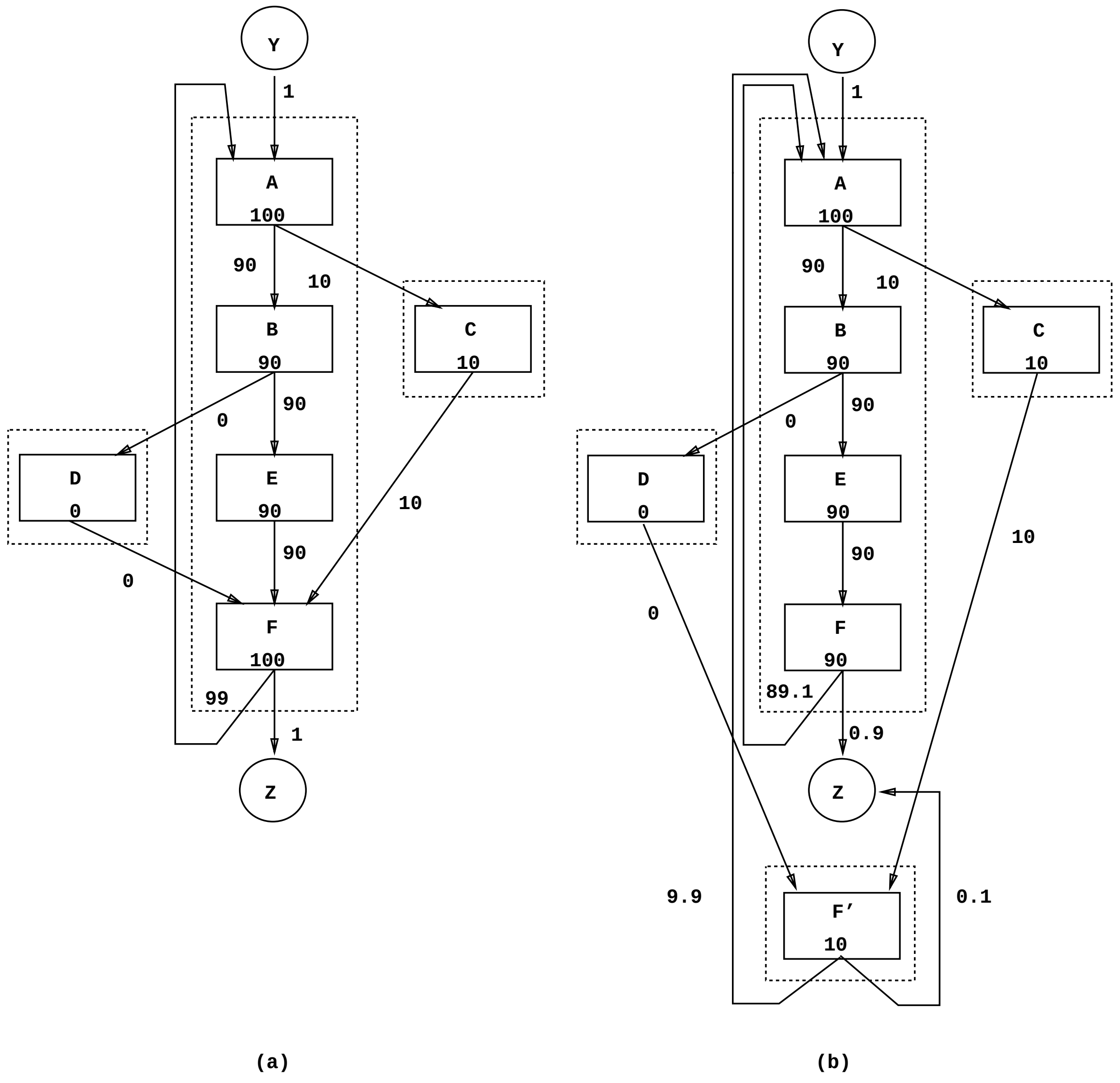


Figure 3.3 Example of the superblock formation procedure, (a) after trace selection, (b) after tail duplication.

part of the $\{A, B, E, F\}$ trace starting at basic block F is duplicated. Each duplicated basic block forms a new superblock that is appended to the end of the function. The result is shown in Figure 3.3(b). Note that there are no longer any side entrances into the most frequently traversed trace, $< A, B, E, F >$; it has become a superblock.

Superblocks are similar to the extended basic blocks. An extended basic block is defined as a sequence of basic blocks $\mathrm{B}_1 ... \mathrm{B}_k$ such that for $1 \leq i < k$, B_i is the only predecessor of B_{i+1} and B_1 does not have a unique predecessor [52]. The difference between superblocks and extended basic blocks is mainly in how they are formed. Superblock formation is guided by profile information and side entrances are removed to increase the size of the superblocks. It is possible for the first basic block in a superblock to have a unique predecessor.

3.2 Classical Optimizations Applied to Superblocks

Classical optimizations applied to basic blocks and loops may be extended to operate on superblocks and superblock loops. The premise behind this approach being successful is that additional optimization opportunities are exposed when a single path of control is focused on. In particular, superblocks identify the most likely path of control with trace selection and systematically exclude the constraints from infrequent paths with tail duplication. Additional optimization opportunities are found because constraints associated with unlikely paths of control are ignored and thus do not inhibit the optimization from occurring. This approach differs from classical interbasic block optimization techniques, which consider the constraints of all paths equally to ensure correctness.

The superblock optimization strategy has many similarities to partial redundancy elimination (PRE) and partial dead code elimination (PDE) techniques [53],[54],[55]. The major

Table 3.1 Superblock classical optimizations.

Optimization	Scope
constant propagation	superblock
forward copy propagation	superblock
backward copy propagation	superblock
memory copy propagation	superblock
common subexpression elimination	superblock
redundant load elimination	superblock
redundant store elimination	superblock
constant folding	superblock
strength reduction	superblock
constant combining	superblock
operation folding	superblock
operation cancellation	superblock
code reordering	superblock
dead code removal	superblock
loop invariant code removal	superblock loop
loop global variable migration	superblock loop
loop induction variable strength reduction	superblock loop
loop induction variable elimination	superblock loop
loop induction variable reassociation	superblock loop

difference is that new optimization algorithms are not developed, but rather by performing superblock formation, additional opportunities are exposed for traditional optimizations. Additionally, profile information is inherently used to remove additional redundancies along the the most important execution paths via superblock formation. A distinct advantage provided by PRE and PDE is their more general applicability to eliminate global partial redundancies, whereas with superblock techniques, the scope is limited to that of superblocks.

Table 3.1 shows the list of classical optimizations that have been extended to superblocks in the IMPACT compiler [31],[51]. The nonloop-based code optimizations work on a single superblock at a time. The loop-based code optimizations work on a single superblock loop at a time. A superblock loop is a superblock that has a frequently taken backedge from its last node to its first node. In the remainder of this section, a series of examples illustrating the application of superblock optimizations will be given.

3.2.1 Local optimizations extended to superblocks

Traditionally, local optimization techniques are limited to the scope of one basic block at a time. Global optimization techniques overcome this limitation by considering the entire function for identifying optimization opportunities. However, global optimizations may only be applied if there are no constraints along any possible path of execution to inhibit the transformation. Frequently, there are instances in which an optimization opportunity is inhibited by an infrequently executed path.

To illustrate the additional optimization opportunities found with superblock, consider the case of common subexpression elimination shown in Figure 3.4. The original program is shown in Figure 3.4(a). After trace selection and tail duplication, the equivalent program is shown in Figure 3.4(b). Because of tail duplication, opC cannot be reached from opB; therefore, common subexpression elimination can be applied to opA and opC. The resultant code segment after optimization is shown in Figure 3.4(c).

Another frequently applied optimization technique, dead code removal, can be extended to be more effective for superblocks. Traditional dead code removal removes operations whose value will never be used in the future. For superblocks, an operation whose value is not used in the superblock and is not live at the end of the superblock can also be considered as dead code. However, the operation is not deleted but rather copied to all control flow paths that exit in the middle of the superblock in which its value may be used. In this manner, an operation is eliminated from the superblock. To avoid confusion with traditional dead code

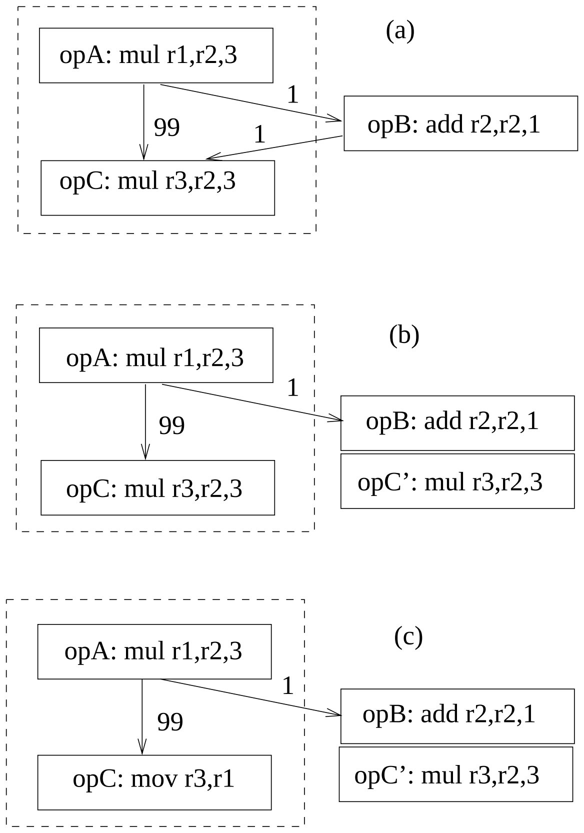


Figure 3.4 Example of superblock common subexpression elimination, (a) original program segment, (b) program segment after superblock formation, (c) program segment after common subexpression elimination.

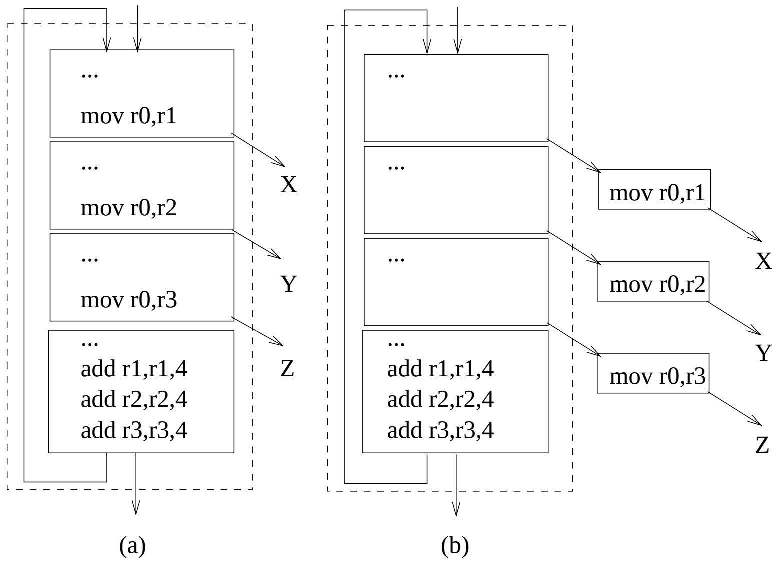


Figure 3.5 Example of superblock operation migration, (a) original program segment, (b) program segment after operation migration.

removal, this optimization technique is referred to as *operation migration*, which is effective because program control rarely exits from the middle of a superblock. An example illustrating operation migration in a superblock is given in Figure 3.5.

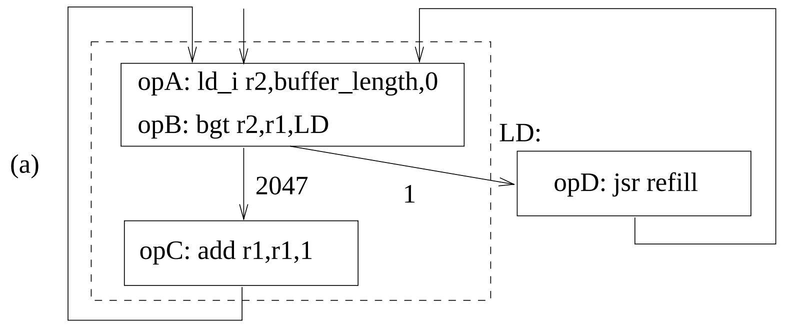
The program is a simple loop that has been unrolled three times. The loop index variable (r0) has been expanded into three registers (r1, r2, r3) using induction variable expansion, which will be discussed in Section 3.3. If the loop index variable is live outside the superblock loop, then it is necessary to update the value of r0 before each exit as shown in Figure 3.5(a). Since these values of r0 are only referenced when the superblock is exited early, these update instructions (e.g., mov r0,r1; mov r0,r2; and mov r0,r3) can be pushed out the side exits of the superblock. By performing operation migration, instructions are moved from important execution paths to less important paths to reduce the overall dynamic instruction count. The example code segment after operation migration is shown in Figure 3.5(b).

3.2.2 Loop optimizations extended to superblocks

Superblock loop optimizations can identify more optimization opportunities than traditional loop optimizations that must account for all possible execution paths within a loop. Superblock loop optimizations reduce the execution time of the most likely path of execution through a loop. In traditional loop optimizations, a potential optimization may be inhibited by a rare event such as an error in an input file, or a function call to refill a large character buffer in text processing programs. In superblock loop optimizations, function calls that are not in the superblock loop do not affect the optimization of the superblock loop.

The increased optimization opportunities created by limiting the search space to within a superblock (versus the entire loop body) for loop invariant code removal is illustrated by the example in Figure 3.6. In Figure 3.6(a), opA is not loop invariant (in the traditional sense) because its source operand is a memory variable, and opD is a function call that may modify any memory variable (assuming that the compiler does not perform interprocedural memory disambiguation). On the other hand, opA is invariant in the superblock loop. The result of superblock loop invariant code removal is shown in Figure 3.6(b).

Another effective loop optimization applied to superblocks is global variable migration, which moves frequently accessed memory variables, such as globally declared scalar variables, array elements, or structure elements, into registers for the duration of the loop. Loads and stores to these variables are replaced by register accesses. Figure 3.7 shows an example of superblock loop global variable migration. The memory variable x[r0] cannot be migrated to a register in traditional global variable migration, because r0 is not loop invariant in the entire loop. On the other hand, r0 is loop invariant in the superblock loop, and x[r0] can be migrated to a register by superblock global variable migration. The result is shown in Figure 3.7(b). The



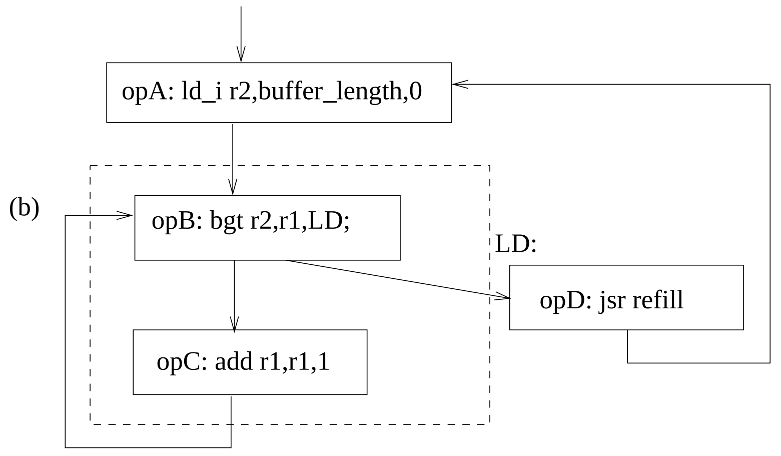
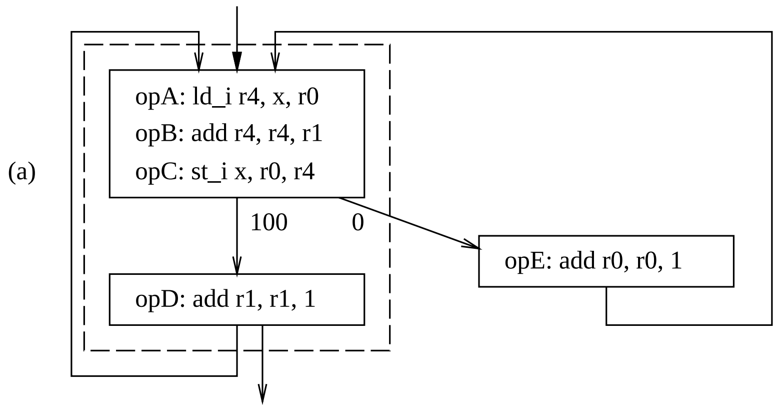


Figure 3.6 Example of superblock loop invariant code removal, (a) original program segment, (b) program segment after loop invariant code removal.



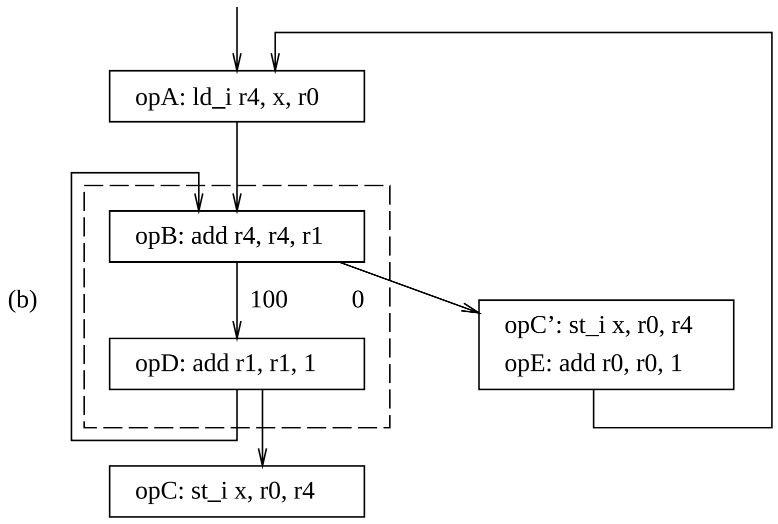


Figure 3.7 An example of superblock loop global variable migration, (a) original program segment, (b) program segment after loop global variable migration.

load of the global variable, opA, is placed at the entry point of the superblock loop. At each exit point of the superblock loop, a copy of the store of the global variable, opC, is inserted.

3.3 Superblock ILP Optimization

Superblock formation along with classical optimization techniques are not sufficient to expose sufficient ILP to effectively utilize VLIW and superscalar processors. In particular, conventional compiler optimization techniques are designed for scalar processors [52]. The primary objective of a traditional optimizer is to reduce the number and complexity of the instructions executed by the processor. Due to their limited amount of execution resources, scalar processors do not significantly benefit from increased ILP. Superscalar and VLIW processors, on the other hand, achieve higher performance by exploiting ILP with multiple data paths and function units. In most cases, superscalar and VLIW processors can reduce the execution time of an application even when the number of instructions executed is moderately increased as long as the dependence height is reduced.

Many researchers and product developers have addressed compile-time transformations to remove dependences between instructions. Kuck et al. discussed transformations, such as scalar expansion and variable renaming, to eliminate anti- and output dependences [56]. Nakatani and Ebcioglu presented an operation combining method that removes flow dependences between pairs of instructions [57]. Anantha and Long described a parallelizing compiler that employs loop unrolling, loop peeling, and variable renaming to assist Aiken and Nicolau's percolation scheduling [58],[59]. Many techniques to eliminate dependences between iterations were implemented in the Bulldog, Multiflow Trace, Cydra 5, and the IBM experimental VLIW compilers [49],[60],[61],[62]. Several height reduction techniques that reduce the length of dependence chains in arithmetic calculations have been proposed [63],[64]. The transformations discussed in this section utilize the concepts presented in these previous studies. The major contribution of the work presented in this section is the application of ILP optimization techniques to control-intensive programs in the context of superblocks.

There are two major categories of ILP optimizations, superblock enlarging optimizations and superblock dependence removing optimizations. Each is presented in the remainder of this section.

3.3.1 Superblock enlarging optimizations

Superblock enlarging optimizations increase the size of the most frequently executed superblocks so that the superblock scheduler can manipulate a larger number of instructions. It is more likely the scheduler will find independent instructions to schedule at every cycle in a superblock when there are more instructions to choose from. The superblock enlarging optimizations utilized in the IMPACT compiler consist of branch target expansion, loop unrolling, and loop peeling.

Branch target expansion. Branch target expansion expands the target superblock of a likely taken control transfer that ends a superblock. The target superblock is copied and appended to the end of the original superblock. Note that branch target expansion is not applied to control transfers that are loop backedges. Branch target expansion is repeatedly applied to increase the size of a superblock until one of the following conditions exists: a predefined maximum superblock size is reached, the branch ending the superblock does not favor one direction by a large enough amount, or the ratio of the execution frequency of the last branch to the first instruction of the superblock is below a threshold.

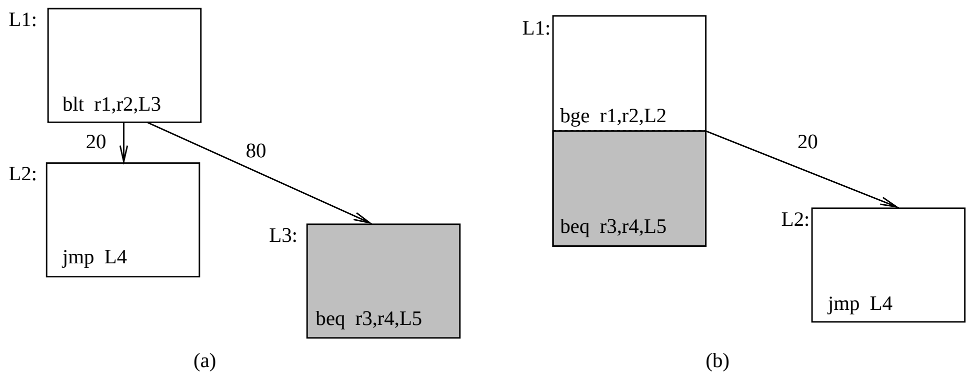


Figure 3.8 Example of branch target expansion, (a) original program segment, (b) program segment after expansion.

An example to illustrate branch target expansion is presented in Figure 3.8. The shaded superblock (SB3) is frequently entered from SB1. To enable optimization and scheduling among the instructions in both superblocks, a copy of SB3 is appended to SB1. Since SB3 was originally along the taken path of the branch which ended SB1, its condition and direction are reversed to maintain correctness. If SB3 could only be entered from SB1, the original version of SB3 is unnecessary and is later removed.

Loop unrolling. Loop unrolling is a technique commonly used to overlap the execution of multiple iterations of a loop. A loop unrolled N times has $N - 1$ copies of the loop body appended to the original loop. The control transfers to the beginning of the loop are adjusted to account for the unrolling. If the iteration count is known on loop entry, it is possible to remove many of these control transfers by using a preconditioning loop to execute the first modulo N iterations. All of the loop examples used in this section are assumed of this type for simplicity. However, in practice for C programs, this is often not the case. After loop unrolling, the loop body contains N iterations of the loop, which can be scheduled as a single unit by the compiler. An example of loop unrolling is presented later in this section in conjunction with the discussion of register renaming.

Loop peeling. Loop peeling is a special purpose optimization applied to superblock loops which do not iterate frequently. Loop peeling refers to stripping off the first several iterations of the loop. A separate copy of the loop body is made for each peeled iteration. The original loop body is moved to the end function to handle executions of the loop that require more than the number of peeled iterations. Peeled iterations are essentially straight-line code and can be merged with preceding and succeeding superblocks to create a single large superblock. The transformation is most effective for a nested loop structure in which the outer loop iterates frequently and the inner loop iterates infrequently. For this situation, loop peeling converts the inner loop to straight-line code and merges the peeled iterations with the body of the outer loop. The outer loop is then a superblock loop and can be optimized and scheduled with techniques applied to inner loops (i.e., loop unrolling and register renaming).

Superblock loop peeling is only effective when the peeled loop iterates the exact number of times it was peeled. In cases in which the loop iterates more or less frequently, the peeled code must be exited to maintain correctness. Therefore, the applicability of superblock loop peeling is limited to loops with highly predictable iteration counts. However, as will be discussed in Chapter 7, predicated execution allows much more widespread use of loop peeling to increase ILP. With predicated execution, the loop iteration count does not have to be determined precisely. Rather, execution is maintained within the peeled loop body for any iteration count less than or equal to the number of times the loop is peeled. As a result, loop peeling is useful to a large class of loops which iterate infrequently with predicated execution.

3.3.2 Superblock dependence-removing optimizations

The second category of superblock ILP optimizations is dependence-removing optimizations. These optimizations eliminate data dependences between instructions within frequently executed superblocks, which increases the ILP available to the code scheduler. The transformations consist of register renaming, accumulator variable expansion, induction variable expansion, search variable expansion, operation combining, strength reduction, and tree height reduction. The first four transformations are most effective for exposing increasing levels of ILP in unrolled loops. The remaining three increase the utilization of a superscalar or VLIW processor's arithmetic hardware. A side effect of most of these transformations is to substantially increase the number of processor registers utilized by the program.

Register renaming. Reuse of registers by the compiler and variables by the programmer introduces artificial anti- and output dependences and restricts the effectiveness of a superscalar or VLIW processor. Many of these artificial dependences can be eliminated with register renaming, which assigns unique registers to different definitions of the same register [56]. A common use of register renaming is to rename registers within individual loop bodies of an unrolled loop. An example to illustrate this is shown in Figure 3.9. The instruction latencies shown in Table 3.2 are used for all examples in this section.

A simple inner loop (Figure 3.9(a)) and its corresponding assembly code (Figure 3.9(b)) are shown. Without either unrolling or renaming, each loop iteration requires seven cycles to execute. If the loop is unrolled three times (Figure 3.9(c)), each loop iteration requires an average of 6.3 cycles. Finally, if register renaming is applied in addition to loop unrolling (Figure 3.9(d)), the average execution time of each loop iteration is reduced to 2.7 cycles.

(a) Original Loop

```
for (j=0; j<n; j++) {
   C(j) = A(j)+B(j)
}
```

(b) Assembly Code

```
     Assembly          IT
L1:  ld_f f2,A,r1      0
     ld_f f3,B,r1      0
     add_f f4,f2,f3    2
     st_f C,r1,f4      5
     add r1,r1,4       5
     blt r1,r5,L1      6

7 cycles / 1 iteration
```

(c) After Loop Unrolling

```
     Assembly          IT
L1:  ld_f f2,A,r1      0
     ld_f f3,B,r1      0
     add_f f4,f2,f3    2
     st_f C,r1,f4      5
     add r1,r1,4       5
     ld_f f2,A,r1      6
     ld_f f3,B,r1      6
     add_f f4,f2,f3    8
     st_f C,r1,f4     11
     add r1,r1,4      11
     ld_f f2,A,r1     12
     ld_f f3,B,r1     12
     add_f f4,f2,f3   14
     st_f C,r1,f4     17
     add r1,r1,4      17
     blt r1,r5,L1     18

19 cycles / 3 iterations
```

(d) After Register Renaming

```
     Assembly             IT
L1:  ld_f f21,A,r11        0
     ld_f f31,B,r11        0
     add_f f41,f21,f31     2
     st_f C,r11,f41        5
     add r12,r11,4         0
     ld_f f22,A,r12        1
     ld_f f32,B,r12        1
     add_f f42,f22,f32     3
     st_f C,r12,f42        6
     add r13,r12,4         1
     ld_f f23,A,r13        2
     ld_f f33,B,r13        2
     add_f f43,f23,f33     4
     st_f C,r13,f43        7
     add r11,r13,4         2
     blt r11,r5,L1         7

8 cycles / 3 iterations
```

Figure 3.9 Example of loop unrolling and register renaming. All examples in this section assume a superscalar processor with infinite resources and no register renaming hardware. Also, the issue times (IT) shown are for the code after code scheduling, but to preserve clarity, the unscheduled code is shown. Sorting by issue time yields the scheduled code.

Table 3.2 Instruction latencies for all examples presented in this section.

Function	Latency	Function	Latency
Int ALU	1	FP ALU	3
Int multiply	3	FP conversion	3
Int divide	10	FP multiply	3
branch	1	FP divide	10
memory load	2	memory store	1

Accumulator variable expansion. An accumulator variable accumulates a sum or product in each iteration of a loop. A common example is the computation of a dot product between two vectors. For loops of this type, the accumulation operation often defines the critical path within the loop. Accumulator variable expansion eliminates redefinitions of an accumulator variable within an unrolled loop by creating k temporary accumulators (k refers to the number of accumulation instructions for that accumulation register in the unrolled loop body). Each temporary accumulator replaces one definition of the original accumulator in the loop. In this manner all flow, anti-, and output dependences are eliminated between the accumulation instructions. To recover the value of the original accumulator variable, the temporary accumulators are summed at all exit points of the loop.

An algorithm to perform accumulator variable expansion for a loop is shown in Figure 3.10. Note that accumulator variables may only be modified in the loop by increment or decrement instructions. Also, since the total value of the accumulator is not computed until the loop is exited, no instructions in the loop other than accumulation instructions may use the accumulator variable.

An example to illustrate the application of accumulator variable expansion is presented in Figure 3.11. This loop (Figure 3.11(a)) is the innermost loop for matrix multiplication. After conventional compiler optimization (Figure 3.11(b)), each iteration of the loop requires eight cycles to execute. Loop unrolling and register renaming (Figure 3.11(c)) further improve the average execution time to 4.7 cycles per iteration. However, it is clear that the accumulation of the sum into $r1$ limits the ILP in the loop. Accumulator variable expansion (Figure 3.11(d)), removes the dependences between instructions which increment/decrement $r1$ by introducing three temporary accumulators, $r11$, $r12$, and $r13$. With this transformation, an average of 3.3

```
accumulator_variable_expansion(loop)
{
  for each variable V in loop:
    Determine if V is an accumulator variable by checking if the following
    conditions are satisfied:
      1. All instructions modifying V are increment/decrement instructions
      2. V is only referenced in the above increment/decrement instructions
      3. The loop contains more than one of the above increment/decrement instructions
    If V is an accumulator variable, do this transformation to make the
    increment/decrement instructions independent:
      1. Let k be the number of increment/decrement instructions
      2. Allocate k new virtual registers (temporary accumulators)
      3. Insert initialization code for the above k registers into the loop
         preheader as follows:
         a. Initialize the first register to V's value
         b. Initialize the other k-1 registers to zero
      4. For each increment/decrement instruction, replace V by one of the above
         k temporary accumulators using each accumulator exactly once
      5. At all loop exit points, insert a summation of the k temporary
         accumulators to generate V's exit value
}
```

Figure 3.10 Algorithm for accumulator variable expansion.

cycles is required for each iteration. Also, the application of the induction variable expansion, described next, would improve this time to 2.7 cycles per iteration.

Induction variable expansion. Induction variables are used within loops to index through loop iterations and through regular data structures such as arrays. The value of an induction variable is used to compute the address of data structures and, therefore, must be computed before the data access is performed. When data access is delayed due to dependences on the induction variable computation, ILP is typically limited. Induction variable expansion eliminates flow, anti-, and output dependences between definitions of induction variables and their uses within an unrolled loop body by creating k temporary induction variables (k is the number of instructions which update this induction variable in the unrolled loop body). Each temporary induction variable replaces one definition of the original induction variable in the

(a) Original Loop

```
for (k=0; k<n; k++) {
  C(i,j) = C(i,j) +
           A(i,k) * B(k,j)
}
```

(b) Assembly Code

	Assembly	IT
	ld_f f1,C,r2	--
L1:	ld_f f3,A,r4	0
	ld_f f5,B,r6	0
	mul_f f7,f3,f5	2
	add_f f1,f1,f7	5
	add r4,r4,4	0
	add r6,r6,r8	0
	blt r4,r9,L1	5
	st_f C,r2,f1	--

8 cycles / 1 iteration

(c) After Unrolling and Renaming

	Assembly	IT
	ld_f f1,C,r2	--
	ld_f f31,A,r41	0
	ld_f f51,B,r61	0
L1:	mul_f f71,f31,f51	2
	add_f f1,f1,f71	5
	add r42,r41,4	0
	add r62,r61,r8	0
	ld_f f32,A,r42	1
	ld_f f52,B,r62	1
	mul_f f72,f32,f52	3
	add_f f1,f1,f72	8
	add r43,r42,4	1
	add r63,r62,r8	1
	ld_f f33,A,r43	2
	ld_f f53,B,r63	2
	mul_f f73,f33,f53	4
	add_f f1,f1,f73	11
	add r61,r63,r8	2
	add r41,r43,4	2
	blt r41,r9,L1	11
	st_f C,r2,f1	--

14 cycles / 3 iterations

(d) After Accumulator Expansion

	Assembly	IT
	ld_f f11,C,r2	--
	mov_f f12,0	--
	mov_f f13,0	--
L1:	ld_f f31,A,r41	0
	ld_f f51,B,r61	0
	mul_f f71,f31,f51	2
	add_f f11,f11,f71	5
	add r42,r41,4	0
	add r62,r61,r8	0
	ld_f f32,A,r42	1
	ld_f f52,B,r62	1
	mul_f f72,f32,f52	3
	add_f f12,f12,f72	6
	add r43,r42,4	1
	add r63,r62,r8	1
	ld_f f33,A,r43	2
	ld_f f53,B,r63	2
	mul_f f73,f33,f53	4
	add_f f13,f13,f73	7
	add r61,r63,r8	3
	add r41,r43,4	3
	blt r41,r9,L1	7
	add_f f11,f11,f12	--
	add_f f11,f11,f13	--
	st_f C,r2,f11	--

10 cycles / 3 iterations

Figure 3.11 Example of accumulator variable expansion.

```
induction_variable_expansion(loop)
{
  for each variable V in loop:
    Determine if V is an induction variable by checking if the following
    conditions are satisfied:
      1. All instructions modifying V are increment/decrement instructions
      2. The increment/decrement value is the same for all of the above
         increment/decrement instructions and it is invariant in the loop
      3. The loop contains more than one of the above increment/decrement instructions
    If V is an induction variable, do this transformation to make the
    increment/decrement instructions independent:
      1. Let k be the number of increment/decrement instructions
      2. Let m be the loop invariant increment/decrement value
      3. Allocate k+2 new virtual registers (k+1 temporary induction
         registers and one modified increment/decrement value z)
      4. Let the k+1 temp induction registers be numbered p = 0 to k
      5. Insert initialization code for the above registers into the loop
         preheader as follows:
         a. Initialize register p with V's value + p * m
         b. Initialize register z with k * m
      6. Replace all references to V before the first increment/decrement
         instruction by references to temp induction register 0
      7. Replace all references to V between the pth increment/decrement
         instruction and the (p+1)th increment/decrement instruction by
         references to temp induction register p
      8. Remove all the increment/decrement instructions from the loop
      9. Before each branch back to the start of the loop, increment the
         k+1 registers by register z's value
}
```

Figure 3.12 Algorithm for induction variable expansion.

loop. Also, the increments of each temporary induction variable are moved to the end of the unrolled loop body to eliminate the flow dependences between each definition of a temporary induction variable and its uses. Each temporary induction variable is initialized to its correct value in the loop preheader.

An algorithm to perform induction variable expansion for a loop is presented in Figure 3.12. Note that there are two major distinctions between induction variables and accumulator variables. First, the value of an accumulator variable may only be used by accumulation instructions in the loop, whereas induction variables are used by at least one other instruction in the loop. Second, the increment of an accumulator variable may vary with each iteration of the loop; however, induction variables must be incremented by a loop invariant amount.

An example loop to illustrate the application of induction variable expansion is shown in Figure 3.13(a). After conventional compiler optimization (Figure 3.13(b)), each loop iteration requires six cycles to execute. With loop unrolling and register renaming (Figure 3.13(c)), this is reduced to 2.7 cycles per iteration. However, the increments of $r2$ changed by register renaming are still flow dependent. Induction variable expansion (Figure 3.13(d)) changes the increments of the three registers renaming created for $r2$ so that the definitions are independent. This further reduces the number of cycles to two per iteration. The improvement becomes more pronounced the more the loop is unrolled. For example, the same loop unrolled eight times would require 1.6 cycles per iteration after renaming but only 0.8 cycles per iteration after induction variable expansion.

Search variable expansion. A single value, such as a maximum or minimum, is often determined for matrices or arrays. Such values will be referred to as search values. Within an unrolled loop body, the chain of flow dependences between successive tests and updates of a search variable often defines a critical path. Similar to accumulator variable expansion and induction variable expansion, search variable expansion eliminates this chain of dependences by creating k temporary search variables. Each temporary search variable replaces one definition of the original search variable in the loop. In this manner, each loop body in the unrolled loop determines a value for the search variable during execution. When the loop is exited, the value of the original search variable is obtained by comparing the values of all temporary search variables.

(a) Original Loop

```
for (i=0; i<n; i++) {
   C(j) = A(j)*B(j)
   j = j + K
}
```

(b) Assembly Code

	Assembly	IT
L1:	ld_f f3,A,r2	0
	ld_f f4,B,r2	0
	mul_f f5,f3,f4	2
	st_f C,r2,f5	5
	add r2,r2,r7	5
	add r1,r1,1	0
	blt r1,r6,L1	5

6 cycles / 1 iteration

(c) After Unrolling and Renaming

	Assembly	IT
L1:	ld_f f31,A,r21	0
	ld_f f41,B,r21	0
	mul_f f51,f31,f41	2
	st_f C,r21,f51	5
	add r22,r21,r7	0
	ld_f f32,A,r22	1
	ld_f f42,B,r22	1
	mul_f f52,f32,f42	3
	st_f C,r22,f52	6
	add r23,r22,r7	1
	ld_f f33,A,r23	2
	ld_f f43,B,r23	2
	mul_f f53,f33,f43	4
	st_f C,r23,f53	7
	add r21,r23,r7	2
	add r1,r1,3	0
	blt r1,r6,L1	7

8 cycles/ 3 iterations

(d) After Induction Expansion

	Assembly	IT
	mov r21,r2	--
	add r22,r21,r7	--
	add r23,r22,r7	--
	mul r71,r7,3	--
L1:	ld_f f31,A,r21	0
	ld_f f41,B,r21	0
	mul_f f51,f31,f41	2
	st_f C,r21,f51	5
	ld_f f32,A,r22	0
	ld_f f42,B,r22	0
	mul_f f52,f32,f42	2
	st_f C,r22,f52	5
	ld_f f33,A,r23	0
	ld_f f43,B,r23	0
	mul_f f53,f33,f43	2
	st_f C,r23,f53	5
	add r21,r21,r71	5
	add r22,r22,r71	5
	add r23,r23,r71	5
	add r1,r1,3	0
	blt r1,r6,L1	5

6 cycles / 3 iterations

Figure 3.13 Example of induction variable expansion.

Operation combining. Flow dependences between pairs of instructions each with a compile-time constant source operand can be eliminated with operation combining [57]. Flow dependences that can be eliminated with operation combining often arise between address calculation instructions and memory access instructions along with loop variable increments and loop exit branches. To illustrate the application of operation combining, consider the following two flow dependent instructions:

I1: r1 = r2 op1 C1

I2: r3 = r1 op2 C2 $\Rightarrow$ r3 = r2 op2 (C1 op3 C2)

The instructions are combined in two steps. First, the nonconstant source operand of $I2$ is replaced by the nonconstant source operand of $I1$. Therefore, $r1$ is replaced by $r2$. If the destination and source registers for $I1$ are the same, the positions of $I1$ and $I2$ are exchanged rather than switching source operands. Second, the constant source operands are evaluated according to the $op1$ and $op2$ operations. For this case, if both $op1$ and $op2$ are add operations, $C2$ is replaced by $C1 + C2$.

Clearly, the combination of operations is limited to those of the same precedence and data type (e.g., an add and a multiply operation cannot be combined). The current implementation allows the following operations on the left to be combined with the operations on the right (f indicates floating point, otherwise integer is assumed):[1]

(add, subtract) $\Rightarrow$ (add, subtract, compare, memory load, memory store, conditional branch)

[1]Note that if the evaluation of the constants during operation combining results in an overflow or underflow, the compiler does not perform the transformation.

(a) Original Loop

```
while (t < 10.0) {
   i = i + 1
   t = A(i+2) - 3.2
}
```

(b) Assembly Code

```
          Assembly         IT
L1:  add r1,r1,4           0
     ld_f f2,r1,8          1
     sub_f f3,f2,3.2       3
     blt f3,10.0,L1        6
     7 cycles / 1 iteration
```

(c) After Combining

```
          Assembly         IT
L1:  ld_f f2,r1,12         0
     add r1,r1,4           0
     sub_f f3,f2,3.2       2
     blt f2,13.2,L1        2
     5 cycles / 1 iteration
```

Figure 3.14 Example of operation combining.

(multiply)	$\Rightarrow$	(multiply)
(add_f, subtract_f)	$\Rightarrow$	(add_f, subtract_f, compare_f, conditional branch_f)
(multiply_f, divide_f)	$\Rightarrow$	(multiply_f, divide_f)

An example code segment to illustrate the effectiveness of operation combining is presented in Figure 3.14(a). Without operation combining (Figure 3.14(b)), each iteration of the loop requires seven cycles to execute. However, the flow dependence between the first two instructions and the last two instructions can be eliminated with operation combining. Note that the first two instructions must exchange positions when operation combining is performed. After operation combining, the execution time of each loop iteration is reduced to five cycles.

Strength reduction. Strength reduction is a common technique employed by compilers to replace long latency instructions with one or more instructions which collectively require less time. In many existing compilers, integer multiply by a compile-time constant is replaced by a sequence of left shifts and adds [65]. For example, $r2 = r1 * 10$ can be replaced by

$\text{temp1} = \text{r1} \ll 3$

$\text{temp2} = \text{r1} \ll 1$

$\text{r2} = \text{temp1} + \text{temp2}$

The applicability of this transformation depends on whether the sequence of instructions generated by strength reduction will execute in fewer cycles than the original instruction. The application of strength reduction is typically limited in a scalar processor by this restriction. However, many of the instructions generated during strength reduction are independent and can be executed concurrently on a superscalar or VLIW processor. Therefore, there are more opportunities to apply strength reduction for superscalar and VLIW processors. In addition to applying strength reduction for integer multiply, superscalar and VLIW processors may benefit from reduction of integer divide and integer remainder by a compile-time constant.

Tree height reduction. Scalar processor compilers typically generate code for arithmetic expressions by minimizing both the total number of instructions and the total number of temporary registers required. For superscalar and VLIW processors, however, these methods often limit performance by restricting parallel computation of individual components of an arithmetic expression. Tree height reduction exposes ILP in the computation of an arithmetic expression by first constructing an expression tree to represent the arithmetic expression [63],[64]. The tree is then balanced to reduce the height, which represents the number of cycles to compute the expression using a specific processor model.

The compiler used in this study uses a modified version of the algorithm proposed by Baer and Bovet [63] that examines intermediate code rather than source code. This tree height reduction algorithm utilizes communicativity and associativity to reduce the height of expressions

(a) Original Source Code

```
A = B * (C + D) * E * F / G
```

(b) Assembly Code

Assembly	IT
add_f f1,fC,fD	0
mul_f f1,f1,fB	3
mul_f f1,f1,fE	6
mul_f f1,f1,fF	9
div_f fA,f1,fG	12

22 cycles

(C) After Height Reduction

Assembly	IT
add_f f1,fC,fD	0
mul_f f2,fB,fE	0
div_f f3,fF,fG	0
mul_f f1,f1,f2	3
mul_f fA,f1,f3	10

13 cycles

Figure 3.15 Example of tree height reduction.

using -, +, *, /, (,). Note that this algorithm does not apply distributive property. It also assumes that all operations have the same latency which is not true for the processor model studied and, therefore, limits its effectiveness in this case. Currently, more advanced techniques for height reduction that utilize the distributive property and allow different latencies for operations are being examined in IMPACT.

An example arithmetic computation to illustrate the effectiveness of tree height reduction is presented in Figure 3.15(a). With conventional code optimization techniques (Figure 3.15(b)), the computation of the expression requires 22 cycles. With tree height reduction (Figure 3.15(c)), two multiply instructions and an add instruction can be executed in parallel with the divide, thereby reducing the execution time to 13 cycles.

3.4 Superblock Scheduling

Superblock scheduling extracts the ILP from the program and maps it to the processor resources. Speculative execution is the major technique employed during scheduling to achieve a high degree of instruction overlap. This section begins with a basic explanation of superblock scheduling and then goes into a detailed discussion of speculative execution in superblocks. The section is concluded with a comprehensive example of superblock scheduling.

3.4.1 Superblock scheduling algorithm

Superblock scheduling is a two-step procedure applied to each superblock independently. The first step is to build a dependence graph which represents all the data and control dependences between instructions within a superblock. There are three types of data dependences, flow, anti-, and output. Control dependences enforce the ordering between a branch instruction and other instructions before and after a branch. There is a control dependence between a branch and a subsequent instruction J if the branch must execute before instruction J. Correspondingly, there is a control dependence between an instruction J and a subsequent branch if the branch must execute after instruction J.

The second step in the superblock scheduling algorithm is to perform list scheduling using the dependence graph, instruction latencies, and resource constraints of the processor. The general idea of a list scheduling algorithm is to pick, from a set of nodes (instructions) that are *ready* to be scheduled, the best combination of nodes to issue in a cycle. The best combination of nodes is determined by using heuristics which assign priorities to the ready nodes [32]. A node is ready if all of its parents in the dependence graph have been scheduled and the result produced by each parent is available.

The efficiency of the schedule is based on two important factors. First, if the number of instructions the scheduler has to choose from is large, the scheduler is more likely to utilize all available resources to take full advantage of the VLIW or superscalar processor. The number of instructions the scheduler may examine is increased by applying superblock formation and superblock enlarging optimizations discussed in Sections 3.1 and 3.3. Second, if the number of dependences is reduced, a more compact code schedule can be found. Data dependences are removed using the ILP optimization techniques presented in Section 3.3. Control dependences are removed using a combination of renaming and hardware support for speculative execution. Enabling speculative execution in superblocks is discussed in this next subsection.

3.4.2 Speculative execution in superblocks

The instructions within a superblock are placed linearly in the instruction memory. Thus, the side exits of the superblock correspond to conditional branches where the branch is not likely taken. To efficiently schedule code within superblocks, it is essential that the compiler be able to move instructions above and below branches. Let J and BR denote two instructions where J is the instruction to move and BR is a branch instruction. The term *LIVE-OUT(BR)* is defined as the set of variables which may be used before being redefined when BR is taken (i.e., the superblock is exited). Moving J from above to below BR (downward code motion) is relatively straightforward. If BR does not depend on J, then J may be moved below BR. If the destination of J is in *LIVE-OUT(BR)*, then a copy of J must be inserted between BR and its target.

To reduce the critical path of a superblock, upward code motion is more common. For instance, moving a load instruction earlier to hide the load delay is frequently done. When an instruction is moved upward across a branch, it is executed speculatively since the result of the instruction is only needed when the branch is not taken. For upward code motion, moving J from below to above BR, there are two major restrictions:

Restriction 1: The destination of J is not in *LIVE-OUT(BR)*.

Restriction 2: J will never cause an exception that may terminate program execution when BR is taken.

Restriction 1 usually has very little effect on code scheduling after superblock ILP optimizations. Register renaming is the most common transformation to overcome this restriction. By renaming the destination register of an instruction to a new register, it is guaranteed not to be in *LIVE-OUT(BR)*.

A much more serious problem with speculative execution is the prevention of premature program termination due to exceptions caused by speculative instructions. Exceptions by speculative instructions that would not have occurred if the code was not scheduled must be prevented. This limitation is enforced by Restriction 2. The two most extreme interpretations of Restriction 2 are to fully enforce it and to completely ignore it. Fully enforcing Restriction 2 results in only allowing instructions that cannot cause exceptions to be candidates for speculative execution. This model is referred to as *restricted speculation* [66],[67]. For conventional processors, restricted speculation does not allow memory load, memory store, integer divide, and all floating-point instructions to be speculated. The performance of restricted speculation is limited because of its inability to move long latency or instructions within a long dependence chain upward in the superblock. Frequently, memory loads and floating-point instructions have greater than single cycle latency and are part of long dependence chains.

Restriction 2 may be completely ignored if there are nonexcepting or silent versions of all potentially excepting instructions in the instruction set architecture. This model is referred to as *general speculation* [66],[67]. If an exception occurs for a silent instruction, it is simply ignored and a garbage value is written into the destination of the instruction. Note that exceptions such as page faults and TLB misses are not ignored for silent instructions, but rather handled as if the instruction was not speculative. The garbage value written to the destination may be propagated to other speculative instructions which use the result. However, for programs which would have never excepted without scheduling, the invalid result is guaranteed to not affect the correctness of the program execution. This is true because if the speculative instruction indeed excepts, it must be for an execution of the superblock in which an exit branch is taken, i.e., the speculative instruction was unnecessarily executed. By enforcing Restriction 1, the scheduler guarantees that the destinations written by all speculative instructions are in the *LIVE-OUT()* sets of any branches they moved above.

For undebugged programs or programs which rely on traps during normal operation, general speculation is not a desirable scheduling paradigm. In the next chapter, the sentinel speculation model is discussed which allows the scheduling freedom provided by general speculation but allows precise detection and recovery from exceptions for speculative instructions.

3.4.3 Superblock scheduling example

To illustrate superblock scheduling and compare the effects of restricted and general speculation, an example C loop segment is brought through each step of the superblock compilation procedure in this section. The C source code for the example is presented in Figure 3.16 and the corresponding assembly code in Figure 3.17. The loop segment traverses a linked list data structure computing the sum of the absolute value of a weight field contained within each element of the list.

The first step of the superblock compilation procedure is trace selection to identify the sequences of blocks that will be converted into superblocks. Figure 3.18 shows the portion of the

```
avg = 0;
weight = 0;
count = 0;
while(ptr != NULL)  {
    count = count + 1;
    if(ptr−>wt < 0)
        weight = weight − ptr−>wt;
    else
        weight = weight + ptr−>wt;
    ptr = ptr−>next;
}
if(count != 0)
    avg = weight/count;
```

Figure 3.16 Example C source code segment.

```
(i1)           ld_i    r1,    ptr,   0
(i2)           mov     r7,    0               // avg
(i3)           mov     r2,    0               // count
(i4)           mov     r3,    0               // weight
(i5)           beq     r1,    0,     L3
(i6)    L0:    add     r2,    r2,    1
(i7)           ld_i    r4,    r1,    0        // ptr−>wt
(i8)           bge     r4,    0,     L1
(i9)           sub     r3,    r3,    r4
(i10)          jmp     L2
(i11)   L1:    add     r3,    r3,    r4
(i12)   L2:    ld_i    r1,    r1,    4
(i13)          bne     r1,    0,     L0
(i14)   L3:    beq     r2,    0,     L4
(i15)          div     r7,    r3,    r2
(i16)          st_i    avg,   0,     r7
(i17)   L4:
```

Figure 3.17 Assembly code segment.

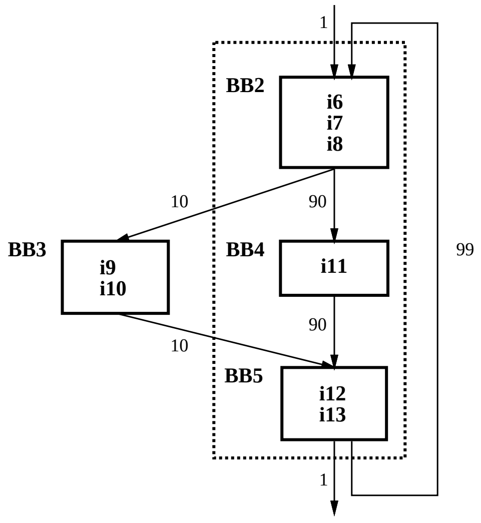


Figure 3.18 Loop portion of control flow graph after trace selection.

control flow graph corresponding to the *while* loop after trace selection. The dashed box outlines the most frequently executed path of the loop. To convert the trace into a superblock, tail duplication is applied to eliminate the side entrance from BB3 to BB5. The loop portion of the program segment after the completion of superblock formation is shown in Figure 3.19. During tail duplication, BB5 is copied to form superblock 2 (SB2). Also, since BB3 only branches to BB5, the target of the jump instruction, i10, can be appended (refer to the discussion of branch target expansion in Section 3.3) to the end of BB3. Therefore, BB3 and BB5 are merged to form a second superblock, SB2.

Classical and ILP optimizations are then applied to each of the superblocks to increase the number of instructions visible to the scheduler and to reduce the number of dependences among the instructions in the superblock. Renaming destinations that are in the *LIVE-OUT()* set of one or more previous branch instructions overcomes Restriction 1 for upward code motion. For this example, the superblock loop is unrolled two times and register renaming is applied to the load instructions to allow overlap of the unrolled loop iterations. Note that additional optimiza-

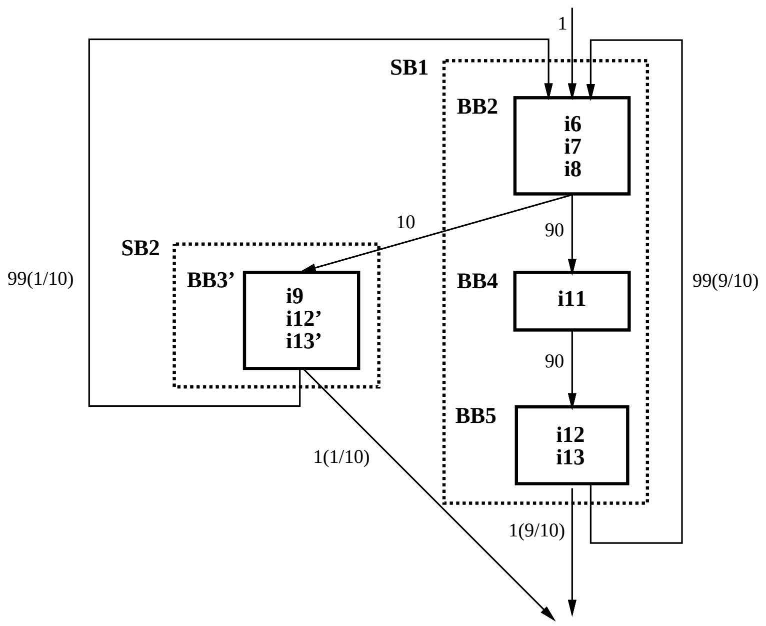


Figure 3.19 Loop portion of control flow graph after superblock formation and branch target expansion.

tion opportunities exist for accumulator expansion (r2 and r3 may be expanded). However, for ease of following the example, expansion optimizations are not applied. The loop segment after superblock optimizations is presented in Figure 3.20. Note that once the loop is unrolled and renamed, branch I9 must branch to L1$'$ to restore r1 and r4 before the code at L1 is executed.

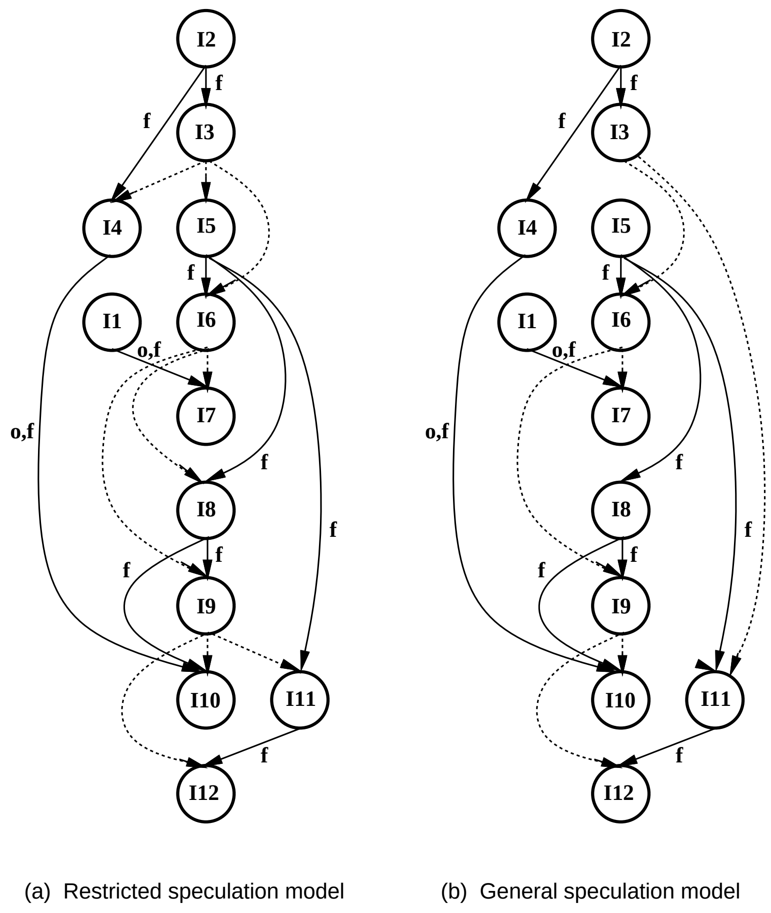
The superblock is now ready for scheduling. A dependence graph for each superblock is constructed as the first phase of superblock scheduling. Using the *LIVE-OUT()* information for each branch (Figure 3.21) and the scheduling model, the dependence graphs shown in Figure 3.22 are built. The data dependences are represented by solid arcs and labelled with f for flow and o for output (there are no anti-dependences in the superblock). The control dependences are represented by dashed arcs. In both models the number of data dependences is the same. However, the number of control dependences is reduced from nine with the restricted speculation model to six with the general speculation model. More importantly, the control

```
                  ld_i   r1,   ptr,  0
                  mov    r7,   0             // avg
                  mov    r2,   0             // count
                  mov    r3,   0             // weight
                  beq    r1,   0,    L3
(I1)      L0:     add    r2,   r2,   1
(I2)              ld_i   r4,   r1,   0       // ptr−>wt
(I3)              blt    r4,   0,    L1
(I4)              add    r3,   r3,   r4
(I5)              ld_i   r5,   r1,   4       // ptr−>next
(I6)              beq    r5,   0,    L3
(I7)              add    r2,   r2,   1
(I8)              ld_i   r6,   r5,   0       // ptr−>wt
(I9)              blt    r6,   0,    L1’
(I10)             add    r3,   r3,   r6
(I11)             ld_i   r1,   r5,   4       // ptr−>next
(I12)             bne    r1,   0,    L0
          L3:     beq    r2,   0,    L4
                  div    r7,   r3,   r2
                  st_i   avg,  0,    r7
          L4:
                     .
                     .
                     .

          L1’:    mov    r1,   r5
                  mov    r4,   r6
          L1:     sub    r3,   r3,   r4
                  ld_i   r1,   r1,   4       // ptr−>next
                  bne    r1,   0,    L0
```

Figure 3.20 Assembly code of C segment after superblock formation and loop unrolling.

```
live-out(I3) = {r1, r3, r4}
live-out(I6) = {r2, r3, r7}
live-out(I9) = {r3, r5, r6}
```

Figure 3.21 Live-out sets for superblock loop branch instructions.



I2
I3
I4
I5
I1
I6
I7
I8
I9
I10
I11
I12
f
o,f
(a) Restricted speculation model
(b) General speculation model

Figure 3.22 Dependence graphs for the restricted and general speculation models.

Model:	**Restricted**	**General**
Restrictions:	*1 and 2*	*1*
Schedule:		
t1:	{I1, I2}	{I1, I2, I5}
t2:		
t3:	{I3, I4, I5}	{I3, I4, I8, I11}
t4:		{I6}
t5:	{I6, I7, I8}	{I7, I9, I10, I12}
t6:		
t7:	{I9, I10, I11}	
t8:		
t9:	{I12}	
Without hardware support:	*executes properly*	*segmentation violation*

Figure 3.23 Code schedules and execution results obtained with the restricted and general speculation models.

dependences limiting the upward movement of each of the loads are either removed, such as the control dependence from I3 to I6, or switched to more distant instructions, such as the control dependence from I9 to I11 is changed I3 to I11.

Scheduling the dependence graphs assuming no limitations on processor resources results in the code schedules shown in Figure 3.23. For the restricted speculation model, the loop takes nine cycles to execute and the program executes properly without additional hardware support. With the general speculation model, the loop execution time is reduced to five cycles, because the load instructions, I5, I8, and I11, can be initiated sooner, thereby reducing the critical path length in the superblock. With the aggressive code scheduling done with general speculation, it is possible to introduce spurious exceptions for load instructions I8 and I11. This will occur if the link list has an odd number of elements in it. Therefore, without silent versions of load instructions, the program may terminate incorrectly. But, by converting the speculative load instructions into their silent version, the code will execute properly.

CHAPTER 4

SENTINEL SPECULATION MODEL

There are two major problems associated with speculative code motion. The first problem is that the result value of a speculative instruction that is not required to execute must not affect the execution of the subsequent instructions. This may be effectively achieved by compile-time renaming transformations as discussed in the previous chapter. A more serious problem with speculative code motion is correct handling of exceptions. An exception that occurs for a speculative instruction that is not supposed to execute must be ignored. On the other hand, an exception for a speculative instruction that is supposed to execute must be signaled. Accurately detecting and reporting exceptions are required to identify program execution errors at the time of occurrence. Also, for exceptions that do not terminate program execution, exception recovery must be possible.

In this chapter, a set of architectural features and compile-time scheduling support, collectively referred to as *sentinel speculation*, is described. Sentinel speculation provides an effective framework for speculative execution, and also provides a means to efficiently handle exceptions that occur for speculative instructions.

4.1 Background and Related Work

Varying degrees of speculative code motion can be supported with different speculation models. In Chapter 3, the two most extreme speculation models, restricted and general speculation, were presented. Neither of these models handles exceptions efficiently. With the restricted speculation model, the compiler may only speculate those instructions it can guarantee will never cause exceptions. Without sophisticated compile-time analysis, the performance of this approach is extremely limited. With the general speculation model, speculative instructions, which may potentially cause an exception, are converted into their silent version. In this manner, any exceptions which may arise due to speculative instructions are ignored. In this section, a survey of proposed speculation models which handle exceptions to varying degrees and with varying hardware/software costs is presented. The models will be presented in the context of superblock scheduling. However, they can easily be generalized to other scheduling paradigms, such as trace scheduling [10], modulo scheduling [68] and enhanced pipelining [69].

4.1.1 Instruction boosting speculation model

The instruction boosting speculation model provides an effective means for speculating instructions by removing both restrictions for upward code motion within a superblock (for definitions of the restrictions on upward code motion, see Section 3.4) [70],[71]. The restrictions are overcome by providing sufficient hardware storage to buffer results until the branches an instruction is moved above are committed. If all branches are found to be correctly predicted, the machine state is updated by the boosted instructions' effects. If one or more of the branches are incorrectly predicted, the buffered results are thrown away. Two sets of buffer storage are required for this speculation model, shadow register files and shadow store buffers. The shadow register files hold the results of all boosted instructions which write into a register, while the shadow store buffers hold the results of all boosted store instructions.

Exceptions for boosted instructions are handled by marking in the appropriate shadow structure whether an exception occurred during execution. At the excepting instruction's commit point, the contents of the shadow structure are examined to determine if an exception condition exists. If an exception condition exists, all information in the shadow structure is discarded and a branch is made to a compiler-generated recovery block. The excepting instruction is identified by sequentially re-executing all speculative instructions which are committed by the same branch instruction. The exception condition is therefore regenerated in a sequential processor state. Operands of speculative instructions are preserved by ensuring that speculative instructions do not update the architectural register file until they are committed. Therefore, an uncommitted speculative instruction may always be re-executed by retrieving its operands from the architectural register file. Finally, the exception is handled (either terminating program execution or recovering from the exception) using traditional exception handling techniques since the exception is regenerated in a sequential processor state.

4.1.2 Writeback suppression speculation model

The writeback suppression speculation model also provides an effective means for speculating instructions by eliminating Restriction 2 for upward code motion [72]. This technique is based on two main concepts: delay the exception for a speculative potentially excepting instruction (SPEI) until its execution is confirmed, and prevent corruption of the source operands of instructions by systematically suppressing updates to the register file after a speculative instruction has excepted. Writeback suppression uses knowledge of the original basic block of speculated instructions to determine which instructions to suppress. Any instruction that was originally from a later basic block than the excepting speculative instructions is prevented from updating the processor state, whereas instructions from earlier basic blocks update the state normally.

Once the processor confirms that the exception should take place, the exception is either signaled or recovery is initiated. When recovery is possible, the excepting instruction's PC is obtained from a pushdown stack which records the PC of all instructions which except. The exception is processed using traditional exception handling techniques. Finally, the excepting instruction and all instructions whose execution was suppressed during the initial execution are re-executed. Normal execution resumes from the point at which the exception was detected.

4.1.3 Partial ignoring of speculative exceptions

The Multiflow Trace system employs a technique to allow speculative execution of instructions which produce a floating-point value [73]. A speculative floating-point instruction that excepts, writes *NaN* into its destination register. Exceptions are detected by the use of *NaN* by a nonspeculative instruction. This method, however, has difficulties determining the original excepting instruction, and is not guaranteed to signal an exception if the result of a speculative exception-causing instruction is conditionally used. Also, an equivalent integer *NaN* must be provided for this method to work for integer instructions.

A large number of current and past architectures disable a subset of exceptions to improve performance through speculative execution. The Cydra 5 architecture is able to disable exceptions for floating-point memory instructions and arithmetic operations [18],[74]. The HP Precision Architecture supports nonexcepting floating-point instructions and nonexcepting loads of address zero [75]. The SPARC architecture supports nonexcepting floating-point instructions [76].

In summary, instruction boosting provides an effective framework for speculative code motion of instructions and handling of exceptions that occur for speculative instructions. However, the hardware overhead is very large, and the number of branches that an instruction can be boosted above is limited to a small number. Write-back suppression requires less hardware overhead than boosting; however, the speculation distance is limited by the number of bits provided in each instruction to mark the speculation distance. General speculation, on the other hand, provides for the largest speculation freedom at a much lower implementation cost. The problem is that there is no guarantee of detecting exceptions and determining the cause of an exception. In the next section, a new speculation model referred to as sentinel speculation is introduced. With a modest amount of architectural support, sentinel speculation permits all the scheduling freedom of general speculation, while allowing exceptions to be always detected and the excepting instruction accurately identified.

4.2 The Sentinel Speculation Model

In this section, a speculation model referred to as sentinel speculation is introduced [77],[78]. Sentinel speculation combines a set of architectural features with sufficient compile-time support to accurately detect and report exceptions for compiler-scheduled speculative instructions. The basic idea behind this technique is to provide a *sentinel* for each *potentially excepting instruction* (PEI). The sentinel reports any exceptions that were caused when the PEI is speculated. The sentinel can either be an existing instruction in the program or a newly created instruction. In the following subsections, the model of execution, the required architectural support, the algorithm for sentinel superblock scheduling, and several other important issues are described.

4.2.1 Model of execution

Conceptually, each instruction, J, can be divided into two parts, the nonexcepting part that performs the actual operation, and the sentinel part that flags an exception if necessary. The nonexcepting part of J can be speculatively executed, provided the sentinel part of J remains in J's *home block*. The home block of an instruction is the original basic block in which the instruction resides before compile-time scheduling. The sentinel part of J can be eliminated if there is another instruction, K, in J's home block which uses the result of J. The sentinel part of K will signal any exceptions caused by both J and K, which makes it a shared sentinel between J and K. Applying this argument one level further, if an instruction, L, in K's home block which uses the result of K can be found, its sentinel part may serve as the shared sentinel of J, K, and L. In this case, the semantics of K are defined so as to propagate an incoming exception from J to L's sentinel.

For each PEI, a recursive search may be applied to identify a tree of instructions which use its result. The search terminates along a path when an instruction that has no uses in its home block is encountered.[1] Such an instruction is termed an *unprotected instruction*. If all instructions in a PEI's tree of uses are speculatively executed, an explicit instruction must be created to act as the sentinel of the PEI. The explicit sentinel is restricted to remain in the PEI's home block.

Since some instructions may never result in exceptions, e.g., integer add, the sentinel part is not required for all instructions. An instruction only requires a sentinel part if it may cause an exception, or it is used to report an exception for a dependent PEI.

[1]Note that a post dominating use is sufficient to guarantee that all exceptions will be detected. However, a use in the home block is required in our implementation to facilitate earlier reporting of exceptions, re-execution of fewer instructions for recovery, and reducing register lifetimes.

Table 4.1 Exception detection with sentinel speculation.

spec	*src(J).ex_tag* †	*J causes except.*	*dest(J).ex_tag*	*dest(J).data*	*except. signal*
0	0	0	0	result of J	none
0	0	1	0	-	yes, except. PC = PC of J
0	1	0	0	-	yes, except. PC = $src(J).data$ ‡
0	1	1	0	-	yes, except. PC = $src(J).data$ ‡
1	0	0	0	result of J	none
1	0	1	1	PC of J	none
1	1	0	1	$src(J).data$ ‡	none
1	1	1	1	$src(J).data$ ‡	none

† union of all source operand exception tags of J

‡ the first source operand of J whose exception tag is set

4.2.2 Architectural support

In order to support sentinel speculation, several extensions are required to the processor architecture. The first extension is an additional bit in the opcode field of an instruction to represent a speculatively executed instruction. This additional bit is referred to as the *speculative modifier* of the instruction. The compiler sets the speculative modifier for all instructions that are speculatively scheduled. A second extension is an *exception tag* added to each register in the register file. The exception tag is used to mark an exception that occurs when a speculative instruction is executed.[2] The exception tag associated with each register must be preserved along with the data portion of that register whenever the contents of the register are temporarily stored to memory during context switch.

A summary of exception detection using the sentinel speculation model is shown in Table 4.1. For each instruction, J, three inputs are examined, the speculative modifier of J, the exception tag of the source registers of J, and whether J causes an exception. A single bit is used for the exception tag to simplify this discussion.

[2]Note that the minimum exception tag required is a single bit. However, in some cases a larger tag may be useful to indicate the type of exception to assist in debugging and exception handling.

Execution of a speculative instruction. When J is a speculative instruction, exceptions will not be signaled immediately. If all the source register exception tags of J are reset, conventional execution results when J does not cause an exception. When J does cause an exception, the exception tag of the destination register is set, and the program counter (PC) of J is copied into the data field of the destination register. The PC of J can be obtained from a PC History Queue which keeps a record of the last m PC values to enable reporting exceptions with nonuniform latency function units [18],[73]. If one or more of the source register exception tags of J are set, an exception propagation occurs. This is independent of whether J causes an exception or not. For this case, the destination register exception tag is set, and the data of the source register with exception tag set are copied into the destination register. If more than one of the source registers of J have their exception tag set, the data field of the first such source is copied into the destination register. The implications regarding this issue will be discussed in Section 4.2.6.

Execution of a nonspeculative instruction. If J is not a speculative instruction, conventional execution results if all source registers have their exception tags reset. When J causes an exception, the exception is signaled immediately, and J is reported as the exception-causing instruction. Conversely, when one or more of the source register exception tags are set, an exception has occurred for a speculatively executed instruction for which J serves as the sentinel. The exception is, therefore, signaled and the data contents of the source register with its exception tag set are reported as the PC of the exception-causing instruction. Again, if more than one source register has its exception tag set, the data field of the first such source operand is reported as the PC of the exception causing instruction.

Additional sentinel instruction. The final extension to the processor is an additional instruction called $check(reg)$. This instruction is inserted as the explicit sentinel when no use of a speculative PEI exists in its home block. This instruction does not perform any computation, but rather is merely used to check the exception tag of its source register. For most processors, a new opcode does not have to be created, but rather a move instruction can be used instead. The destination register of the move is either set to the same as the source register or to a register hardwired to 0, such as R0 in the MIPS R2000 [79].

4.2.3 Sentinel superblock scheduling algorithm

As previously discussed, superblock code scheduling consists of two major steps, dependence graph construction and list scheduling. The dependence graph contains dependence arcs to represent all data and control dependences between instructions in the superblock. With the sentinel speculation model, only restriction (1) (Section 3.4) is enforced for inserting control dependences. Therefore, a control dependence arc from a branch instruction, BR, to another instruction, J, is inserted if the location written to by J is used before being redefined when BR is taken. This is the same restriction applied using the general speculation model. As with general speculation, memory stores are not allowed to be speculative. However, in Section 4.4, an extension to remove this constraint will be discussed.

Prior to list scheduling, an additional step is added for sentinel superblock scheduling. In this step, potential sentinels are identified for each PEI the scheduler is allowed to speculate. In general, any instruction from a PEI's home block which uses the result of the PEI is a potential sentinel. However, a simplifying assumption to recognize potential sentinels only along one path in the dependence graph is made. This assumption is utilized to reduce the complexity associated with exception recovery (Section 4.3). The overhead associated with limiting the number of potential sentinels is that a larger number of explicit sentinels may be inserted than are required. This overhead is discussed further in Chapter 5.

An algorithm to identify potential sentinel instructions for all PEIs in a superblock is presented in Figure 4.1. For each PEI, a leaf node in the dependence subgraph is identified. Then all instructions along that path are marked as potential sentinels for the PEI. Instructions with a successor in the chain are marked as protected. Protected instructions may be freely speculated since the next instruction in the chain will check all exceptions propagated or caused by the instruction. The last instruction in the chain is marked as unprotected. If an unprotected instruction is speculated, an explicit sentinel must be created by the scheduler to check all exception conditions propagated through the chain. The last instruction in the chain is also recorded as the last potential sentinel for the PEI.

A modified form of list scheduling for superblocks to insert the necessary explicit sentinels is then performed as the final step of sentinel superblock scheduling. The algorithm used is presented in Figure 4.2 with the additions to the basic superblock scheduling algorithm in bold type. The only modification required for sentinel superblock scheduling is to insert an explicit sentinel instruction when an unprotected instruction is speculated. The explicit sentinel is restricted to be scheduled in the instruction's home block by adding the appropriate control dependences. Note that the algorithm contains two function calls for handling exception recovery. Exception recovery with sentinel speculation is discussed in Section 4.3.

```
identify_potential_sentinels(superblock) {
    /* initialization, mark all instructions as protected */
    for each instruction in superblock, J {
        J→protected = 1
        J→sentinel = NULL
    }
    /* identify potential sentinels */
    for each instruction in superblock, J {
        if (J not allowed to be speculative)
            continue
        if ((J is unprotected) OR (J is potentially excepting)) {
            use = instruction in home_block(J) such that there is a flow
                dependence from J to use
            /* use in home block serves as the sentinel for J if J is speculated */
            if (use) {
                J→protected = 1
                use→protected = 0
                J→sentinel = use
                /* Do not allow potential sentinel to move to subsequent block */
                add control dependence from use to use→post_branch
            }
            /* No use in the home block so instruction is marked as unprotected */
            else {
                J→protected = 0
            }
        }
    }
    /* Identify last potential sentinel for each PEI which may be speculative */
    for each instruction in superblock, J {
        if (J is potentially excepting) {
            last = J
            while (last→sentinel)
                last = last→sentinel
            J→last_potential_sentinel = last
        }
    }
}
```

Figure 4.1 Algorithm to identify potential sentinel instructions.

```
schedule(superblock) {
    build_dependence_graph(superblock)
    identify_potential_sentinels(superblock)
    clear resource_usage_map
    issue_time = 0
    while (unscheduled set of instructions is not empty) {
        issue_time += 1
        active_set = set of unscheduled instructions that are ready
        sort active set according to instruction priority
        for each instruction in active_set, J {
            if (not all resources J requires are free)
                continue
            if ((enable_recovery) AND (! compatible_with_active_intervals(J)))
                continue
            /* J is scheduled at issue_time */
            mark required resources of J busy in resource_usage_map
            delete J from set of unscheduled instructions
            J→issue_time = issue_time
            if (J is speculative) {
                set speculative modifier of J
                /* create an explicit sentinel if speculate an unprotected instruction */
                if (J is unprotected) {
                    create a new instruction, check(dest(J))
                    J→sentinel = check
                    add flow dependence J to check
                    /* Restrict explicit sentinel to remain in J's home block */
                    add control dependence from J→prev_branch to check
                    add control dependence from check to J→post_branch
                    insert check into set of unscheduled instructions
                }
            }
            if (enable_recovery) update_intervals(J)
            /* check for control-flow hazards associated with an downward code motion */
            for each branch J moved below in superblock, BR {
                if (J→dest not live when BR is taken)
                    continue
                insert a copy of J into target superblock of BR
            }
        }
    }
}
```

Figure 4.2 Sentinel superblock scheduling algorithm.

4.2.4 Sentinel speculation example

To illustrate exception detection with sentinel speculation, consider the assembly code fragment shown in Figure 4.3(a). For simplicity, it will be assumed in this example that each instruction requires one cycle to execute and that the processor has no limitations on the number of instructions that can be issued in the same cycle. Also, it will be assumed that memory loads and stores are the only instructions that may cause exceptions. In the example, potentially excepting instructions B and C may be speculated; therefore, a sentinel instruction must be kept in the home block of B and C to check their exception status if they are speculated.

The potential sentinels for B are identified as D and F. Since F is the last use in the chain of flow dependences, it is marked as unprotected (Figure 4.3(a)). Similarly, the potential sentinel for C is E, which is unprotected since it is the last use in this chain. The code segment after scheduling is shown in Figure 4.3(b). Four instructions (B, C, D, and E) are moved above the branch (A); therefore, their speculative modifiers are set. Instruction E, though, is unprotected, so an explicit sentinel (G) must be inserted into E's home block to check the exception condition of C. In the final schedule, instructions F and G serve as sentinels for the potentially excepting instructions B and C, respectively.

An execution sequence for the scheduled code segment in which instruction B causes an exception is shown in Figure 4.4. For this example, it is assumed that the branch, instruction A, is not taken. The initial states of all the registers are further assumed to have reset exception tags and some unknown data fields. In the first cycle, instruction B causes an exception. However, since it is a speculative instruction, the exception is not yet signaled. Instead, the exception tag of the destination register of instruction B is set, and the PC of instruction B is copied into the destination register's data field. In the second cycle, instruction D finds the

```
  A: beq r2,0,L1
  B: ld_i r1,r2,0
  C: ld_i r3,r4,0
  D: add r4,r1,1
† E: mul r5,r3,9
† F: st_i r2,4,r4

† unprotected instruction
```

(a)

```
* B[1]: ld_i r1,r2,0
* C[1]: ld_i r3,r4,0
* D[2]: add r4,r1,1
* E[2]: mul r5,r3,9
  A[2]: beq r2,0,L1
‡ F[3]: st_i r2,0,r4
‡ G[3]: check r5
*       speculative instruction
‡       sentinel
[n]     indicates in which cycle the instruction is executed
```

(b)

Figure 4.3 Example of sentinel speculation, (a) original program segment, (b) program segment after scheduling.

exception tag of its first source register set. However, since it is also a speculative instruction, it propagates the exception information to its destination register. Finally, in cycle 3, instruction F detects that the exception tag of its first source register is set. Since instruction F is not a speculative instruction, an exception is signaled and the cause of the exception is reported as the contents of $r4$.

Note that in this example, if instruction B again results in an exception but the branch instruction A is taken instead, the exception is completely ignored. This result is correct because if the branch is taken, instruction B should not have been executed and, therefore, should not disrupt the program's execution.

4.2.5 Handling uninitialized data

The use of an uninitialized register can potentially cause incorrect exceptions to be reported with the sentinel speculation model. Registers that are referenced before being defined in a function may have their exception tag set from the execution of a previous function or program. The use of this register will therefore lead to an immediate or eventual exception

Initial

	except tag	data
r1	0	
r2	0	
r3	0	
r4	0	
r5	0	

After Cycle 1

	except tag	data
r1	1	B
r2	0	
r3	0	
r4	0	
r5	0	

B causes an exception

After Cycle 2

	except tag	data
r1	1	B
r2	0	
r3	0	
r4	1	B
r5	0	

After Cycle 3

	except tag	data
r1	1	B
r2	0	
r3	0	
r4	1	B
r5	0	

signal exception

report B as source

Figure 4.4 Example of exception detection using sentinel speculation.

signal. However, this exception should not be reported. To prevent an exception from occurring with uninitialized registers, the compiler performs live variable analysis [52] and inserts additional instructions into the beginning of a function to reset the exception tags of the corresponding registers. Therefore, spurious errors associated with referencing uninitialized registers or variables are prevented.

4.2.6 Reporting multiple exceptions

Multiple exceptions in a program are handled efficiently with sentinel speculation. The exceptions can occur either within different basic blocks or within the same basic block. When two exceptions occur in different basic blocks, the exceptions are guaranteed to be detected in the proper order because exceptions for all instructions of a basic block are checked before the basic block is exited. The requirement of a sentinel in the home block of each speculative instruction enforces this condition.

For multiple exceptions in the same basic block, exceptions are not guaranteed to be detected in the proper order according to the original code sequence. Multiple excepting instructions in the same basic block may either have different sentinels or share a sentinel. With different sentinels, the first sentinel executed will signal the first exception. When two excepting instructions share a sentinel, multiple source registers of the sentinel instruction will have their exception tags set. In this case, one of the exceptions is arbitrarily first signaled. If a recovery mechanism is utilized, as discussed in the next section, the second exception is reported when the sentinel is re-executed. The order of reporting two exceptions in the same basic block is difficult to maintain in most systems. In many cases, instructions within a basic block are reordered by conventional compiler code optimizations. Therefore, an order of reporting exceptions in the same basic block is not maintained with the sentinel speculation model.

4.3 Exception Recovery

For many types of exceptions, it is desirable to recover from the exception rather than abort program execution. Recovery generally consists of repairing the excepting instruction and continuing program execution. Recovery with speculative instructions is difficult because the exception condition may not be raised until long after the instruction is executed. Also, other speculative instructions that use the result of the excepting instruction are likely to have executed. Therefore, when the exception is detected and repaired, a chain of dependent speculative instructions requires re-execution to generate a correct program state. The speculation model must ensure that the excepting instruction and all dependent instructions are re-executable up to the point where the exception condition is checked.

4.3.1 Recovery model

The compiler support required to ensure recovery is dependent on the recovery model utilized. In this chapter, the recovery model assumed is as follows. A sentinel that detects an exception condition sets the processor PC to the excepting instruction's PC. The processor then enters an exception handling state which terminates when the execution reaches the sentinel again. The excepting instruction is re-executed as a nonspeculative instruction to regenerate the exception condition. After the exception is repaired, re-execution of the subsequent instructions proceeds.

Not all instructions between the excepting speculative instruction and the sentinel require re-execution. Instructions that are not re-executed are simply discarded. The minimal set of instructions that must be re-executed are those that are flow dependent on the excepting instruction. Flow-dependent instructions propagate the exception condition for the excepting speculative instruction and, therefore, must be re-executed to obtain the correct result. Any superset of the flow-dependent instructions may be chosen for re-execution. However, the re-execute set must be known by the compiler to ensure proper recovery.

Those nonspeculative instructions thus re-executed are done so as normal nonspeculative instructions (e.g., if they produce an exception, the exception is signaled immediately). Those speculative instructions thus re-executed are done so as normal speculative instructions (e.g., if they produce an exception, no exception is signaled but the exception tag and data field of the excepting instruction's destination register are set appropriately) with one modification.

Transparent exceptions must be handled immediately for speculative instructions in the exception handling state. Transparent exceptions include such events as page faults and TLB misses which occur independently of the program logic. This is necessary to ensure that speculative instructions that did not except in their original execution produce the correct result during re-execution. When execution reaches the original sentinel instruction again, the exception handling state is exited and normal execution resumes.

Note that other models of recovery may be utilized in conjunction with sentinel speculation. One effective alternative is the use of recovery blocks [71],[80]. In this model, the compiler generates the exact sequence of instructions that must be re-executed when an exception is detected by a particular sentinel. The advantage of this scheme is the reduced complexity in the exception handling state. The disadvantage of this scheme is that the static code size increases due to the recovery blocks.

4.3.2 Restartable instruction interval

To ensure recovery with the recovery model discussed in the previous section, each PEI that is speculated and its sentinel must delineate the endpoints of a restartable instruction interval. The interval consists of two types of instructions based on their execution status during recovery, those which are re-executed (RE) and those which which are not re-executed (NRE). An instruction interval is restartable if all elements of the interval satisfy the following constraints. First, none of the instructions in the interval may prevent re-execution of the RE instructions in the interval. These instructions will be referred to as *irreversible instructions*. For the purposes of this thesis, an irreversible instruction has one of the three following properties: the instruction destroys the exception tag of a live register, the instruction modifies an element of

```
update_interval_live_set(interval, J) {
    if (J is in interval→RE) {
        if (src_i(J) not in interval→def)
            interval→use = interval→use + {src_i(J)}
        if ((J is a memory load) AND (J not in interval→mem_def))
            interval→mem_use = interval→mem_use + {J}
        interval→def = interval→def + {dest_i(J)}
        if (J is a memory store)
            interval→mem_def = interval→mem_def + {(J)}
    }
}
```

Figure 4.5 Algorithm to calculate live information for an instruction interval.

the processor state, which causes intolerable side effects, or the instruction cannot be executed more than one time. As a result, synchronization, I/O, and subroutine call instructions break restartable intervals.[3] Based on these properties, memory stores are not considered irreversible instructions.

The second constraint is that the operands of all RE instructions that are live at the start of the interval are not overwritten by any instruction in the interval. An operand is live if it is used by an RE instruction before it is defined by an RE instruction in the interval. The live set of an interval is calculated using techniques for standard dataflow analysis [52]. However, only the RE instructions in the interval are considered in the process. An algorithm to compute the live information for an instruction interval is presented in Figure 4.5. Live information is computed for both register and memory operands since both must be maintained to ensure a restartable interval.

To illustrate the computation of an interval live set, consider the example in Figure 4.6(a). The register live set consists of $r2$, $r3$, and $r5$. Although $r3$ is computed by B before it is

[3]Note that if an architecture provides a means for the compiler to save/restore exception tags, a subroutine call alone is not an irreversible instruction. However, in this discussion it is assumed that subroutine calls are irreversible.

```
PEI,RE     A:  ld_i r1,r2,0
NRE        B:  add r3,r4,1
RE         C:  mul r4,r1,r5
RE         D:  add r6,r3,r4
RE         E:  ld_i r4,r3,0
RE         F:  sub r7,r1,1
Sentinel   G:  check r4
```

(a)

```
PEI,RE     A:  ld_i r1,r2,0
NRE        B:  add r3,r4,1
NRE        C:  mul r4,r1,r5
RE         D:  add r6,r3,r4
RE         E:  ld_i r4,r3,0
RE         F:  sub r3,r1,1
Sentinel   G:  check r4
```

(b)

Figure 4.6 Instruction interval examples, (a) a restartable instruction interval, (b) a non-restartable instruction interval.

used in the original execution of the interval, instruction B is an NRE instruction; therefore, $r3$ will not be re-defined during re-execution. Consequently, its contents must be preserved all the way to the end of the interval (during the original execution) and from the beginning of the interval (during re-execution) to ensure that D may re-execute correctly. Also, note that even though B uses $r4$ before it is defined, $r4$ is not included in the live set, because B is an NRE instruction. Therefore, its input operands may be modified without affecting the restartability of the interval.

The instruction interval shown in Figure 4.6(a) is thus restartable since none of the registers in the live set are modified in the interval. An example interval that is not restartable is shown in Figure 4.6(b). The register live set consists of $r2$, $r3$, and $r4$. The conditions for restartability are violated by two instructions in the interval. Instruction E overwrites $r4$ which prevents D from properly re-executing. Similarly, F overwrites $r3$ which prevents D and E from properly re-executing.

The compiler must maintain a restartable instruction interval for all PEI/sentinel pairs that are generated to ensure that exception recovery may be performed. From the point of view of individual instructions, the compiler must satisfy the restartability constraints for all intervals which span an instruction. In the remainder of this section, the required scheduler and register allocator support to maintain restartable instruction intervals is presented. Also, a discussion of various recovery models and the recovery model utilized for the experimental evaluation is presented.

4.3.3 Scheduler support

Several additional restrictions must be added to the instruction scheduler to ensure that all instruction intervals are restartable. The additional restrictions are as follows.

(1) A speculative instruction cannot be moved beyond any irreversible instruction.

(2) Exceptions for speculative instructions are not propagated across irreversible instructions.

(3) A speculative instruction may not modify any of its own input operands.

(4) All instructions which overwrite an operand in an instruction interval's live set may not be scheduled in the interval.

The first two scheduling restrictions are used to prevent an irreversible instruction from being included in any instruction interval. The first restriction is handled by inserting additional control dependences during dependence graph construction. A dependence arc is inserted from each irreversible instruction into all subsequent instructions in the superblock. The second restriction is maintained by modifying the definition of home block to account for irreversible instructions. Each irreversible instruction defines an additional basic block boundary as far as the scheduler is concerned. In this manner, the identify_potential_sentinels algorithm (Figure 4.1) will not search beyond an irreversible instruction for flow-dependent instructions.

The last two scheduling restrictions are used to ensure that the live operands of all RE instructions in the interval are not destroyed by any instruction in the interval. Compile-time renaming is utilized to overcome the third restriction. The destination of an instruction that may be speculated and overwrites one of its source operands (self-anti-dependence) is renamed to a new register. All uses of the original register are then replaced with the renamed register. If necessary, a copy instruction is inserted to restore the proper value of the original register.

The fourth restriction is overcome by modifying the sentinel superblock scheduling algorithm. The modifications employ two functions that check and update the status of instruction intervals. The calls to these functions are included in the sentinel superblock scheduling algorithm shown in Figure 4.2. The first function is used to determine if scheduling an instruction at the current time is compatible with all active intervals. An instruction interval is active at the current time if the start of the interval has been scheduled and the end of the interval has not been scheduled. An instruction is compatible with an active interval if the interval remains restartable when the instruction is added. An NRE instruction is compatible by default since there are no restrictions on the redefinition of its input operands.

An RE instruction is compatible if all instructions that modify any of its input operands that are live in the interval may be scheduled after the end instruction of the interval. Instructions that modify live operands are identified by traversing the anti-, output, memory anti- and memory output dependences of a candidate RE instruction. If the modifying instruction is independent of the interval ending instruction, it can be scheduled after the interval end point without a problem. However, if the modifying instruction is dependent on the instruction that ends the interval, the restartability of the interval may be maintained only by breaking the interval. An interval is broken by selecting an earlier potential sentinel for the end point.

The new interval end point must be independent of the modifying instruction. Therefore, the modifying instruction must be able to be scheduled in the home block. If dependences prevent scheduling the modifying instruction in the home block, the candidate instruction is not allowed to be scheduled at the current time. An algorithm to determine if an instruction is compatible with all active intervals is presented in Figure 4.7.

The last modification to the sentinel superblock scheduling algorithm is a function which updates the contents of all active intervals after each instruction is scheduled. The algorithm used to update the active intervals is shown in Figure 4.8. When a PEI is scheduled speculatively, a new active interval is created. The interval begins with the PEI and its end point is set to the last potential sentinel of the PEI. The last potential sentinel is selected as the end point of the interval to ensure that the interval is restartable for the potential sentinel selected as the actual sentinel of the PEI. Since the last sentinel is the instruction which ends the chain of flow-dependent potential sentinels, enforcing all scheduling restrictions to the last potential sentinel is sufficient to guarantee restartability.

The update algorithm also prevents instructions from overwriting the operands live in the interval by inserting additional dependence arcs. Similarly to the previous algorithm, instructions that modify live source operands are identified by traversing the anti-, output, memory anti-, and memory output dependences for a new instruction added to a interval. A dependence arc is added from the interval end point to the modifying instruction to restrict the modifying instruction from entering the interval. If the modifying instruction is dependent on the end of the interval, the interval must be broken. This is necessary to prevent a circular dependence condition between the modifying instruction and the end of the interval. The potential sentinel farthest down in the chain of flow dependences that is not dependent on the modifying

```
compatible_with_active_intervals(J) {
    /* Create a temporary interval for J if it is a speculated PEI */
    if ((J is speculative) AND (J is potentially excepting)) {
        create a new active interval, temp
        temp→start = J
        temp→end = last potential sentinel of J
        temp→RE = temp→NRE = {}
    }
    compatible = 1
    for each active interval, interval {
        if (J in NRE for interval) continue
        /* Save the contents of all fields of interval, so they can be later restored */
        original = copy all elements of interval
        interval→RE = interval→RE + {J}
        update_interval_live_set(interval, J)
        /* Determine if any of J's operands in the use set of the interval are modified by
                instructions which cannot be scheduled outside the interval */
        for each dependence arc out of J, dep {
            if (((dep→type is anti or output) AND (dep→operand in interval→use)) OR
                    ((dep→type is memory anti or memory output) AND (dep→to_instr in interval→mem_use))) {
                /* An instruction not dependent on the end of the interval may always be moved
                        after the end of the interval to satisfy the dependence constraint */
                if (there is no dependence path from dep→to_instr to interval→end)
                    continue;
                /* Otherwise, the interval can be broken if dep→to_instr can be moved into
                        the home block of instruction which ends the interval */
                else if (there is a dependence path from dep→to_instr to interval→prev_br) {
                    compatible = 0
                    break
                }
            }
        }
        restore contents of interval with original
        if (! compatible) break
    }
    delete temp
    return (compatible)
}
```

Figure 4.7 Algorithm to determine if an instruction is compatible with all active intervals.

```
update_intervals(J) {
    /* Create a new interval for a speculated PEI */
    if ((J is speculative) AND (J is potentially excepting)) {
        create a new active interval, interval
        interval→start = J
        interval→end = last potential sentinel of J
        interval→RE = interval→NRE = {}
        interval→use = interval→def = interval→mem_use = interval→mem_def = NULL
    }
    /* De-activate all intervals which end with J, note that J must be
            non-speculative to end an interval */
    for each active interval, interval {
        if (interval→end==J) deactivate interval
    }
    /* update all active intervals with J */
    for each active interval, interval {
        /* The recovery model utilized defines if J is an RE or NRE instruction for each interval */
        if (J in NRE for interval) {
            interval→NRE = interval→NRE + {J}
            continue
        }
        interval→RE = interval→RE + {J}
        update_interval_live_set(interval, J)
        for each dependence arc out of J, dep {
            if (((dep→type is anti or output) AND (dep→operand in interval→use)) OR
                    ((dep→type is memory anti or memory output) AND (dep→to_instr in interval→mem_use))) {
                if (there is a path in the dependence graph from dep→to_instr to interval→end) {
                    /* break up the interval to satisfy the dependence constraint */
                    S = farthest instruction in the flow dependence chain of potential sentinels for
                            interval→start that is not dependent on dep→to_instr
                    if (S is not speculated) {
                        mark S as unprotected
                        interval→end = S
                    }
                    else {
                        /* Create an explicit sentinel since all potential sentinels for the broken
                                interval have been speculated */
                        create a new instruction, check(dest(S))
                        add a flow dependence from S to check
                        add a control dependence from S→prev_branch to check
                        add a control dependence from check to S→post_branch
                        interval→end = check
                        add check into set of unscheduled instructions
                    }
                }
                insert a dependence of type dep→type between interval→end and dep→to_instr
            }
        }
    }
}
```

Figure 4.8 Algorithm to update all active intervals.

```
   A:  jsr foo               A[1]:  jsr foo
   B:  ld_i r5,r3,0        * D[2]:  ld_i r1,r6,0
   C:  beq r5,0,L1           B[2]:  ld_i r5,r3,0
†  D:  ld_i r1,r6,0        * E'[2]: add r10,r2,1
†  E:  add r2,r2,1           C[3]:  beq r5,0,L1
   F:  st_i r4,0,r7        ◇ G[4]:  add r8,r1,1
‡  G:  add r8,r1,1           F[5]:  st_i r4,0,r7
   H:  ld_i r6,r2,0          H'[5]: ld_i r6,r10,0
                             I[5]:  mov r2,r10
```

†	instruction considered for speculative execution
‡	last potential sentinel for D
∗	speculative instruction
◇	sentinel for D
[n]	indicates in which cycle the instruction is executed

(a) (b)

Figure 4.9 Example of sentinel superblock scheduling to ensure recovery, (a) original program segment, (b) program segment after scheduling.

instruction is selected as the new end point. If no such instruction exists, an explicit sentinel is created to serve as the end point of the interval.

An example to illustrate the handling of the scheduling restrictions is presented in Figure 4.9. For this example assume that each instruction requires one cycle to execute, the processor has unlimited resources, and only memory load and store instructions may cause exceptions. It is further assumed that all instructions in an interval are RE. The first restriction is for *A*. Instruction *A* is an irreversible instruction; therefore, no speculative code motion is allowed across it. Instruction *D* may be speculated provided several constraints are observed. First, *H* overwrites a source operand of *D*. Therefore, *H* must be scheduled after the end point of the interval started by *D*, namely *G*. Similarly, if the compiler cannot determine that instructions *D* and *F* access different memory locations, *F* must be scheduled after *G* (due to memory anti-dependence).

Instruction E may also be speculated in the example. Instruction E is self anti-dependent; therefore, the destination of E must be renamed to a new register ($r10$ in the example). All uses of the original register $r2$ are also renamed to $r10$, and a copy instruction, I, is inserted assuming $r2$ is live outside the code segment. Let E' be the instruction derived from E by renaming $r2$ to $r10$. Since the copy instruction I is anti-dependent on E', the copy is restricted to be scheduled after the end point of all intervals which contain E'. Thus, I is scheduled after G. The final schedule with all restrictions observed is shown in Figure 4.9(b). Note that if speculative instruction D causes an exception, the exception is detected by its sentinel G. Instructions D, B, E', and C are then re-executed during exception handling mode. All instructions re-execute correctly since their source operands have not been destroyed.

Additional compile-time renaming is also effective to minimize the number of anti-dependences that must be enforced during scheduling due to the fourth restriction. In our current implementation, anti- and output dependence removing transformations are applied to superblocks prior to scheduling [44].

4.3.4 Register allocator support

The register allocator must also be modified to ensure that all PEI/sentinel intervals are restartable. The following additional restrictions must be utilized by the register allocator so that exception recovery is possible.

(1) The contents of a register in the live set of an interval may not be overwritten in the interval.

(2) A destination register of an RE instruction may not be spilled in an interval.

The first restriction is handled by adding all instructions in an interval to the live range of each register in the interval's live set. By extending the live range of a register across all intervals in which it is live, the register contents are preserved across the necessary instructions to ensure restartability of all intervals. The algorithm to construct live ranges of each virtual register is augmented to add the contents of the intervals in which the register is live. Traditional graph coloring may then be applied to achieve the desired allocation.

In the example shown in Figure 4.9(b), assuming all instructions are re-executed during exception recovery, the live set of the interval from D to G consists of $r2$, $r3$, and $r6$. All instructions in the interval are thus added to the live ranges of these registers. As a result, even though D may be the last use of $r6$, the register allocator may not re-use the physical register mapped to $r6$ until after G.

The first restriction also implies that register source operands of speculative PEIs may not be spilled to memory by the register allocator. This is necessary with the recovery model used in this thesis because during exception handling the processor does not know how to re-execute spill load instructions to restore appropriate spilled source register operand values. In the current implementation, the register allocator enforces this restriction by de-speculating a speculative instruction whose source operands are spilled. De-speculation or downward code movement back to the PEI's home block is performed incrementally until either the live range becomes allocatable or the instruction's home block is reached. At the point at which the home block is reached, the instruction is no longer speculative, and the register allocator is free to spill its source operands.

The second restriction is necessary to ensure that improper exceptions are not signaled when a speculative instruction's destination is spilled to memory. In order to spill the register, a store instruction will read the contents of the speculative instruction's destination register. If that register contains an exception condition, an exception will be signaled. However, execution may not reach the home block of the speculative instruction. Therefore, an improper exception signal may occur. Preserving destination registers of speculative instructions may be achieved using the same de-speculation process. Speculative instructions whose destination live range cannot be allocated are incrementally moved downward until either the live range becomes allocatable or the speculative instruction's home block is reached. Again, once the home block is reached, the instruction is no longer speculative and the register allocator is free to spill its destination operand.

Note that if the architecture provides a special set of spill instructions, the second restriction for register allocation may be eliminated. The spill instructions must save/restore the exception tag along with the data contents of the register. Furthermore, the spill instruction that saves the contents of a register must ignore the exception status of the register to prevent signalling improper exceptions. Finally, the spill instructions must be included in the RE sets of all intervals which span them to ensure that updated values are placed on the stack during re-execution.

The current implementation of the scheduler does not utilize any information regarding register usage to guide the schedule. Therefore, in superblocks with a large amount of register pressure, the register allocator will be required to de-speculate many speculative instructions to satisfy the restrictions for exception recovery. More advanced scheduling techniques which integrate parts of register allocation and scheduling may be used to achieve a more efficient schedule [81],[82]. Currently, these techniques are being studied to improve the performance of sentinel superblock scheduling in regions with large register pressure.

To summarize, by enforcing constraints for scheduling and register allocation, one can guarantee that all speculative PEI/sentinel pairs form a restartable instruction interval. Therefore, an exception for a speculative instruction may be repaired and all RE instructions in the interval started by the PEI may be re-executed to achieve a correct program state. The overhead associated with enforcing these constraints is reduced scheduling freedom caused by additional dependence constraints and speculation limits imposed by the register allocator. Also, additional instructions to accomplish renaming are typically necessary.

4.4 Allowing Speculative Stores

A limitation of sentinel speculation up to this point of discussion is that it does not allow speculative store instructions. In this section, an extension to sentinel speculation is described which allows store instructions to move above branch instructions. In the following subsections, the additional architectural and compiler support required for speculative stores is presented.

4.4.1 Additional architectural support

In order to support speculatively executed store instructions, the operation of the data memory subsystem must be modified. In this discussion, it will be assumed that an N entry store buffer exists between the CPU and the data cache [47].

Operation of a conventional store buffer. A store buffer has three primary functions. First, it creates a new entry for each store instruction executed by the CPU. Each store buffer entry consists of the store address, store data, and several status bits. Address translation is performed during insertion to determine if an exception (access violation or page fault) has occurred. If an exception occurs, it is handled immediately. The store buffer also supplies data to the CPU whenever a load with a matching address to a valid store buffer entry is executed. Finally, the store buffer releases entries to update the data cache. The store buffer operates as a first-in-first-out circular queue. When the data cache is available and the buffer is not empty, the entry at the head of the queue is transferred to the data cache.

Operation of store buffer supporting speculative stores. Speculative store instructions can be utilized if the store buffer is modified to allow probationary entries, which are for speculative stores which may or may not require execution. Probationary entries are later confirmed by specific instructions if the predicted path of control is followed or invalidated when a branch direction is mispredicted. To support probationary entries, each store buffer entry requires three additional fields, a confirmation bit, an exception tag, and an exception PC. Also, an additional instruction to confirm store instructions in the store buffer, $confirm\_store(index)$, is needed. Finally, a mechanism to invalidate all probationary store buffer entries whenever a branch prediction miss occurs is required.

Each function of the store buffer requires some modifications to handle probationary entries. The insertion of a store into the store buffer is summarized in Table 4.2. Note that nonspeculative stores enter the buffer as confirmed entries, while speculative stores enter as probationary entries. Also, when the buffer is full, the processor is stalled to wait for an entry to become available. When a load instruction is executed, both confirmed and unconfirmed entries are searched for a matching address. However, a probationary entry with its exception tag set will not participate in the search.[4] This exclusion from the search is to enable re-execution of the load instruction independent from re-execution of a matching excepting store in the store buffer. The releasing function of the store buffer is changed so that probationary stores are not

[4]Note that an exception reflected in the exception tag of a probationary store buffer entry will be subsequently detected by the corresponding $confirm\_store$ instruction of the speculative store.

Table 4.2 Insertion of store into store buffer.

spec	*src(I).ex_tag* †	*I causes except* ‡	*description*
0	0	0	insert a nonspeculative store as a confirmed entry
0	0	1	force all confirmed entries at head of buffer to update cache, save contents of store buffer ◇, process exception
0	1	0	signal exception, report PC = $src(I).data$
0	1	1	signal exception, report PC = $src(I).data$
1	0	0	insert speculative store as a pending entry
1	0	1	insert speculative store as a pending entry, set exception tag, set exception PC to PC of I
1	1	0	insert speculative store as a pending entry, set exception tag, set exception PC to $src(I).data$
1	1	1	insert speculative store as a pending entry, set exception tag, set exception PC to $src(I).data$

† Instruction producing source operand of store contains exception condition, so store must just propagate the exception.

‡ The store instruction results in an exception.

◇ Saving the contents of the store buffer only necessary when speculative stores are allowed.

allowed to update the data cache. This is accomplished by preventing any releases from the store buffer when the entry at the head of the buffer is probationary.

Two additional functions are required for the store buffer, confirming and canceling probationary entries. A probationary store in the store buffer is confirmed by a $confirm\_store(index)$ instruction. The index signifies which entry is confirmed, counting from the tail entry. If the exception tag of the entry being confirmed is set, an exception must be reported. The exception is handled in the same manner as when an exception occurs during insertion of a nonspeculative store instruction. However, the PC of the excepting instruction is provided in the exception PC field of the particular store buffer entry. All probationary stores are canceled when a mispredicted branch is detected. Cancellation of a probationary store is accomplished by resetting the valid bit of the corresponding store buffer entry.

4.4.2 Scheduling support for store movement

An instruction scheduler can be extended to move store instructions above branch instructions in a straightforward manner. Stores are permitted to move above branches by removing control dependences between a store instruction and all preceding branch instructions in a superblock during dependence graph construction. All store instructions are marked unprotected by the identify_potential_sentinels algorithm (Figure 4.1) since store instructions have no destination register. Finally, list scheduling is modified to insert confirm_stores rather than checks as explicit sentinels for stores. Also, the scheduler must set the index field of the confirm_store when a store is speculated. The value of the index is the number of stores (regular and speculative) between a speculative store and its corresponding confirm.

Exception detection is not impaired by the movement of stores. A store instruction will only be confirmed when the branches it moved across have all been predicted correctly at compile time. If any of the branches are incorrectly predicted, the store is canceled. An exception for a speculative store is reported only at the time of confirmation; therefore, only exceptions for those stores that are supposed to be executed will be reported. Also, the confirm_store instruction is restricted to remain in the home block of the store; thus, exceptions occurring in different basic blocks will be reported in the proper order. Again, if multiple exceptions occur in the same basic block, the exceptions will be signaled; however, they are not guaranteed in the order of the original code sequence.

Exception recovery is also possible with speculative stores. The only modification required is to allow re-executed speculative stores to replace their corresponding probationary entries in the store buffer. This is necessary for two reasons. First, multiple store buffer entries are not allowed for a speculative store that is re-executed several times. Second, to ensure proper confirmation, the order that stores are inserted into the buffer must not be altered from the order the compiler calculated during scheduling.

A possible deadlock situation can occur when attempting to insert a store into the buffer if the store buffer is full and the entry at the head of the store buffer is unconfirmed. This situation can be prevented during scheduling by allowing a speculative store to be separated from its confirm by at most $N - 1$ (for an N entry store buffer) stores. All probationary stores, therefore, must either be confirmed or canceled within a range N stores. The size of the store buffer, though, is now an architectural parameter that must be available to the scheduler.

CHAPTER 5

EXPERIMENTAL EVALUATION OF SPECULATIVE EXECUTION

The effectiveness of speculative execution using the superblock compilation techniques is presented in this chapter. The methodology used to conduct the experiments is first described. The results, which are then presented, include the performance of speculative execution in superblocks, cost and effectiveness of superblock ILP optimizations, and instruction/data cache effects.

5.1 Experimental Methodology

All experiments performed for this thesis were done using the IMPACT simulation environment. The simulator models in detail a parameterizable superscalar processor including the prefetch and issue unit, instruction and data caches, branch prediction mechanism, and hardware interlocks. This enables the simulator to accurately model the number of cycles required to execute a program and provide detailed analyses of visible processor components such as the branch predictor and the caches.

5.1.1 Emulation-driven simulation

The IMPACT simulation approach is referred to as *emulation-driven simulation.* With emulation-driven simulation, the compiler generates code for a target, hypothetical processor exploiting advanced architectural features such as speculative and predicated execution. An

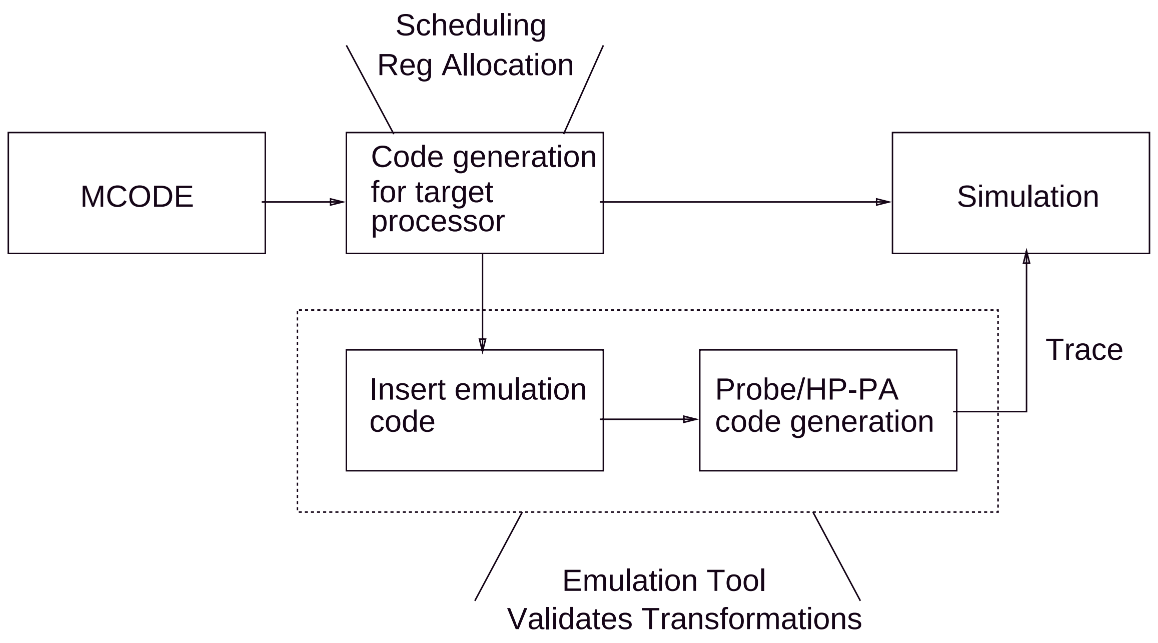

Scheduling
Reg Allocation
MCODE
Code generation for target processor
Simulation
Trace
Insert emulation code
Probe/HP-PA code generation
Emulation Tool
Validates Transformations

Figure 5.1 Simulation process.

emulation module then inserts additional assembly code into the program to model the advanced architectural features on a conventional platform. For this thesis, the emulation platform is a HP PA-RISC 700 series workstation. Code generation of the assembly code for the target processor and the emulation code produces an executable program which can be run on the emulation platform like any normal program.

Figure 5.1 illustrates the emulation-driven simulation procedure. The input to the process is Mcode optimized for the target processor. Mcode is a processor specific version of the IMPACT intermediate representation, Lcode. For all experiments in this thesis, the HP PA-RISC 1.1 instruction set was chosen as the base instruction set. Extensions to the base instruction set are provided to support speculative and for later experiments predicated execution. The Mcode is generated using all the superblock compilation techniques discussed in Chapter 3.

The first step is to schedule and register allocate the Mcode input for the target processor. The target processor is allowed to have an arbitrary mix of function units and registers. For example, if the target architecture can issue eight instructions per cycle, the scheduler reorders code based upon this model. Following this stage, the code is in a form which could be theoretically executed by the target architecture. Since the target architecture is not available, emulation code is inserted to model the target architecture on the emulation platform.

The code is then instrumented to gather branch direction and memory address data for the simulation. The emulation code and the Mcode for the target processor are carefully distinguished so only the Mcode is instrumented. Code generation of the final instrumented Mcode and emulation code produces an executable file. This executable serves two important purposes. First, it can be run to ensure that correct program results are generated. This verifies that all the code transformations performed are indeed correct. Second, it generates the trace information required to drive the simulator.

Simulation is performed on the target architecture's code using the memory address and branch direction data from the executable. The result is an accurate measure of the number of cycles required to execute the program on the target architecture. Due to the complexity of simulation, uniform sampling is used to reduce the simulation time for the large benchmarks [83]. For the 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 the sampling error to be less than 1% for all benchmarks.

Table 5.1 Benchmark set.

Benchmark	Description
008.espresso	truth table minimization (SPEC CINT92)
022.li	lisp interpreter (SPEC CINT92)
023.eqntott	Boolean equation minimization (SPEC CINT92)
026.compress	file compression (SPEC CINT92)
052.alvinn	neural network trainer (SPEC CFP92)
056.ear	inner ear simulation (SPEC CFP92)
072.sc	spreadsheet (SPEC CINT92)
cccp	GNU C preprocessor, version 1.35
cmp	file comparison
eqn	format math formulas for troff
grep	pattern search
lex	lexical analyzer generator
qsort	quick sort
tbl	format table for troff
wc	count characters, words and lines
yacc	parser generator

5.1.2 Benchmarks

All evaluations presented in this thesis use the set of sixteen benchmarks shown in Table 5.1. The benchmarks consist of five of the six programs from the SPEC CINT92 suite. Additionally, the two C benchmarks from the SPEC CFP92 suite, *052.alvinn*, and *056.ear*, are utilized. The remaining nine benchmarks are common Unix utility programs. A short description of each benchmark is also presented in Table 5.1. These benchmarks were chosen because of their control-intensive nature and traditional lack of exploitable ILP.

The IMPACT compiler makes extensive use of execution profile information during the compilation procedure. A description of the input files used to generate the profile information is presented in Table 5.2. For the benchmarks in which inputs were readily available, a set of 20

Table 5.2 Benchmark inputs used for profiling.

Benchmark	Description of Profiling Inputs
008.espresso	20 truth tables
022.li	4 lisp files (8-queens, 3 gabriel benchmarks)
023.eqntott	5 Boolean equations
026.compress	20 files of varying size
052.alvinn	1 set of sensory input (SPEC reference input)
056.ear	1 sound file (SPEC reference input)
072.sc	3 spread sheets (3 SPEC reference inputs)
cccp	20 C files of varying size
cmp	20 pairs of files with varying degrees of similarity
eqn	20 technical papers containing equations
grep	20 pairs of search strings and text files
lex	5 lexers for C, Lisp, Pascal, awk and pic
qsort	1 random sequence of 102400 numbers
tbl	20 technical papers containing tables
wc	20 files of varying size
yacc	10 grammars

random files were selected to provide a wide range of training data. For the other benchmarks, as many inputs as typically could be obtained were used for profiling.

The input file used for each benchmark to collect performance data is presented in Table 5.3. Overall, a large amount of care was taken to select a suitable input for measurement purposes. Also, the measured input was chosen to be different than all of the inputs on which the program was profiled to provide a more realistic evaluation. However, due to lack of available inputs, three of the benchmarks, *052.alvinn*, *056.ear*, and *072.sc*, were profiled and measured on a common input. For the SPEC benchmarks, one of the SPEC reference inputs was chosen. The one exception was for *022.li*, which used a scaled-down version of the SPEC reference input to reduce the simulation time.

Table 5.3 Input for each benchmark used to do measurements.

Benchmark	Description of Measured Input
008.espresso	one of the SPEC reference inputs (bca.in)
022.li	7-queens
023.eqntott	SPEC reference input
026.compress	SPEC reference input
052.alvinn	SPEC reference input
056.ear	SPEC reference input
072.sc	one of the SPEC reference inputs (loada2)
cccp	the file cccp.c from the GNU C Compiler Version 1.35
cmp	two copies of cccp.c
eqn	one large technical paper
grep	the string "while" from cccp.c
lex	IMPACT compiler's C lexer
qsort	one random sequence of 102400 numbers
tbl	one large technical paper
wc	the file cccp.c
yacc	the grammar in c-parse.y from the GNU C Compiler Version 1.35

To provide a more detailed understanding of the characteristics of the benchmarks investigated in this thesis, the dynamic instruction mix is presented in Table 5.4. Instructions are divided into five categories, memory load, memory store, integer ALU, floating-point ALU, and branch. Furthermore, because instructions are weighted by their execution frequency, the instruction mix is dynamic. The instruction set as previously described is an extended version of the HP PA-RISC 1.1 as generated by the IMPACT compiler. For these data, only classical optimizations are applied by the compiler. Therefore, superblock transformations have not been done.

The most interesting data in the table are the relatively high frequencies of branches. Typically, researchers report about 20-25% of dynamic instructions are branches in integer programs, whereas in this study, an average of greater than 32% of dynamic instructions are branches.

Table 5.4 Dynamic instruction mix for the benchmarks.

Benchmark	Load	Store	IALU	FALU	Branch
008.espresso	0.242	0.027	0.493	0.000	0.238
022.li	0.345	0.129	0.217	0.000	0.310
023.eqntott	0.252	0.004	0.287	0.000	0.456
026.compress	0.245	0.120	0.387	0.000	0.248
052.alvinn	0.356	0.095	0.013	0.360	0.176
056.ear	0.262	0.153	0.115	0.304	0.166
072.sc	0.304	0.053	0.309	0.008	0.327
cccp	0.235	0.067	0.195	0.000	0.504
cmp	0.422	0.193	0.193	0.000	0.193
eqn	0.300	0.135	0.262	0.000	0.303
grep	0.235	0.076	0.096	0.000	0.593
lex	0.267	0.181	0.097	0.000	0.456
qsort	0.235	0.195	0.321	0.000	0.249
tbl	0.276	0.028	0.347	0.000	0.348
wc	0.255	0.194	0.220	0.000	0.331
yacc	0.253	0.035	0.348	0.000	0.364
Average	0.280	0.105	0.244	0.042	0.329

Two extreme cases, *cccp* and *grep*, consist of more than 50% branches. This difference is due to several reasons. First, the HP PA-RISC 1.1 instruction set contains support for more powerful instructions than many RISC instruction sets, such as compare-and-branch and post-increment loads/stores. As a result, the overall number of instructions is reduced which tends to increase the relative percentage of branches. Second, the high percentage of branches reflects recent compiler optimization advances to reduce the number of dynamic instructions. Compilers, such as IMPACT, successfully target eliminating arithmetic and memory instructions with new optimization techniques. However, branch instructions are generally not eliminated. Again, the relative frequency of branches is increased. The high frequency of branches observed in these benchmarks is a strong motivation for the techniques presented in this thesis.

Table 5.5 Simulated processor architecture.

In-order issue superscalar processor with register interlocking
Uniform function units except branches (varied 1 to issue width)
Extended version of HP PA-RISC 1.1 instruction set
- Silent versions of all excepting instructions
- Predicated execution support
64 integer, 64 floating-point, 64 predicate registers
Dcache: perfect or 64K, direct mapped, blocking, 64 byte blocks, 12 cycle miss penalty, write-thru, no write allocate
Icache: perfect or 64K, direct mapped, blocking, 64 byte blocks, 12 cycle miss penalty
Bus: single transaction model with streaming support, 8 bytes/cycle bandwidth
BTB: 1K entries, direct mapped, 2-bit counter, 2 cycle misprediction penalty

It should be noted that the "average" data presented in Table 5.4 is the arithmetic mean. This convention is used throughout this dissertation for all figures and tables. Although a harmonic mean is truly correct when taking the average of ratios, the arithmetic mean was chosen because of its more simple and well-understood interpretation.

5.1.3 Processor model

The processor modeled for these experiments (target processor) is outlined in Table 5.5. The target processor is an in-order issue superscalar processor with register interlocking. The issue rate of the processor is varied from one to eight, where issue rate is the maximum number of instructions the processor can fetch and issue per cycle. The processor is assumed to have uniform function units except for the branches. Thus, there is no restriction on the type of instructions that may be issued each cycle except for the branches. The cache model used for each individual experiment is perfect unless specified as 64k entries. The assumed instruction latencies are those given in Table 5.6. These correspond to those of the HP PA-RISC 7100.

Table 5.6 Instruction latencies.

Function	Latency	Function	Latency
Int ALU	1	FP ALU	2
memory load	2	FP multiply	2
memory store	1	FP divide (SGL)	8
branch	1 / 1 slot	FP divide (DBL)	15

All experiments utilize either the restricted or general speculation model to support speculative code motion. Sentinel speculation was not used due to its scheduling complexity and simulation overhead. The interested reader is referred to [78] for an experimental evaluation of the sentinel speculation model.

5.2 Results

Performance improvement is presented in this section using a speedup calculation. Speedup is computed by dividing the total execution cycles of the base configuration by the total execution cycles of the evaluated configuration. The base configuration for all experiments presented in this section is an issue-1 processor with basic block compilation support. The same cache models are assumed for both configurations. Thus, for those experiments that utilize perfect caches, the execution cycles for the base configuration are derived using a perfect cache, whereas for those experiments that use a finite cache, the execution cycles for the base configuration are derived using the same sized finite cache.

The importance of expanding the scope of optimization/scheduling beyond basic blocks and utilizing speculative execution for ILP processors is shown in Figure 5.2. In this figure, the speedup obtained by increasing the issue rate of the target processor from two to eight is shown.

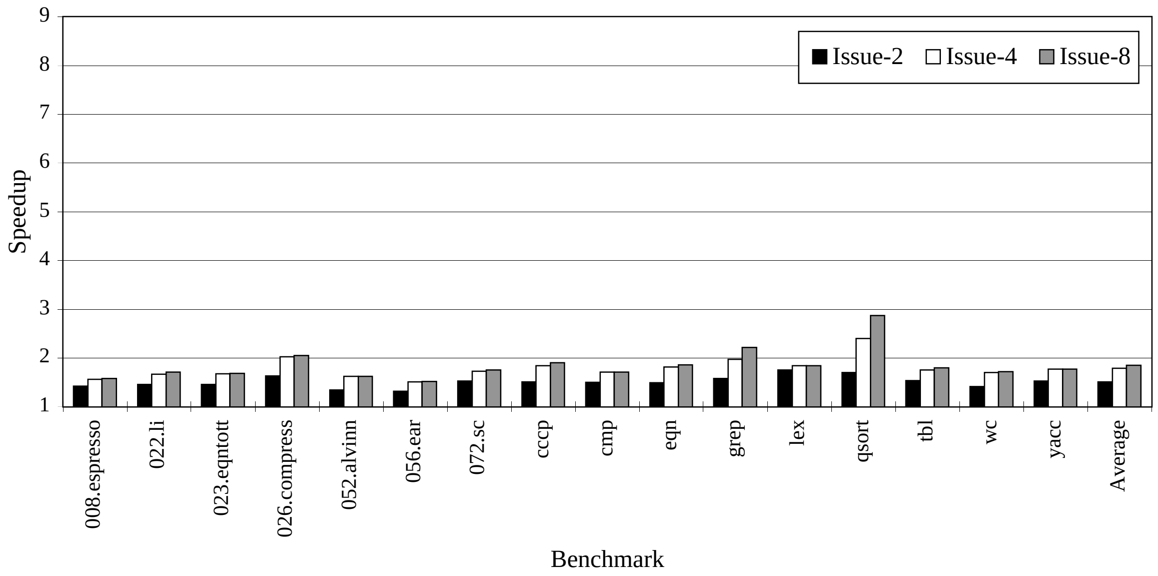


Figure 5.2 Performance comparison of three processor models using basic block compilation support.

The target processor has no restrictions on the combination of instructions that can be issued in the same cycle and perfect caches are assumed. Code is optimized and scheduled using only traditional basic block techniques. As shown, almost no performance gain is seen when the processor resources are increased. This is attributable to the lack of ILP available in the basic blocks of these benchmarks. This result could be anticipated, though. The instruction mix data presented previously in Table 5.4 indicated that approximately 32% of dynamic instructions are branches. This translates into basic blocks consisting of two to five instructions. With such a small number of instructions, the available ILP is also very small.

5.2.1 Effectiveness of speculative execution

Using the previous figure as motivation, the experiment is repeated in Figure 5.3 with extensions to overcome the performance limitations. The compilation scope is increased from basic blocks to superblocks by using the superblock compilation techniques discussed in Chapter 3.

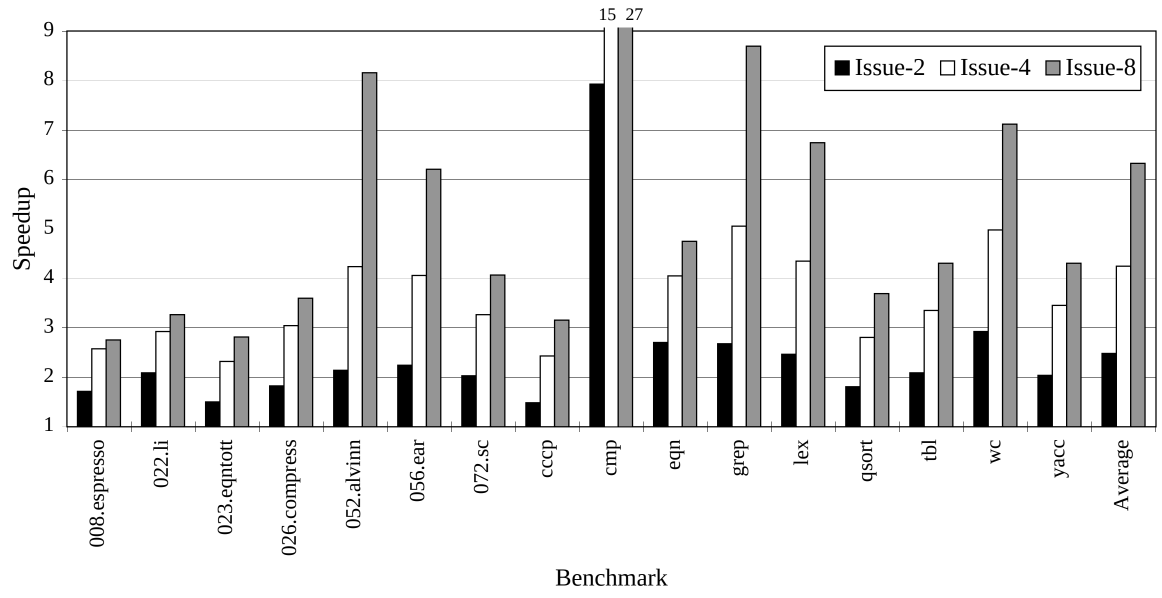


Figure 5.3 Performance comparison of three processor models supporting the general percolation model and using superblock compilation support.

Code motion support across basic block boundaries is facilitated by providing architectural support for general speculation.

Figure 5.3 shows that the performance increase is dramatic. In many cases (most notably for *cmp*), it is superlinear. These large performance improvements are due to a combination of factors. First, the efficiency of the code is increased by employing classical optimizations on the superblocks. Second, ILP is increased by performing superblock ILP optimizations. As a result of the increased ILP, large performance improvements are achieved by increasing the available resources in the target processor. On average, the speedup of an issue-8 processor with superblock compilation support is over six times that of an issue-1 processor with basic block compilation support.

The performance improvement enabled with the addition of speculative execution support is examined in more detail in Figures 5.4 - 5.6 for processor issue rates of 2, 4, and 8, respec-

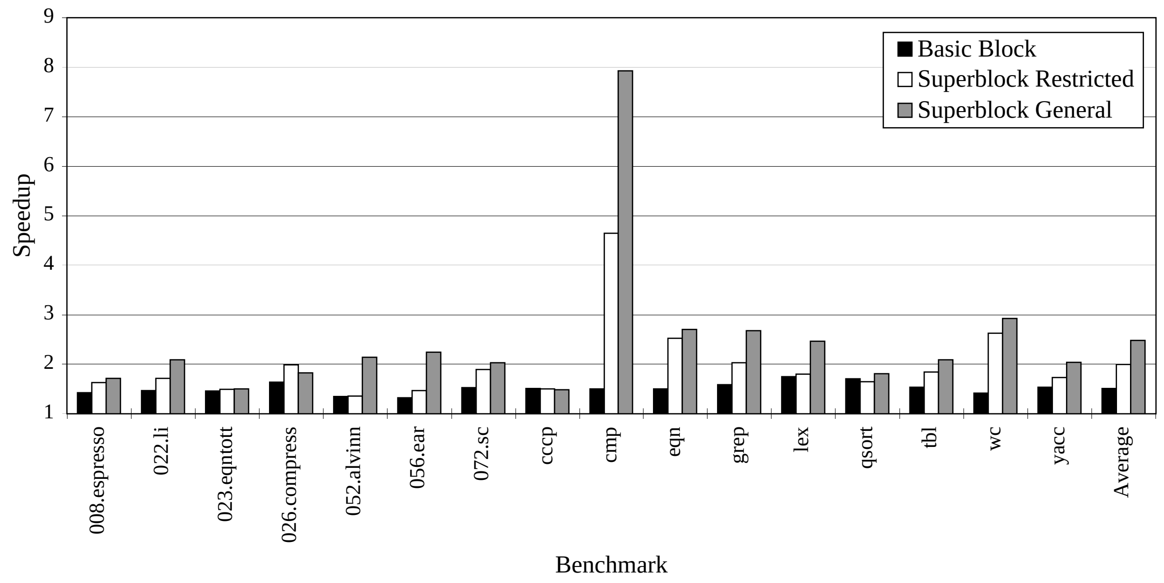


Figure 5.4 Performance comparison of an issue-2 processor with varying levels of compiler and speculation support.

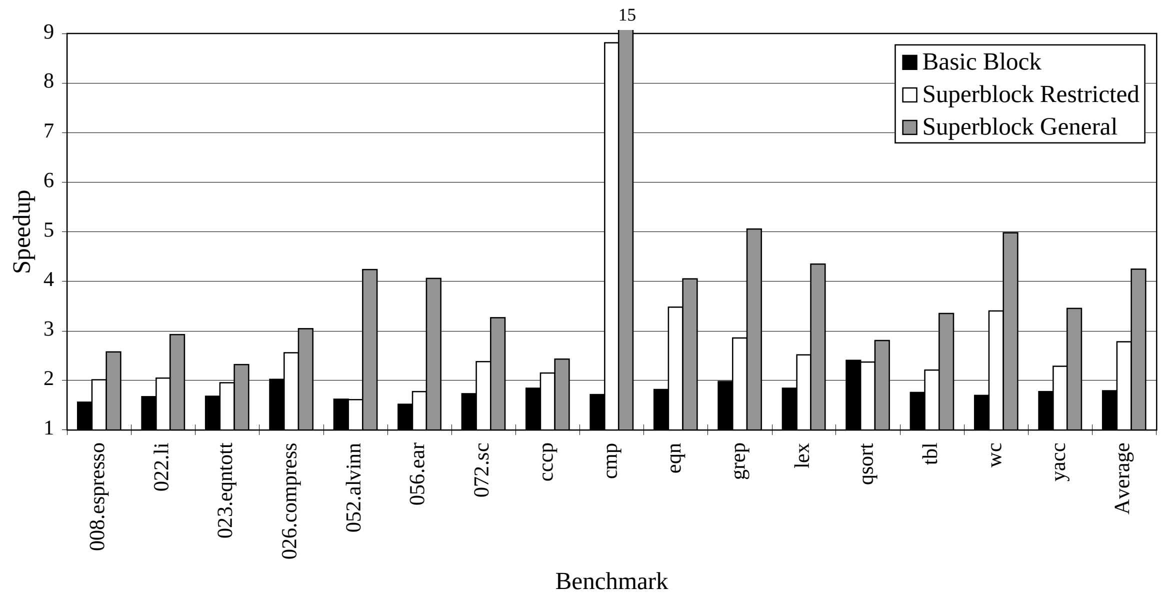


Figure 5.5 Performance comparison of an issue-4 processor with varying levels of compiler and speculation support.

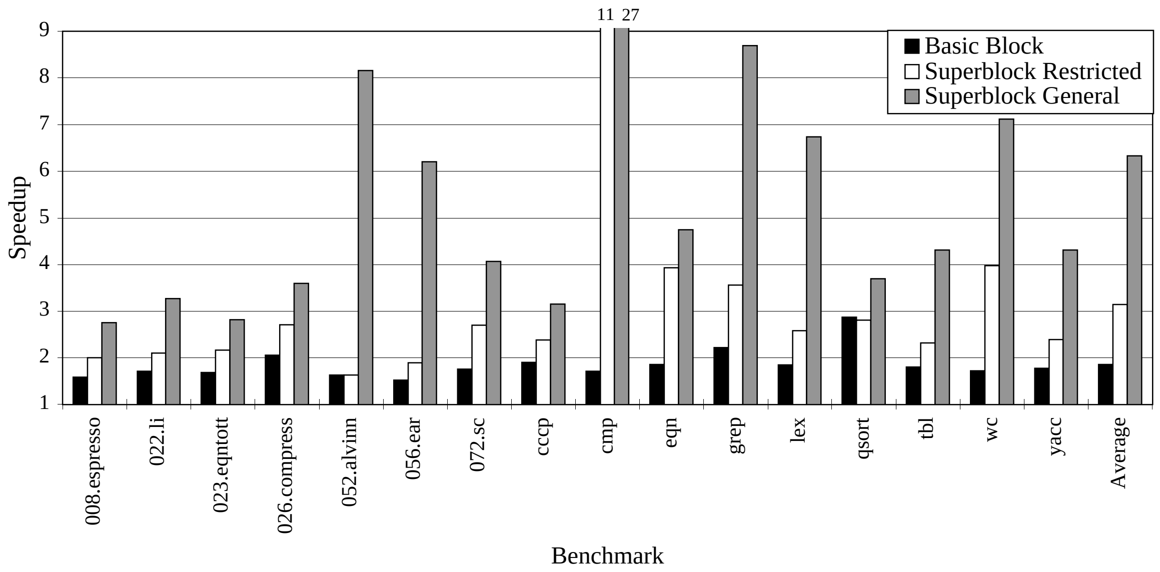


Figure 5.6 Performance comparison of an issue-8 processor with varying levels of compiler and speculation support.

tively. These experiments again assume no limitations on the combination of instructions which may be executed each cycle and utilize perfect caches. These figures isolate the effects of two important components of performance. First, comparing the basic block and the superblock restricted configurations shows the performance gain achieved by expanding the compiler optimization/scheduling scope from basic blocks to superblocks. Second, comparing the superblock restricted and the superblock general configurations shows the additional performance gain provided by general speculation support in superblocks.

For an issue-2 processor (Figure 5.4), only a small performance gain is achieved with superblock compilation support. A large percentage of this gain is also achieved with only restricted speculation. This trend indicates the compiler is generally effective utilizing the processor resources without general speculation for narrow issue processors. With an issue-4 processor (Figure 5.5), more substantial gains are achieved. Superblocks with restricted speculation expose new optimization and scheduling opportunities to improve performance. However, the full effects of the optimizations are not realized without support for general speculation. The majority of the performance gain is only achieved with general speculation support. This is especially true for the superblock dependence removing optimizations. These optimizations eliminate data dependences between instructions, but if the scheduler cannot move the instructions above branches, the scheduler cannot take full advantage of the potential ILP.

The trends with the issue-4 processor generally continue and become more magnified for the issue-8 processor in Figure 5.6. For issue-8, on average only half the maximal performance level (3x speedup) is achieved with superblocks and restricted speculation. By adding general speculation support, the average performance level is doubled (6x speedup). Overall, the importance of general speculation is shown to increase as the issue rate of the processor increases.

The code motion performed by the compiler with speculative execution support can significantly affect the dynamic number of instructions executed by the target processor. Intuitively, one would expect the number of dynamic instructions to increase since speculative code motion increases the execution frequency of instructions. The effect of general speculation on the dynamic instruction count for an issue-8 processor is presented in Table 5.7. The table contains the number of dynamic instructions executed for three configurations, basic block, superblock restricted, and superblock general. In addition, the ratios of the instruction count for superblock restricted and superblock general with respect to the basic block are provided in parenthesis.

The table surprisingly indicates a relatively small increase in the number of dynamic instructions for most benchmarks. For some, such as *cmp* and *grep*, the number of dynamic instructions is actually less than with speculative execution. The reason for this trend is the

Table 5.7 Effect of speculation on the dynamic instruction count for an issue-8 processor.

Benchmark	Basic Block	Superblock Restricted	Superblock General
008.espresso	379M	437M (1.16)	484M (1.28)
022.li	30M	29M (0.97)	31M (1.05)
023.eqntott	767M	845M (1.10)	1029M (1.34)
026.compress	71M	82M (1.16)	89M (1.25)
052.alvinn	2669M	3568M (1.34)	3573M (1.34)
056.ear	11366M	11077M (0.97)	11263M (0.99)
072.sc	93M	105M (1.12)	109M (1.17)
cccp	2376K	3522K (1.48)	3678K (1.55)
cmp	2751K	1539K (0.56)	932K (0.34)
eqn	41M	46M (1.12)	45M (1.09)
grep	1452K	1200K (0.83)	1282K (0.88)
lex	32M	33M (1.04)	35M (1.11)
qsort	40M	45M (1.11)	48M (1.18)
tbl	2529K	2391K (0.95)	2490K (0.98)
wc	1933K	1585K (0.82)	1491K (0.77)
yacc	36M	37M (1.02)	43M (1.17)
Average	-	- (1.05)	- (1.09)

two competing effects. On the one end, additional optimization opportunities are enabled with speculative execution, which tend to reduce the instruction count. These optimizations include loop invariant code removal and global variable migration. For example, an invariant load instruction in a loop that is conditionally executed may not be safely moved out of a loop because it may cause a spurious exception. However, with general speculation, such a load can be safely moved out of the loop by using a silent version of the load. The opposing effect is the increase in dynamic instructions caused by speculating an instruction above one or more branches. By speculating the instruction, it executes regardless of the direction the branch takes. Speculative code motion therefore increases the instruction count. Overall, the net of these competing effects is the final reported dynamic instruction count.

Table 5.8 Speculative load characteristics for an issue-8 processor.

Benchmark	Total Loads	Speculative Loads
008.espresso	95M	63M (0.67)
022.li	10M	6681K (0.64)
023.eqntott	353M	287M (0.81)
026.compress	13M	8644K (0.64)
052.alvinn	950M	886M (0.93)
056.ear	2384M	2047M (0.86)
072.sc	32M	22M (0.69)
cccp	551K	356K (0.65)
cmp	271K	267K (0.98)
eqn	6924K	3640K (0.53)
grep	319K	300K (0.94)
lex	9739K	8492K (0.87)
qsort	13M	8036K (0.60)
tbl	1894K	1136K (0.60)
wc	171K	136K (0.80)
yacc	11M	9547K (0.83)
Average	-	- (0.75)

The most important class of instructions to speculate is the loads because they typically have long latency and begin dependence chains. The use of speculative loads for an issue-8 processor is presented in Table 5.8. The table contains the total dynamic loads in each benchmark and the total dynamic loads which are speculative. The number in parenthesis is the ratio of speculative loads to total loads. As shown in the table, an extremely large fraction of the loads is speculative. Values range from a low of 60% for *qsort* and *tbl* to a high of 98% for *cmp*. Clearly, the compiler takes strong advantage of speculative loads to achieve a compact schedule.

5.2.2 Effectiveness of superblock optimizations

The individual performance contributions of superblock formation and superblock ILP optimizations are broken down in Figure 5.7. For both superblock configurations, the processor is assumed to support general speculation, has no limitations on the combination of instructions that may be issued each cycle, and has perfect caches. From the figure, the dominant effect that superblock ILP optimizations have on the overall performance is shown. Superblock formation alone generally yields only modest performance gains. However, with the addition of superblock ILP optimizations, large performance gains are achieved. Several of the most distinct examples of this behavior occur for *056.ear*, *lex*, and *yacc*. This trend could be anticipated, though. Superblock formation only combines basic blocks into superblocks increasing the potential for instruction overlap by enlarging the scope. Overlap is still significantly limited by data dependences and the inability to exploit loop-level ILP, whereas the superblock ILP optimizations, such as loop unrolling and induction variable expansion, aggressively transform the superblocks to increase ILP in loops and straight-line code. As a result, the scheduler has many more opportunities to reorder instructions and achieve a compact schedule.

For several of the benchmarks, the general trend is not observed. For example, for *022.li* and *eqn*, the majority of the overall performance is achieved with superblock formation. For these benchmarks, the superblock optimizations are relatively ineffective at increasing ILP. This can be attributed primarily to memory disambiguation difficulties and inherent data dependences which could not be broken.

The major side effect of superblock formation and superblock optimization is the code expansion that is incurred. The code expansion resulting from superblock techniques is presented in Figure 5.8. The size has been normalized with respect to the static size of the basic

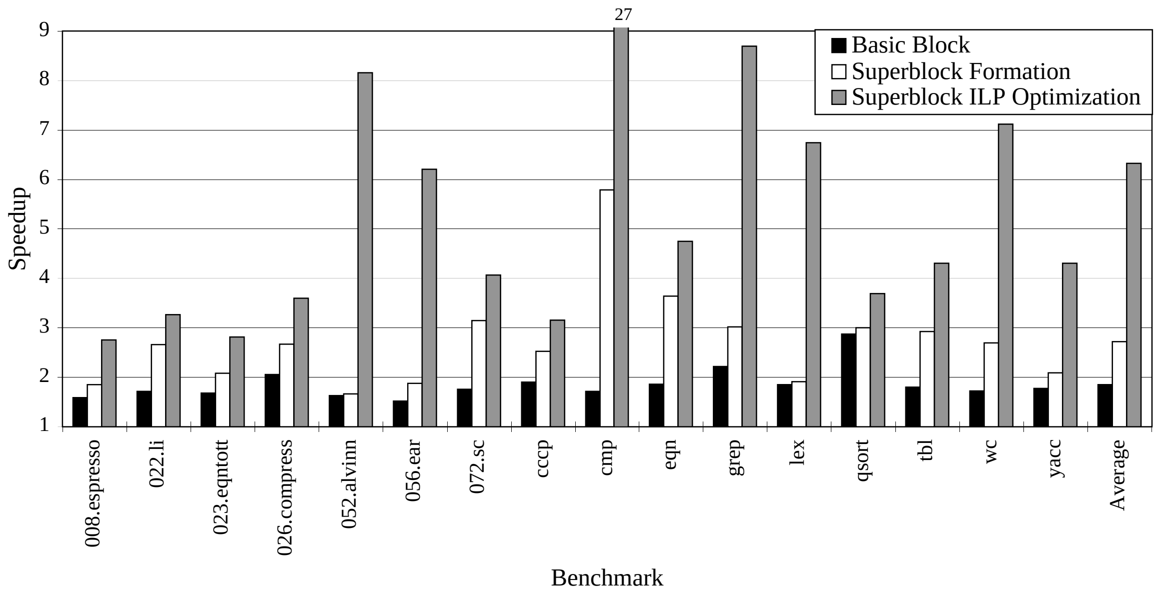


Figure 5.7 Effectiveness of superblock formation and optimization for an issue-8 processor.

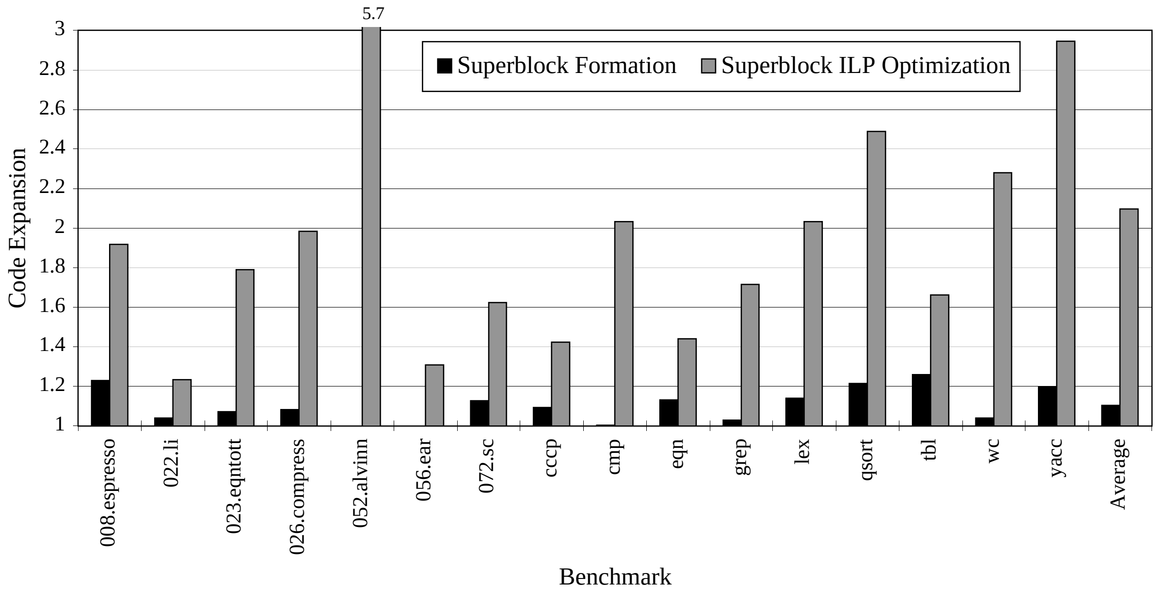


Figure 5.8 Code expansion of superblock formation and optimization.

block code. From the figure, superblock formation generally only produces modest code size increases. On average, an approximately 10% increase is observed. The largest increases occur for *008.espresso*, *qsort*, and *tbl*, each experiencing slightly more than a 20% growth.

In contrast to superblock formation, a relatively large increase in static code size occurs with superblock ILP optimizations. The average measured increase is about 2.1 times. The large increases are mainly due to loop unrolling. For an issue-8 processor, loops are generally unrolled 4 to 16 times depending on the static and profile characteristics of the loop. Branch target expansion also tends to significantly increase code size. The optimizations do utilize profile information to control code expansion by expanding only important sections of the code. However, the overall size increase is still significant. It should be noted that the large code expansion values reported are a bit misleading due to the small size of some of the benchmarks. Four of the five benchmarks experiencing the largest code expansions, namely *052.alvinn*, *cmp*, *qsort*, and *wc*, are all less than 300 lines of C source code. Therefore, increasing the code size of these even five times is not that serious since the overall size is still extremely manageable.

5.2.3 Cache effects

Up to this point in the evaluation of speculative execution and superblock techniques, a perfect cache model has been used. In this section, this restriction is removed to study the effects of finite cache models on performance. Additionally, the effect of superblock optimizations and speculative code motion on the instruction and data caches is examined.

The experiment presented earlier with perfect caches in Figure 5.3 is repeated in Figure 5.9 with 64K instruction and data caches. Note that the base configuration also uses 64K caches to obtain the cycle count total for this experiment. The target processor has no limitations on

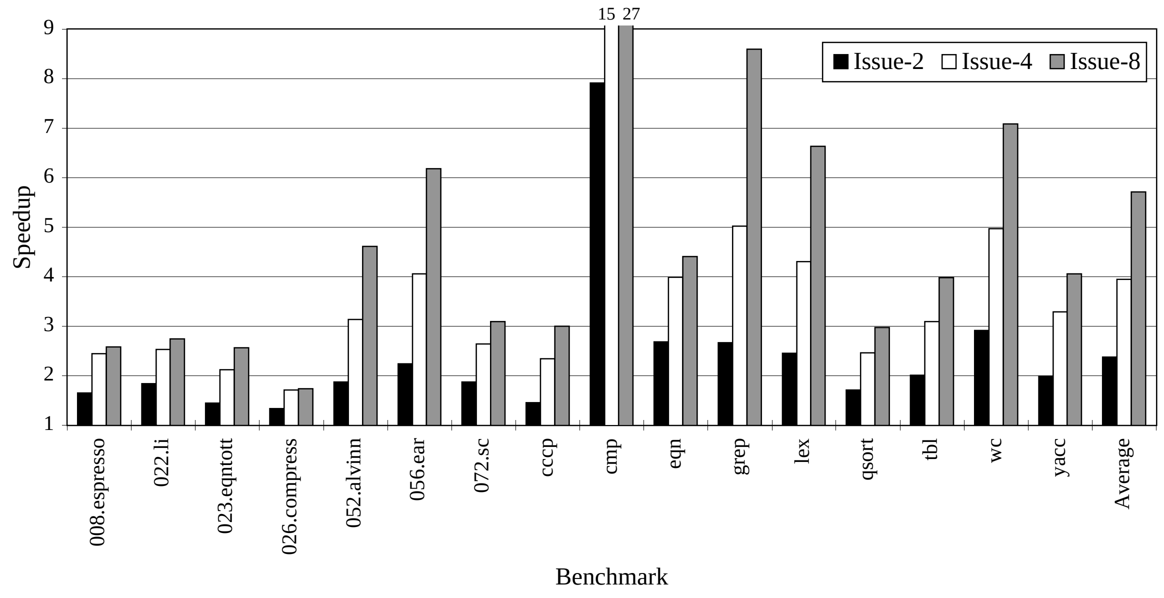


Figure 5.9 Performance comparison of three processor models with 64K caches supporting general speculation and using superblock compilation support.

the combination of instructions that can be issued each cycle and supports general speculation. Furthermore, superblock compilation techniques are used for all the evaluated configurations.

Several important points can be made by contrasting the perfect and finite cache experiments. First, the general trends visible in the perfect cache experiment are maintained in the finite cache experiment. The compiler takes advantage of the ILP exposed in the code to achieve continual performance improvements as the available processor resources are increased. However, the speedup values reported are reduced for all issue rates. This result is expected though, because as one increases both the execution cycles of the base configuration as well as the evaluated configuration, their ratio reduces.

A second point is that the finite cache has a larger negative effect on performance as the issue rate of the target processor is increased. This trend occurs because as the issue rate of the processor increases, the total execution cycles decrease. Assuming the number of cycles

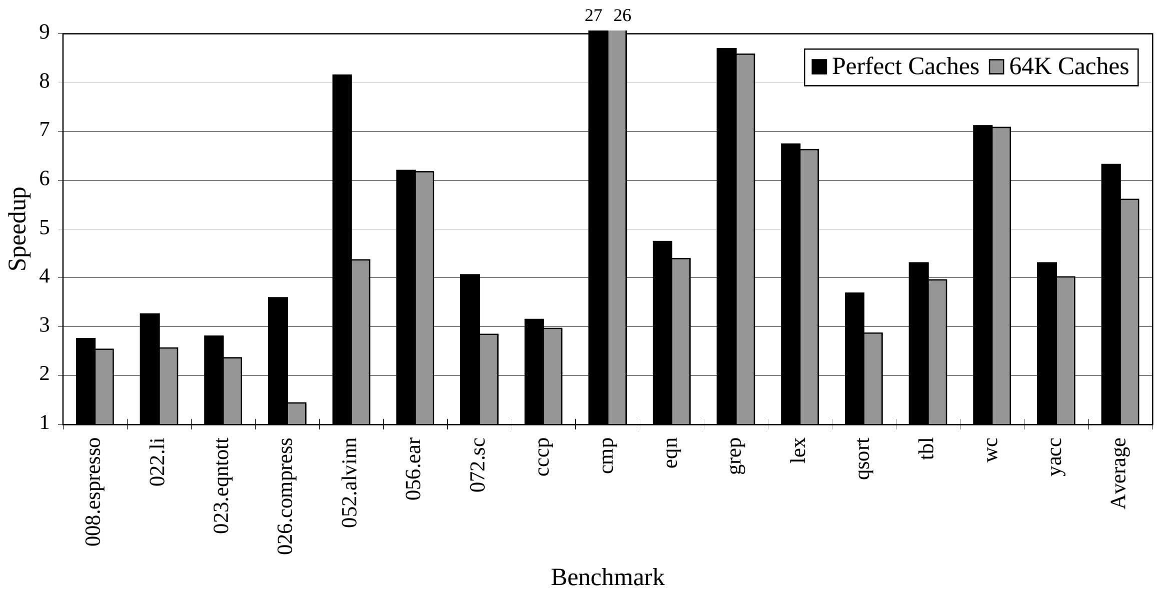


Figure 5.10 Performance comparison of perfect cache model and the 64K cache model for an issue-8 processor.

stalled for cache misses remains relatively constant, a larger fraction of the total cycles is spent stalling for cache misses. Additionally, as the issue rate of the processor increases, each stalled cycle represents a loss of a larger number of potentially executed instructions. For example, an issue-8 processor loses eight instructions for every stalled cycle, whereas an issue-2 processor only loses two instructions. As a result, the cost of each stalled cycle grows in proportion to increases in the available processor resources.

The actual amount of performance lost due to finite caches is shown more clearly in Figure 5.10. For this experiment, the execution cycles for both perfect and finite caches are normalized with respect to the base configuration with perfect caches. Again, superblock compilation techniques are used for both experimental configurations. For many of the benchmarks, only a small performance loss is incurred with the finite cache. The instruction and data working sets for most of these benchmarks have little difficulty fitting into a 64K cache.

However, for several of the benchmarks, the cost is large. For example in *026.compress*, performance drops approximately 2.5 times with a finite cache, which is due to a high frequency of data cache misses (24.8% miss rate). The working set of *026.compress* is extremely large and causes thrashing in a 64K direct-mapped data cache. Another benchmark, *052.alvinn*, has a relatively low miss rate (3.5%), but the combination of the extremely parallel benchmark and with a large fraction of loads (35.6% from Table 5.4) causes the large performance loss. Noticeable performance losses are also observed for *072.sc* and *qsort* primarily because of data cache stalls.

The effects of speculative code motion on the caches are examined in more detail in Table 5.9. In the table, the number of instruction and data cache misses is reported for restricted and general speculation. The ratio of each data entry for general speculation with respect to restricted speculation is shown in parenthesis. To magnify the effects, a 4K instruction cache and perfect data cache are used for the instruction cache evaluation. In correspondence, a perfect instruction cache and a 4K data cache are used for the data cache evaluation. For both experiments, an issue-8 processor is assumed.

From Table 5.9, the effect of general speculation on instruction cache misses varies across the benchmarks. The predominant effect is an increase in the number of instruction cache misses caused by speculating instructions above branches. Speculative code motion causes instructions to be executed more frequently. As a result, the instruction working set grows, causing an increase in the number of cache misses. In contrast, seven of the benchmarks show the counterintuitive result a decrease in instruction cache misses with general speculation. This behavior is primarily attributable to small differences in the code layout resulting from additional superblock optimizations enabled with general speculation. The varied code layout

Table 5.9 Effect of speculation on the instruction and data caches for an issue-8 processor.

	Icache Misses		Dcache Misses	
Benchmark	Restricted	General	Restricted	General
008.espresso	2523K	2410K (0.96)	7131K	7730K (1.08)
022.li	1273K	1074K (0.84)	900K	986K (1.10)
023.eqntott	641K	655K (1.02)	12M	12M (0.96)
026.compress	1470K	1735K (1.18)	3956K	4973K (1.26)
052.alvinn	334K	322K (0.96)	78M	71M (0.92)
056.ear	33M	27M (0.80)	110M	104M (0.95)
072.sc	1675K	1400K (0.84)	2740K	3424K (1.25)
cccp	41K	45K (1.10)	29K	28K (1.00)
cmp	26	32 (1.23)	85K	32K (0.38)
eqn	2431K	2238K (0.92)	775K	742K (0.96)
grep	70	73 (1.04)	2966	3061 (1.03)
lex	77K	107K (1.38)	288K	316K (1.10)
qsort	289	254 (0.88)	446K	410K (0.92)
tbl	54K	55K (1.02)	33K	42K (1.26)
wc	26	33 (1.27)	3223	2536 (0.79)
yacc	342K	426K (1.24)	790K	1060K (1.34)
Average	-	- (1.04)	-	- (1.02)

causes fewer collisions in the directed mapped cache. Additionally, fewer collisions occur in the BTB introducing fewer branch prediction misses. Fewer BTB misses indirectly decrease the instruction cache misses by reducing the frequency at which instructions are fetched down the wrong branch path. These effects magnified by the small instruction cache result in the observed reduction in instruction cache misses.

The effects of general speculation on the data cache misses are also shown in Table 5.9. The predominant trend is an increase in misses with general speculation. Large increases are observed for *026.compress*, *072.sc*, *tbl*, and *yacc*. This is a direct result of speculating load instructions during scheduling. With general speculation, the compiler can freely speculate loads to achieve a compact schedule. These speculative loads introduce new cache misses which did not exist with restricted speculation. On the other hand, two of the benchmarks, *cmp* and *wc*, show a large reduction in cache accesses and misses with general speculation. This behavior is due to the substantial number of superblock optimizations enabled with general speculation. A large fraction of the loads and stores are eliminated from important loops with loop invariant code removal and global variable migration. These optimizations are not possible with restricted speculation because the loads are conditionally executed and may introduce spurious exceptions. With fewer loads and stores in the important loops, the number of data cache misses is significantly reduced.

The effects of superblock ILP optimizations on the instruction and data caches are presented in Table 5.10. The same experimental parameters and data format as the previous experiment (Table 5.9) are used. From the table, large increases in the number of instruction cache misses are observed. This is primarily due to the set of superblock enlarging optimizations that are applied, including loop unrolling and branch target expansion.

The most notable increases occur for *026.compress* and *052.alvinn*. Both of these benchmarks are relatively small and their working sets have little difficulty fitting into 4K instruction caches with only superblock formation applied. However, after superblock ILP optimizations, their instruction working sets exceed the 4K capacity of the cache, and large increases in misses is the result. It is interesting to note that loop unrolling is the major source of increase for *052.alvinn*, which has most of its inner loops unrolled 16 times. Without loop unrolling, all of the inner loops can simultaneously reside in the instruction cache, but afterwards the loops interfere with one another causing a large number of misses. In comparison, branch target expansion is the optimization which causes the large increase in instruction cache misses for *026.compress*. There are a large number of important superblocks each of which is judiciously

Table 5.10 Effect of superblock ILP optimization on the instruction and data caches for an issue-8 processor.

	Icache Misses		Dcache Misses	
Benchmark	SB Formation	SB Optimization	SB Formation	SB Optimization
008.espresso	1465K	2410K (1.65)	7582K	7730K (1.02)
022.li	1104K	1074K (0.97)	954K	986K (1.03)
023.eqntott	699K	655K (0.94)	11M	12M (1.02)
026.compress	1169	1735K (1483.02)	3787K	4973K (1.31)
052.alvinn	2900	322K (111.14)	81M	71M (0.88)
056.ear	2512K	27M (10.76)	189M	104M (0.55)
072.sc	574K	1400K (2.44)	2880K	3424K (1.19)
cccp	9807	45K (4.66)	27K	28K (1.05)
cmp	10	32 (3.20)	89K	32K (0.36)
eqn	1094K	2238K (2.05)	811K	742K (0.91)
grep	23	73 (3.17)	2899	3061 (1.06)
lex	16K	107K (6.69)	302K	316K (1.05)
qsort	106	254 (2.40)	396K	410K (1.04)
tbl	44K	55K (1.25)	35K	42K (1.22)
wc	14	33 (2.36)	8770	2536 (0.29)
yacc	62K	426K (6.78)	799K	1060K (1.33)
Average	-	- (102.72)	-	- (0.96)

enlarged with branch target expansion. Afterwards, interference occurs among the superblocks causing instruction cache misses to increase. One should note that the average reported in Table 5.10 is significantly skewed by the large increases for *026.compress* and *052.alvinn*.

The effects of superblock ILP optimizations on the data cache are also shown in Table 5.10. For most of the benchmarks, an increase in data cache misses is obtained with superblock ILP optimization. This is attributed to the increase in speculation opportunities created by the optimizations. Loop unrolling combined with register renaming, induction variable expansion, and accumulator variable expansion enable loop iterations to be tightly overlapped using speculation. As previously mentioned, speculation of loads increases their execution frequency which leads to increases in data cache misses. Without these optimizations, only limited speculation opportunities exist due to the large number of data and control dependences. Thus, the number of speculative loads is small without superblock ILP optimizations.

Several of the benchmarks show a drastic reduction in data cache misses with superblock ILP optimizations. Two examples of this behavior occur for *cmp* and *wc*. As previously mentioned, this is mostly due to the increased opportunities for optimizations that remove loads and stores from loops, namely loop invariant code removal and global variable migration. These optimizations are not applicable in the traditional sense because hazardous subroutine calls exist in the loop bodies. As a result, the optimizer must be conservative and not remove any memory instructions. With superblock formation, the hazardous instructions are excluded from the superblock thereby exposing new optimization opportunities. The optimized superblock loop has the majority of the loads and stores eliminated for these two benchmarks, resulting in large decreases in data cache misses.

The interested reader is referred to [32] for more evaluations on the effects of speculation on instruction and data caches. Also, for more evaluation of code expanding optimizations on the instruction cache, the reader is referred to [84].

5.2.4 Limitations of speculative execution in superblocks

This section has illustrated the performance improvement potential of superblocks, speculative execution in superblocks, and superblock ILP optimizations. Over the traditional basic block compilation techniques, large performance improvements are observed. However, there are several factors which motivate enlarging the compilation scope beyond superblocks to exploit ILP. These include a significant amount of under-utilized resources in wide issue proces-

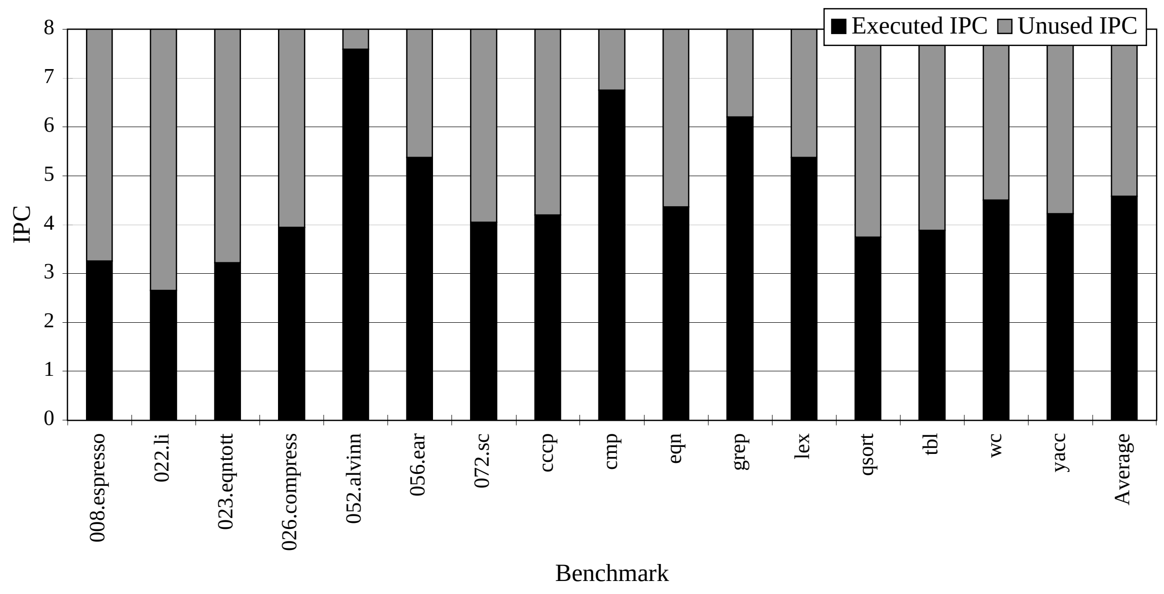


Figure 5.11 Average executed and unused instructions per cycle for an issue-8 processor using superblock compilation techniques.

sors. Also, the superblock performance relies heavily on support to execute a large number of branches each cycle. This section examines these limitations in more detail.

The average fraction of executed and unused instructions per cycle (IPC) for an issue-8 processor achieved with superblock compilation techniques is presented in Figure 5.11. The experimental processor is assumed to have perfect caches and no restrictions on the combination of instructions that may be issued in a cycle. The data in the figure show that across the benchmarks there is a substantial fraction of idle processor resources. Only three benchmarks, *052.alvinn*, *cmp*, and *grep*, sustain greater than six IPC. Six of the benchmarks utilize less than half the available resources. It should be noted that the executed IPC in the figure consists of any issued instructions, both useful and useless. The figure clearly shows that there is a substantial opportunity to increase performance beyond what is achieved with superblocks and speculation alone.

Table 5.11 Superblock characteristics.

Benchmark	SB size	SB Completion Ratio				
		1.00	≥ 0.90	≥ 0.70	≥ 0.50	≥ 0.30
008.espresso	34.2	0.34	0.39	0.50	0.61	0.81
022.li	18.2	0.59	0.63	0.72	0.75	0.83
023.eqntott	19.3	0.46	0.46	0.47	0.53	0.63
026.compress	30.9	0.44	0.45	0.52	0.57	0.75
052.alvinn	88.1	0.96	0.97	0.98	0.98	0.99
056.ear	46.1	0.70	0.70	0.74	0.77	0.81
072.sc	28.5	0.40	0.40	0.43	0.62	0.82
cccp	21.1	0.62	0.62	0.65	0.72	0.89
cmp	57.8	0.78	0.78	0.83	0.87	0.92
eqn	29.6	0.59	0.59	0.62	0.67	0.76
grep	53.6	0.68	0.70	0.74	0.81	0.87
lex	37.6	0.63	0.64	0.66	0.75	0.82
qsort	19.8	0.58	0.58	0.58	0.66	0.77
tbl	16.1	0.74	0.76	0.80	0.85	0.92
wc	24.6	0.44	0.57	0.71	0.79	0.95
yacc	26.7	0.48	0.49	0.54	0.67	0.86
Average	34.5	0.59	0.61	0.66	0.73	0.84

In order to more deeply understand the source of the idle resources, the characteristics of the superblocks are examined in more detail in Table 5.11. The first column of data contains the average number of instructions in each superblock weighted by the execution frequency of the superblock. These data show the average size of the superblocks presented to the scheduler for each benchmark. The table shows that the average superblock size is rather small for most of the benchmarks. Considering that the scheduler is trying to produce a compact schedule for an issue-8 processor, 20-40 instructions are typically insufficient to fully utilize the resources due to moderate-length dependence chains.

An important behavior to note when comparing the superblock size data and the executed IPC data in Figure 5.11 is the correlation between large superblock size and high executed IPC values. The three benchmarks with the largest average size, *052.alvinn*, *cmp*, and *grep*, also have the largest executed IPC. Correspondingly, all of the benchmarks with an average size of less than 20 instructions, execute less than four IPC. This illustrates the need to expand the scheduling scope beyond superblocks in order to examine a larger number of instructions to find sufficient ILP.

The remaining data in Table 5.11 show the weighted average completion ratio of superblocks. The completion ratio is defined as the percentage of time a specified fraction of the superblock is executed. For example, the "$\geq$0.90" column for *008.espresso* indicates that 39% of the time 90% or more of the instructions in the superblocks are executed. Looking at the data inversely, 61% of the time the superblock is exited before 90% of the instructions are executed. From the table, on average more than 50% of the superblock is only executed 73% of the time. This indicates that superblocks are exited prematurely through a side exit a large fraction of the time. This is particularly undesirable because the superblock is optimized and scheduled assuming the entire superblock will execute. Much of the potential performance gained through speculation is lost when a superblock side exit is taken. All speculative instructions which originated below the taken side exit are wasted instructions. The ideal behavior is displayed by *052.alvinn*: large superblocks that predominantly execute to completion.

Superblocks are inherently limited by the restriction that ILP may only be exploited along a single path of control. For many control-intensive programs, no single highly dominant path of execution exists. The data in Table 5.11 shows the need to generalize the superblock techniques to overlap the execution of multiple paths. By exploiting ILP along multiple paths, more opportunities for instruction overlap are created. Additionally, execution can be effectively maintained within the compilation structure preventing the early exit problem seen with superblocks.

An assumption used throughout the experiments up to this point is that the processor can execute any combination of instructions each cycle, including branches. Thus for an issue-8 processor, up to 8 branches can be issued simultaneously. This assumption though is not likely to be met with future ILP processors. Most next generation superscalar processors, such as HP PA-8000, Intel P6, and Sun UltraSPARC, are issue-4 and can process a maximum of one branch each cycle [85],[86]. The Multiflow Trace series machine is an exception to this rule; it could process up to four branches each cycle [73]. However, one can expect the number of branches executed each cycle to remain small due to design and implementation difficulties caused by executing multiple branches per cycle.

The performance of the superblock techniques utilizing general speculation is presented in Figure 5.12 for an issue-8 processor as the maximum number of branches allowed each cycle is varied from eight to one. The figure shows a substantial drop-off across all benchmarks. On average, performance is cut by approximately a factor of two when the number of branches is reduced from eight to one. The benchmarks showing the largest effects are *cmp* and *grep*, where the performance losses are factors of 4 and 3.5, respectively. Much of the large performance gains that were achieved for an issue-8 processor are lost when the processor can only execute 1 or 2 branches each cycle. This behavior occurs because with superblock scheduling, branches tend to get clustered at the bottom of superblocks. The branches are typically data dependent on one or more memory and arithmetic instructions. As a result, the computation instructions are pushed near the top of the superblock and overlapped. The branches are pushed downwards until a time at which their source operands are available. The net result is a long chain of branches

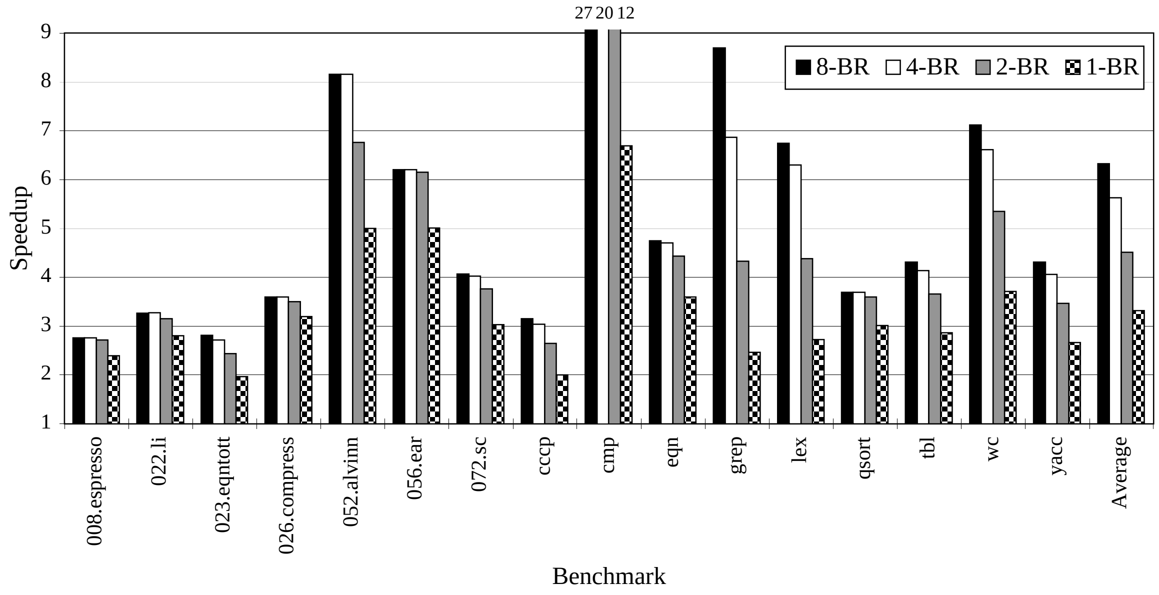


Figure 5.12 Effect of reducing the maximum number of branches executed per cycle for an issue-8 processor.

at the end of each superblock that must be processed. As the branch execution resources are constrained, the schedule lengths of the superblocks are expanded and performance loss results.

The data in Figure 5.12 show that branch resources quickly become the performance bottleneck as their availability is decreased. This provides compelling motivation to reduce the number of branches in the instruction stream. If a significant portion of the branch instructions could be eliminated, the branch resource bottleneck is likely to disappear.

The branch problem is compounded by the tendency to increase the fraction of branches in the instruction stream as more advanced optimization techniques are applied. Advanced optimizations, such as superblock classical optimizations, tend to focus on eliminating memory and ALU instructions. However, few optimizations address the reductions of the number of branches. The net result is an increase in the percentage of instructions that are branches. This behavior is illustrated by comparing the dynamic instruction mixes with only classical

Table 5.12 Dynamic instruction mix after superblock optimizations.

Benchmark	Load	Store	IALU	FALU	Branch
008.espresso	0.228	0.027	0.520	0.000	0.225
022.li	0.323	0.137	0.226	0.000	0.313
023.eqntott	0.223	0.004	0.402	0.000	0.371
026.compress	0.201	0.098	0.450	0.000	0.252
052.alvinn	0.349	0.093	0.028	0.357	0.173
056.ear	0.209	0.171	0.121	0.313	0.186
072.sc	0.255	0.042	0.383	0.011	0.310
cccp	0.189	0.055	0.328	0.000	0.427
cmp	0.265	0.027	0.118	0.000	0.589
eqn	0.197	0.119	0.319	0.000	0.365
grep	0.202	0.092	0.146	0.000	0.560
lex	0.253	0.179	0.119	0.000	0.450
qsort	0.204	0.186	0.379	0.000	0.230
tbl	0.311	0.033	0.279	0.000	0.377
wc	0.123	0.131	0.331	0.000	0.415
yacc	0.240	0.035	0.349	0.000	0.376
Average	0.236	0.089	0.281	0.043	0.351

optimizations and with additional superblock optimizations. The dynamic instruction mix with classical optimizations was presented earlier in Table 5.4, and the dynamic instruction mix after superblock optimizations is shown in Table 5.12. Again, instructions are broken down into five categories, memory load, memory store, integer ALU, floating-point ALU, and branch. It should be noted that the effects of all superblock optimizations (classical and ILP) are included in Table 5.12. Therefore, the code is not primarily optimized for redundancy elimination. The focus of many of the transformations is to remove dependences to increase ILP in the superblocks.

A comparison of the data in Tables 5.4 and 5.12 shows that, on average, superblock optimizations significantly reduce the fraction of loads and stores, while increasing the fraction of ALU and branch instructions. For instance, the average fraction of loads drops from 0.280 to

0.236, whereas the average fraction of branches rises from 0.329 to 0.351. The most notable increase in the fraction of branches occurs for *cmp*, which increases from 0.193 to 0.589. Recall from previous discussions that a large fraction of the loads and stores are removed from the inner loop of *cmp* after superblock formation by applying loop invariant code removal and global variable migration. With the removal of most of the loads and stores, the fraction of branches in the loop rises dramatically. Significant increases in the fraction of branches also occur for *eqn* and *wc* for similar reasons.

Not all the benchmarks show increases in the fraction of branches. One notable exception occurs for *023.eqntott*, in which the fraction of branches is reduced from 0.456 to 0.371. For this benchmark, superblock formation exposes an additional opportunity for constant propagation in the most frequently executed inner loop. Following constant propagation, one of the branches is left with compile-time constants for source operands, allowing it to be evaluated and eliminated. As a result, a branch is eliminated from each iteration of the important loop, substantially reducing the fraction of branches. Despite some optimization opportunities such as this, a large fraction of the dynamic instructions are branches for all the benchmarks. This indicates that more widely applicable techniques are needed to eliminate branches from the instruction stream.

In summary, speculative execution in superblocks alone is limited by several factors. First, there are a large number of resources which cannot be filled by the superblock techniques alone. Thus, there is a wide range of potential performance improvement beyond that achieved with superblock techniques. The limited ILP in superblocks can be attributed to superblocks typically not being large enough and having frequently taken side exits. These characteristics motivate expanding the compiler scope from a single path of execution to overlapping multiple execution paths. The second factor is the large performance loss incurred when the number of branches that a wide-issue processor can execute are reduced. The instruction stream contains a high fraction of branches, and when the number of branches which may be executed is limited, the branch resource bottleneck is exposed. This factor motivates the need for generalized techniques to eliminate branches from the instruction stream to overcome the resource bottlenecks.

These limitations lead directly into the remainder of this thesis, which explores the use of predicated execution. Predicated execution support enables the compiler to eliminate branches from the instruction stream. Also, predication provides an efficient mechanism for the compiler to exploit ILP along multiple execution paths. The next two chapters discuss the architecture and compiler support for predicated execution that is explored in this dissertation.

CHAPTER 6

PREDICATED EXECUTION

As discussed in the previous chapter, utilizing speculative execution alone to extract ILP in the presence of branches has performance limitations. The fundamental limitation is that speculation eliminates dependences between instructions and branches, but does not remove the branches themselves. To overcome this drawback, predicated execution is investigated. Predicated or guarded execution enables a compiler to eliminate branches from the instruction stream. As a result, many of the difficulties introduced by branches can be eliminated. This chapter addresses the architectural support required to accomplish predicated execution. First, an overview of the predicated execution concept and its uses is provided. A brief survey of predicated execution support in present and past processors is also presented. Next, the architectural extensions required to provide efficient support for predicated execution are described. The extensions are applied to the base IMPACT architecture that was used for the experiments presented in the previous chapter.

6.1 Overview

Predicated execution refers to the conditional execution of instructions based on the value of a Boolean source operand, referred to as the predicate. If the value of the predicate is true (a logic 1), the instruction is allowed to execute normally; otherwise (a logic 0), the instruction is nullified, preventing it from modifying the processor state. Figure 6.1 contains a simple

```
for ( i = 0; i < 100; i++ )
    if (A[i] ≤ 50 )
        j = j + 1;
    else
        k = k + 1;
```

(a)

```
    mov   r1,0
    mov   r2,0
    ld_i  r3,A,0
L1:
    ld_i  r4,r3,r2
    bgt   r4,50,L2
    add   r5,r5,1
    jmp   L3
L2:
    add   r6,r6,1
L3:
    add   r1,r1,1
    add   r2,r2,4
    blt   r1,100,L1
```

(b)

```
    mov   r1,0
    mov   r2,0
    ld_i  r3,A,0
L1:
    ld_i  r4,r3,r2
    pgt   p1(U),p2(U̅),r4,50
    add   r5,r5,1 (p2)
    add   r6,r6,1 (p1)
    add   r1,r1,1
    add   r2,r2,4
    blt   r1,100,L1
```

(c)

Figure 6.1 Example of if-conversion, (a) source code segment, (b) assembly code segment, (c) assembly code segment after if-conversion.

example to illustrate the concept of predicated execution. For each iteration of the loop in Figure 6.1(a), either the value of j or k is conditionally incremented. The basic compiler transformation to exploit predicated execution is known as if-conversion [19],[20]. If-conversion replaces conditional branches in the code with comparison instructions that define one or more predicates. Instructions control dependent on the branch are then converted to predicated instructions, utilizing the appropriate predicate value. In this manner, control dependences are converted to data dependences.

Figures 6.1(b) and 6.1(c) show the assembly code for the loop example before and after if-conversion. Note that the variables j and k have been placed in registers *r5* and *r6*, respectively. The first conditional branch, *bgt*, in Figure 6.1(b) is replaced by a predicate define instruction, *pgt*, in Figure 6.1(c). The actual semantics of the *pgt* instruction will be discussed later in this chapter. It is sufficient for this example to say that the predicate *p1* is assigned the value 1 if $r4 > 50$ and 0 otherwise. The predicate *p2* is assigned the complement of *p1*. The instructions incrementing the values of *r5* and *r6* are converted to predicated instructions, associated with predicates *p1* and *p2*, respectively. For each loop iteration, either *r5* and *r6* will be incremented by the predicated add instructions, contingent on the results of the predicate define instruction. Also, note that the jump instruction becomes unnecessary after if-conversion.

Predicated execution exists in some format in many current and past systems. The remainder of this section presents a survey of commercial systems that contain predicated execution support.

6.1.1 Predicated execution support in the Cydra 5

The Cydra 5 system is a VLIW, multiprocessor system utilizing a directed-dataflow architecture [18],[74]. Each Cydra 5 instruction word contains seven operations, each of which may be individually predicated. An additional source operand added to each operation specifies a predicate located within the predicate register file. The predicate register file is an array of 128 Boolean (one bit) registers. Within the processor pipeline after the operand fetch stage, the predicate specified by each operation is examined. If the content of the predicate register is one, the instruction is allowed to proceed to the execution stage; otherwise, it is squashed. Essentially, operations whose predicates are zero are converted to *no_ops* prior to entering the execution stage of the pipeline. The predicate specified by an operation must thus be known by the time the operation leaves the operand fetch stage.

The content of a predicate register may only be modified by one of three operations: *stuff*, *stuff_bar*, or *brtop*. The *stuff* operation takes as operands a destination predicate register and a Boolean value as well as an input predicate register. The Boolean value is typically produced using a comparison operation. If the input predicate register is one, the destination predicate

```
        mov   r1,0
        mov   r2,0
        ld_i   r3,A,0
L1:
        ld_i   r4,r3,r2
        gt     r6,r4,50
        stuff  p1,r6
        stuff_bar  p2,r6
        add   r5,r5,1 (p2)
        add   r6,r6,1 (p1)
        add   r1,r1,1
        add   r2,r2,4
        blt    r1,100,L1
```

Figure 6.2 Example of if-then-else predication in the Cydra 5.

register is assigned the Boolean value. Otherwise, destination predicate is assigned to 0. The *stuff_bar* operation functions in the same manner, except the destination predicate register is set to the inverse of the Boolean value when the input predicate value is one. The *brtop* operation is used for loop control in software pipelined loops and sets the predicate controlling the next iteration by comparing the contents of a loop iteration counter to the loop bound.

Figure 6.2 shows the previous example after if-conversion for the Cydra 5. To set the mutually exclusive predicates for the different execution paths shown in this example requires three instructions. First, a comparison must be performed, followed by a *stuff* to set the predicate register for the true path (predicated on *p1*) and a *stuff_bar* to set the predicate register for the false path (predicated on *p2*). This results in a minimum dependence distance of 2 from the comparison to the first possible reference of the predicate being set.

In the Cydra 5, predicated execution is integrated into the optimized execution of modulo scheduled inner loops to control the prologue, epilogue, and iteration initiation [61],[87]. Predicated execution in conjunction with rotating register files eliminates almost all code expansion otherwise required for modulo scheduling. Predicated execution also allows loops with conditional branches to be efficiently modulo scheduled.

6.1.2 Predicated execution support in ARM

The Advanced RISC Machines (ARM) processors consist of a family of processors, which specialize in low cost and very low power consumption [88]. They are targeted for embedded and multi-media applications. The ARM instruction set architecture supports the conditional execution of all instructions. Each instruction has a four bit condition field that specifies the context for which it is executed. By examining the condition field of an instruction and the condition codes in a processor status register, the execution condition of each instruction is calculated. The condition codes are typically set by performing a compare instruction. The condition field specifies under what comparison result the instruction should execute, such as equals, less than, or less than or equals. When the compare instruction result contained in the processor status register matches the condition field, the instruction is executed. Otherwise, the instruction is nullified. With this support, the ARM compiler is able to eliminate conditional branches from the instruction stream.

6.1.3 Limited predicated execution support in other systems

Many other contemporary processors offer some form of limited support for predicated execution. A conditional move instruction is provided in the DEC Alpha, SPARC V9, and Intel Pentium Pro processor instruction sets [76],[89],[90]. A conditional move is functionally equivalent to that of a predicated move. The move instruction is augmented with an additional source operand which specifies a condition. As with a predicated move, the contents of the source register are copied to the destination register if the condition is true. Otherwise, the instruction does nothing. The DEC GEM compiler can efficiently remove branches utilizing conditional moves for simple control constructs [91]. The HP PA-RISC instruction set provides all branch, arithmetic, and logic instructions the capability to conditionally nullify the subsequent instruction [75]. This feature is utilized extensively in the IMPACT compiler to emulate predicated execution support on the HP platform. This will be discussed more extensively in Chapter 8.

The Multiflow Trace 300 series machines supported limited predicated execution by providing *select* instructions [60]. Select instructions provide more flexibility than conditional moves by adding a third source operand. The semantics of a select instruction in C notation are as follows:

```
select dest,src1,src2,cond
dest = ( (cond) ? src1 : src2 )
```

Unlike the conditional move instruction, the destination register is always modified with a select instruction. If the condition is true, the contents of *src1* are copied to the destination; otherwise, the contents of *src2* are copied to the destination register. The ability to choose one of two values to place in the destination register allows the compiler to effectively choose between computations from "then" and "else" paths of conditionals based upon the result of the appropriate comparison.

Vector machines have had support for conditional execution using mask vectors for many years [92]. A mask of a statement S is a logical expression whose value at execution time specifies whether or not S is to be executed. The use of mask vectors allows vectorizing compilers to vectorize inner loops with if-then-else statements.

6.2 Architectural Support for Predicated Execution

An architecture supporting predicated execution must be able to conditionally nullify the side effects of selected instructions based on the value of its predicate. Additionally, the architecture must support efficient computation of predicate values. The architecture chosen for modification to support predicated execution, the IMPACT architecture model, is a statically scheduled, in-order issue, superscalar processor supported by the IMPACT compiler. The predicated execution model used is based upon those of the Cydra 5 and the HPL PlayDoh architectures [18],[42]. This section will present the base architecture and the proposed modifications for predicated execution to the instruction set and microarchitecture.

6.2.1 IMPACT architecture model

The baseline architecture, shown in Figure 6.3, is composed of the processor, instruction cache and data cache sharing a common memory data bus, and the main memory subsystem. The instruction set is based on the HP PA-RISC 1.1 instruction set with the addition of integer multiply and divide instructions. Silent versions of all excepting instructions are also added to facilitate speculative code motion under the general speculation model.

The processor supports in-order issue to the fully pipelined functional units. Each functional unit may contain up to one of each of the following: an integer unit, a floating-point unit, a load-store unit, and a branch unit. A realistic memory subsystem is modeled to accurately show the benefits and disadvantages of new compiler techniques and architectural support. Figure 6.4 shows the four-stage pipeline including instruction fetch (IF), instruction decode/issue (ID), instruction execute (IE), and write-back/retire (WBR).

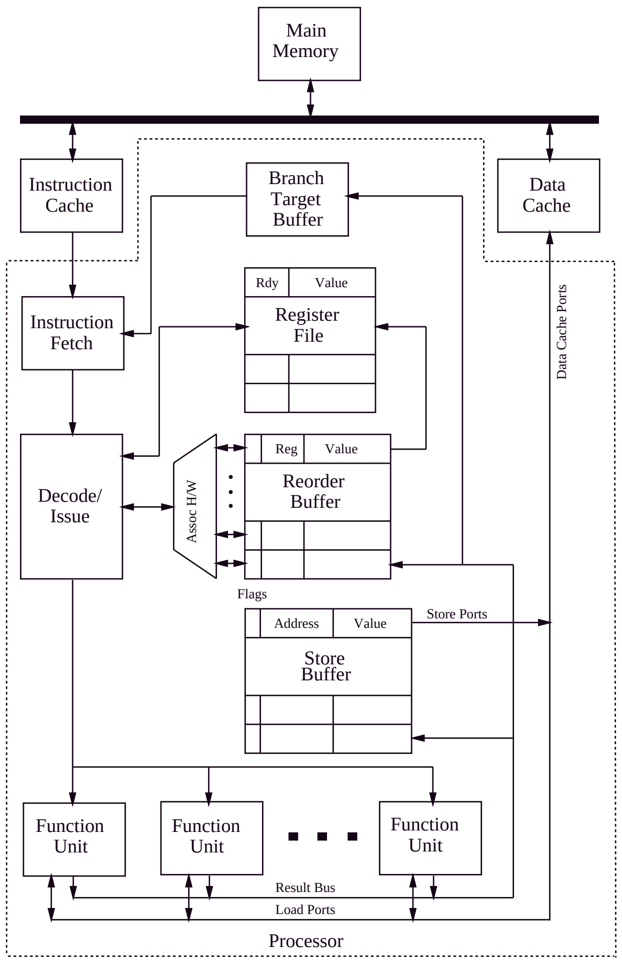


Figure 6.3 IMPACT microarchitecture block diagram.

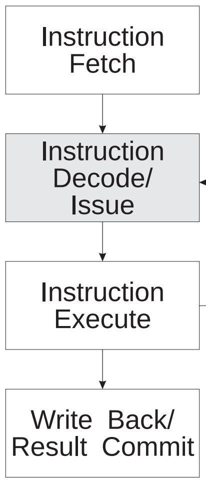


Figure 6.4 Pipeline diagram for the IMPACT architecture.

The processor performs dynamic branch prediction by feeding the fetched instruction addresses into a branch target buffer (BTB). Instructions are speculatively executed until the branch target is determined at the end of the execute stage. If the branch is mispredicted, all instructions fetched after the mispredicted branch are squashed and fetching begins at the correct target address.

The decode stage is responsible for in-order issue in the processor. Instructions are selected from the fetch buffer, decoded, and issued. The in-order model blocks on structural hazards and flow dependences. The reorder buffer is used to maintain a precise state within the processor in the event of an exception or mispredicted branch [93]. The current state of a register within the processor is determined by accessing the register file for the in-order state and by performing an associative search on the reorder-buffer for the most recent value [94]. Stores will only be sent to the data cache when they are retired from the store buffer. This ensures a precise memory state [93].

6.2.2 Instruction set extensions

The Cydra 5 style of supporting full predication is chosen for the IMPACT architecture model. Full predication offers the most efficient and flexible paradigm to support predicated execution. As a result, all instructions in the instruction set architecture are augmented with an additional source operand to hold a predicate specifier. In this manner, every instruction may be predicated. Predicate values are maintained in an Nx1 predicate register file. The details of the predicate register file are discussed in the next section.

Predicates are manipulated via a new set of instructions added to the baseline architecture. These instructions are classified broadly as predicate comparison instructions, predicate clear/set instructions, and predicate save/restore instructions.

Predicate comparison instructions. The most common way to set predicate register values is with a new set of predicate comparison instructions. The predicate comparison semantics used are those of the HPL PlayDoh architecture [42]. Predicate comparison instructions compute predicate values using semantics similar to those for conventional comparison instructions. There is one predicate comparison instruction for each integer, unsigned, float, and double comparison opcode in the original instruction set. The major difference is that these instructions have up to two destination registers and these destination registers are in the predicate register file. The predicate comparison instruction format is shown below.

$$p{<}cmp{>}\ \mathrm{Pout1}(<type>),\ \mathrm{Pout2}(<type>),\ \mathrm{src1},\ \mathrm{src2}\ (\mathrm{P}_{in})$$

This instruction assigns values to *Pout1* and *Pout2* according to a comparison of *src1* and *src2* specified by <*cmp*>. The comparison <*cmp*> can be: equal (eq), not equal (ne), greater than (gt), etc. A predicate <*type*> is specified for each destination predicate. Predicate defining instructions are also predicated, as determined by P_{in}.

Table 6.1 Predicate comparison truth table.

		P_{out}					
P_{in}	Comparison	U	$\overline{U}$	OR	$\overline{OR}$	AND	$\overline{AND}$
0	0	0	0	-	-	-	-
0	1	0	0	-	-	-	-
1	0	0	1	-	1	0	-
1	1	1	0	1	-	-	0

The predicate <*type*> determines the value written to the destination predicate register based upon the result of the comparison and of the input predicate, P_{in}. For each combination of comparison result and P_{in}, one of three actions may be performed on the destination predicate. It can write 1, write 0, or leave it unchanged, indicated by a '-'. Thus, a total of $3^4 = 81$ possible types exist. There are six predicate types that are particularly effective, unconditional (U), OR-type (OR), and AND-type (AND) predicates and their complements. Table 6.1 contains the truth table for these predicate types.

Unconditional destination predicate registers are always defined, regardless of the value of P_{in} and the result of the comparison. If the value of P_{in} is 1, the result of the comparison is placed in the predicate register (or its complement for $\overline{U}$). Otherwise, a 0 is written to the predicate register. Unconditional predicates are utilized for blocks that are executed based on a single condition, i.e., they have a single control dependence. The semantics of the unconditional predicates are analogous to the those of the *stuff* and *stuff_bar* operations in the Cydra 5.

The OR-type predicates are useful when execution of a block can be enabled by multiple conditions, such as logical AND (`&&`) and OR (`||`) constructs in C. OR-type destination predicate registers are set if P_{in} is 1 and the result of the comparison is 1 (0 for $\overline{OR}$); otherwise, the destination predicate register is unchanged. Note that OR-type predicates must be explicitly initialized to 0 before they are defined and used. However, after they are initialized multiple OR-type predicate defines may be issued simultaneously and in any order on the same predicate register. This is true since the OR-type predicate either writes a 1 or leaves the register unchanged which allows implementation as a wired logical OR condition. This property can be utilized to compute an execution condition with zero dependence height using multiple predicate define instructions.

The AND-type predicates are analogous to the OR-type predicates. AND-type destination predicate registers are cleared if P_{in} is 1 and the result of the comparison is 0 (1 for $\overline{AND}$); otherwise, the destination predicate register is unchanged. The AND-type predicate is particularly useful for transformations such as control height reduction [95],[96].

The PlayDoh architecture also provides another predicate type, conditional. The conditional type predicates have semantics similar to regular predicated instructions, such as adds. If the value of P_{in} is 1, the result of the comparison is placed in the destination predicate register (or its complement for $\overline{C}$). Otherwise, no actions are taken. Under certain circumstances, a conditional predicate may be used in place of an OR-type predicate to eliminate the need for an initialization instruction. However, the parallel issue semantics of the OR-type predicates are lost with conditional predicates. For this reason, the IMPACT compiler chooses not to generate conditional type predicates.

As an example of unconditional predicate definition, recall the predicate definition instruction from Figure 6.1:

$$\text{pgt} \;\; \text{p1}(U), \text{p2}(\overline{U}), \text{r4}, 50$$

The predicate source operand is omitted in this example, so it is assumed to be 1. In this case, the value written to predicate register *p1*(U) is the Boolean result of *(r4 > 50)*. Thus, the value written to *p2*($\overline{U}$) is ($\overline{r4 > 50}$). Note that the truth table indicates that in the event that

```
if ( a && b )
        c = c + 1;
else
        d = d + 1;
```

(a)

```
        beq   a,0,L1
        beq   b,0,L1
        add   c,c,1
        jmp   L2
L1:
        add   d,d,1
L2:
```

(b)

```
(1) pclr  p1
(2) peq   p1(OR), p2(U̅),a,0
(3) peq   p1(OR), p3(U̅),b,0 (p2)
(4) add   c,c,1 (p3)
(5) add   d,d,1 (p1)
```

(c)

Figure 6.5 Example usage of OR-type predicate comparisons.

the predicate source operand of a predicate define instruction is false, the value written to a destination predicate register of type U or $\overline{U}$ is 0. In this rare case, an instruction predicated on a false predicate is allowed to modify the processor state.

Figure 6.5 contains an example of the use of OR-type predicates for a block whose execution depends upon multiple conditions. In Figure 6.5(a), the increment of variable c depends upon both a and b being nonzero. The assembly code for this code sequence is shown in Figure 6.5(b) and the if-converted code is shown in Figure 6.5(c). The if-converted code contains three predicate definition instructions, the first of which serves to explicitly initialize the contents of predicate register *p1* to 0. The subsequent predicate definitions require further explanation.

Instruction (2) in Figure 6.5(c) is setting two predicates. Predicate *p1*(OR) controls the "else" case, i.e., the increment of d. Predicate *p2*($\overline{U}$) controls the execution of the second predicate define instruction. Note that in instruction (2), predicate *p1* is being defined as OR-type and predicate *p2* is begin defined as an unconditional complement. From Table 6.1, the value written to *p1* is a one, if *(a == 0)*. This is correct since the "else" case will be executed if *(a == 0)*. The value written to *p2*($\overline{U}$) is ($\overline{a == 0}$).

Instruction (3) in Figure 6.5(c) also sets two predicates. Predicate $p1(OR)$ again controls the execution of the "else" case, and predicate $p3(\overline{U})$ controls the execution of the "then" case, i.e., the increment of c. A one is written to $p1$, if *(b == 0)* and the value of $p2$ is one. So overall, the "else" case is executed if either *(a == 0)* or *(b == 0)*. The value written to $p3$ is one if the Boolean result of $(\overline{b == 0})$ is true and the value of $p2$ is one. That is, the "then" case is executed only if both a and b are nonzero.

Predicate clear/set instructions. The stylized use of OR-type and AND-type predicates described previously requires that the predicates be precleared and preset, respectively. Three sets of instructions are provided for these purposes. First, to individually clear and set individual predicates, *pclr* and *pset* instructions are added to the instruction set. Each takes up to two destination predicate registers and sets the value of zero or one to those destinations. Note that unconditional predicate comparison instructions could also be used for the purpose of setting individual predicates to zero or one. Therefore, the additional instructions may not be required. However, it may be more efficient to provide these special clearing and setting instructions because they do not require any source operands.

The second set of instructions added for clearing and setting predicates consists of *pclr_all* and *pset_all* instructions. These instructions set the entire contents of the predicate register file to zero or one in a single cycle. These instructions are particularly useful before entering a section of predicated code which makes extensive use of OR-type or AND-type predicates. Note, the compiler has to be particularly careful using these instructions to ensure that no live predicates are destroyed.

Finally, as provided in the HPL PlayDoh architecture, instructions to clear and set groups of registers using a mask are provided [42]. These instructions are aptly called *pclr_mask* and *pset_mask*. These instructions set a contiguous group of 32 predicate registers to zero or one using a mask. Thus, any combination of the 32 predicates can be cleared or set using these instructions. For architectures with more than 32 predicates, a sequence of these instructions may be required to initialize all the desired predicates.

Predicate save/restore instructions. Extensions to the base instruction set allow two methods of saving and restoring the contents of the predicate register file. The *pld_blk* and *pst_blk* instructions allow the loading and storing of the predicate register file in 32 bit blocks. These instructions are primarily used to save/restore the caller-save predicates across subroutine calls and to save/restore the callee-save predicates at function entry and exit points. They also make saving the contents of the predicate register file during a context switch more efficient.

The second method acts on individual predicate registers and is only required if the need arises to spill predicate registers. The IMPACT compiler employs an intelligent allocation algorithm method to avoid spilling predicate registers. In the rare situation in which a predicate register has to be spilled, *pld* and *pst* instructions are used. These instructions allow an individual predicate register to be loaded from and stored to memory. In this manner, the compiler has the freedom to handle predicate registers in the same way as the conventional register types.

6.2.3 Microarchitecture extensions

To support predicated execution, some modifications to the baseline architecture presented in Section 6.2.1 are required. The extensions are broadly broken down into two categories: the nullification mechanism and the predicate register file.

Nullification mechanism. The predicate of each instruction determines its execution state. If the predicate is 1, or true, the instruction is executed normally; if the value is 0, or false, the effects of the instruction are nullified. In general, nullification may be accomplished at any point in the processor pipeline before the register file or memory system is modified.

The earliest an instruction may be nullified is during the decode/issue stage. After fetching the value of an instruction's predicate, the instruction is simply not issued if its predicate is 0. This has the advantage of allowing the execution unit to be allocated to other operations. Thus, for critical resources such as divide units, a nullified instruction will never tie it up unnecessarily. Also, for nullified load instructions, superfluous cache and TLB misses will never be generated. On the negative side, the value of the predicate register referenced must be available during decode/issue, so the predicate register must at least be set in the previous cycle. This dependence distance may also be larger for deeper pipelines or if bypass is not available for predicate registers. Increasing the dependence distance between definitions and uses of predicates may adversely affect execution time by lengthening the schedule for predicated code. This nullification model is utilized in the Cydra 5 [18].

The other extreme for nullification is to allow the instruction to execute almost to completion, but to disallow any change of processor state in the write-back stage of the pipeline. Therefore, for instructions that write their result into the register file, this update must be suppressed. For store instructions, they must be prevented from entering the store buffer. This method is useful since it reduces the latency between an instruction that modifies the value of a predicate register and a subsequent instruction which is conditioned based on that predicate register. This reduced latency enables more compact schedules to be generated for predicated code. A drawback to this method is that regardless of whether an instruction is suppressed, it still ties up an execution unit. This method is also likely to increase the complexity of the register bypass logic and force exception signaling to be delayed until the last pipeline stage.

Hybrid nullification schemes are also possible and become more appealing for deeply pipelined machines to balance the effects of both extremes. For the IMPACT architecture, nullification at the decode/issue stage is chosen. The IMPACT architecture contains a very short pipeline (four stages) and the reduced design complexity makes this the preferred choice. Also, it is believed that the negative of increased dependence height incurred by this approach can be overcome with compiler transformations such as predicate promotion. These transformations are discussed in detail in Chapter 7.

Predicate register file. As previously mentioned, an Nx1 register file to hold predicates is added to the baseline architecture. The choice of introducing a new register file to hold predicate values rather than using the existing general purpose register file was made for several reasons. First, it is inefficient to use a 32 bit general register to hold a one bit predicate. Second, register porting is expected to be a significant problem for wide-issue processors. By keeping predicates in a separate file, additional port demands are not added to the general purpose register file. Within the architecture, the predicate register file behaves no differently than a conventional register file. For example, the contents of the predicate register file must be saved during a context switch. Furthermore, the predicate file is partitioned into caller and callee save sections based on the chosen calling convention.

6.2.4 Predicated execution for out-of-order issue processors

Although out-of-order execution is not the focus of this dissertation, a brief discussion of some of the issues involved with predicated execution is presented in this section. Superscalar processors employing out-of-order execution via an algorithm, such as the Tomasulo algorithm, face new problems with predicated execution [46],[97]. The problems mainly stem from the

```
A: ld_i   r1,r2,r3
B: add    r1,r4,r5 (p1)
C: ld_i   r6,r1,0
```

Figure 6.6 Example of the tagging problem with out-of-order execution.

tagging mechanism used to forward results to instructions waiting in the reservation stations. In the conventional Tomasulo algorithm, an instruction deposits its ID or tag into its destination register when it is issued. Subsequent instructions which use this destination register as a source operand receive the producing instruction's tag when the register value is fetched. This tag indicates the instruction which will forward the correct value for the particular register. Since there is guaranteed to be a unique producer of each register value at a given point in the execution of a program, a unique tag is always available.

With predicated execution, the problem is that instructions conditionally write to their destination register. As a result, when an instruction is ready for issuing, there may not be a unique tag from which to obtain its unavailable source operands. For example, consider the execution of the code stream shown in Figure 6.6. When instruction A is issued, it deposits tag A into its destination register, *r1*. Next, instruction B is issued, thereby writing tag B into its destination register, again *r1*. Now, when the operands for instruction C are fetched, the producer of its source operand, *r1*, is assumed to be the last instruction to write to *r1*, namely instruction B. In the cases in which the predicate of instruction B is true, everything is handled properly. However, in the case in which the predicate of instruction B is false, no result will be forwarded to instruction C, which causes an error.

One potential solution is to not allow instruction B to place its tag in its destination register unless its predicate is true. The problem with this is that much of the out-of-order execution

```
op dest,src1[,src2,...] (pred)

old_dest = dest
if ( pred )
    dest = op(src1[,src2,...])
else
    dest = old_dest
```

Figure 6.7 Select instruction execution semantics for predicated instructions.

capabilities of the processor are lost. With this solution, the processor must stall whenever the predicate of an instruction is not available, whereas the underlying principle of out-of-order execution is to continue issuing instructions regardless if their source operands are available. The instructions not ready wait in reservation stations allowing ready instructions to bypass them. Therefore, much of the out-of-order performance potential is sacrificed with this scheme.

A second solution is to utilize the execution semantics of the select instruction, from Section 6.1.3, for all predicated instructions that write to a destination register. A predicated instruction writes one of two values into its destination register as shown in Figure 6.7. When its predicate is true, the standard operation is applied, and the value computed by the instruction is written to the destination register. In contrast, when its predicate is false, the instruction is issued as a move instruction; copying the original contents of the destination register to the destination register. An extra register read operation is thereby required to obtain the original contents of the destination operand. With this approach, predicated instructions always produce a result. Therefore, the tagging problem is eliminated because the last instruction issued that writes to a particular register is guaranteed to produce a value. As a result, a unique tag is guaranteed to be available for each pending register value during execution.

The advantage of this approach is its natural applicability to out-of-order issue processors. The underlying tagging and data forwarding mechanisms require little change to function correctly with predicated execution. A negative of this approach is the increased dependence chain lengths incurred for predicated code. Predicated instructions that target the same destination register are serialized because each instruction must read the original value of its destination register. Therefore, they must wait for all previous instructions targeting the same destination to produce a result. This can be quite undesirable especially when code from mutually exclusive paths is overlapped. A second negative of this approach is the additional register read port that is required to obtain the original contents of the destination register.

Alternative schemes have been proposed in the literature to overcome the tagging difficulties with predicated execution. The use of statically defined tags has been proposed [98]. With statically defined tags, the compiler can force two instructions which write to a common destination register under mutually exclusive predicates to have the same tag. In this manner, a subsequent use will utilize this common tag and be provided the result from the appropriate instruction based on the value of the predicate. This eliminates the serialization effects of using the select instruction semantics.

Another scheme uses a more hardware-oriented approach to provide multiple tags for each register [99]. With this approach, registers are allowed to have multiple tags which indicate multiple outstanding conditional updates of a register. Instead of instructions replacing the tag field of their destination register, predicated instructions just add their tag to the beginning of a list of tags. By not replacing the tag, the instruction does not have to wait for all previous predicated instructions which write to the register to complete. Subsequent uses that await in the reservation stations are provided forwarded data for all tags which match an entry in their source operand lists. A prioritization scheme is used so that the first such tag in the list which provides a result from a nonnullified instruction is the result which is used. Again, this technique eliminates the serialization effects of using the select instruction semantics.

CHAPTER 7

COMPILER SUPPORT FOR PREDICATED EXECUTION

Predicated execution provides a large number of opportunities to enhance and expose ILP in the presence of branches. However, as with speculative execution, an aggressive compiler is required to realize most of the performance advantages. Compiler optimization and transformation techniques focus on eliminating branches from the instruction stream and overlapping the execution of multiple control flow paths using the conditional execution capabilities provided by predication. The compiler support for predicated execution is based on a new structure referred to as the hyperblock. Hyperblocks are a generalized form of superblocks that take advantage of both predicated and speculative execution.

This chapter is a detailed description of hyperblock compilation techniques. The formation procedure for hyperblocks is described first. Second, extensions to traditional optimization, instruction scheduling, and register allocation techniques to enable them to work on hyperblocks are discussed. The presence of predicates introduces new challenges into the compiler backend to understand the meaning of predicates, take advantage of the relations among predicates, and perform transformations in the presence of predicates. Finally, a set of four new optimizations designed specifically for improving the performance of predicated code is presented.

7.1 Hyperblock Formation

As discussed in the previous chapter, the most basic compiler transformation utilizing predicated execution is if-conversion. The traditional approach has been to apply if-conversion to entire innermost loops to enable vectorization or modulo scheduling of loops with conditional branches [19],[87]. This could also be extended to handle certain control structures, such as hammocks, in nonloop portions of the code. The major problem with this approach is that if-conversion is an all or nothing transformation. With the large number of branches and corresponding control flow paths present in nonnumeric applications, a more flexible strategy that efficiently supports selective if-conversion is required. To support such a flexible method, the hyperblock is introduced.

A hyperblock is a collection of connected basic blocks in which control may only enter through the first block, referred to as the entry block. Control flow may leave from any number of blocks in the hyperblock. All control flow between basic blocks in a hyperblock is removed via if-conversion. The goal of hyperblocks is to intelligently group basic blocks from many different control flow paths into a single manageable block for compiler optimization and scheduling.

Hyperblocks are formed using a five-step procedure: region identification, loop backedge coalescing, block selection, tail duplication, and if-conversion. A running example is utilized throughout this section to illustrate hyperblock formation. The example chosen is the inner loop from the benchmark *wc*. The pre-processed C source code for the loop segment is shown in Figure 7.1. This example was chosen for two reasons. First, it contains a loop that accounts for a large fraction of the benchmark execution time, yet is small enough to be presented in the context of this dissertation. Second, the loop has a nontrivial control structure, which presents a challenge to all branch handling strategies.

```
      linect = wordct = charct = token = 0;
      for (;;)
      {
A:       if (--(fp)->cnt < 0)
C:          c = filbuf(fp);
         else
B:          c = *(fp)->ptr++;
D:       if (c == EOF)   break;
E:       charct++;
         if ((' ' < c) &&
F:          (c < 0177))
         {
H:          if (! token)
            {
K:             wordct++;
               token++;
            }
            continue;
         }
G:       if (c == '\n')
I:          linect++;
J:       else if ((c != ' ') &&
L:             (c != '\t'))   continue;
M:       token = 0;
      }
```

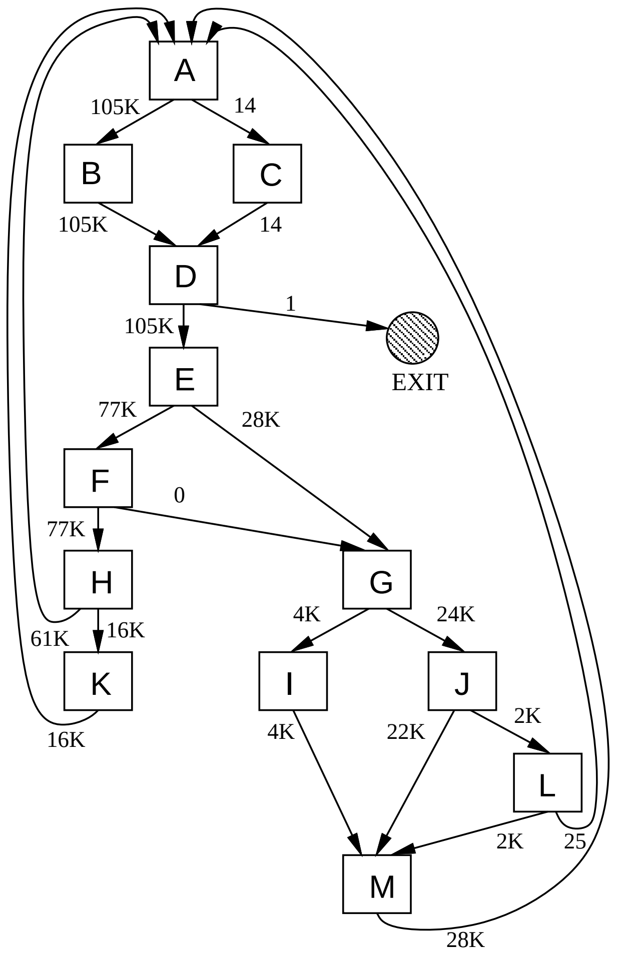
Figure 7.1 Source code for the inner loop of *wc*.

The purpose of *wc* is to count the number of characters, words, and lines in an input file. A character buffer is processed in the loop and re-filled as necessary until the end-of-file marker is encountered. The corresponding assembly code and control flow graph for the loop segment are presented in Figure 7.2. The control flow graph is augmented with the execution frequencies of each control transfer for the measured run of the program. The basic blocks are consistently identified by the letters A through M in both figures. The loop is characterized by small basic blocks and a large percentage of branches. Overall, the loop segment contains 13 basic blocks with a total of 34 instructions. Of the 34 instructions, 14 are branches, 8 conditional, 5 unconditional, and 1 subroutine call. The remainder of this section describes each step of the hyperblock formation procedure.

Step 1 - Region identification. The blocks for a hyperblock are chosen from regions in the control flow graph. A region is a group of basic blocks with a single entry block that

```
LA: ld_i  r98, r3,  0
    add  r27, r98, −1
    st_i  r3, 0, r27
    blt  r98, 1, LC
LB: ld_i  r30, r3, 4
    add  r29, r30, 1
    st_i  r3, 4, r29
    ld_c  r4, r30, 0
LD: beq  r4, −1, EXIT
LE: ld_i  r33, r73, 0
    add  r32, r33, 1
    st_i  r73, 0, r32
    bge  32, r4, LG
LF: bge  r4, 127, LG
LH: bne  0, r2, LA
LK: ld_i  r36, r72, 0
    add  r35, r36, 1
    st_i  r72, 0, r35
    add  r2, r2, 1
    jmp  LA
LG: beq  r4, 10, LI
LJ: bne  r4, 32, LL
LM: mov  r2, 0
    jmp  LA
LI: ld_i  r39, r71, 0
    add  r38, r39, 1
    st_i  r71,0, r38
    jmp  LM
LL: bne  r4, 9, LA
    jmp  LM
LC: mov  Parm0, r3
    jsr  filbuf
    mov  r4, Ret0
    jmp  LD
```

(a)



(b)

Figure 7.2 Inner loop segment of *wc*, (a) assembly code, (b) weighted control flow graph.

dominates all blocks in the region [52]. Typical regions are loop bodies, intervals, if-then-else conditionals, and nested combinations of these. The regions serve as outer boundaries for hyperblock formation. The compiler attempts to identify the largest regions as possible under two constraints. First, a basic block may only reside in a single region. Second, the region may contain no internal cycles. The second constraint is later relaxed to support loop peeling, as will be described in Section 7.3.3. For the *wc* example, the innermost loop body is identified as a region.

Step 2 - Backedge coalescing. The second step of hyperblock formation applies only to loop regions. All loop backedges of loop regions are coalesced into a single backedge. This is done because if-conversion can only remove nonloop branches. By coalescing all the backedges, the control logic that determines the particular backedge that is traversed becomes a candidate for elimination via if-conversion. A single branch back to the loop header is taken whenever any of the backedges were taken in the original loop.

Examination of the control flow graph for the *wc* example in Figure 7.2 shows that the loop contains four loop backedges, i.e., the branches from basic blocks H, K, L, and M to basic block A. These branches currently cannot be eliminated with if-conversion. However, these branches can be retargeted to a new block N, as shown in Figure 7.3. Block N simply contains an unconditional branch back to block A. The loop region now contains only a single loop backedge. Additionally, the four branches to block N are now candidates for elimination with if-conversion.

Step 3 - Block selection. The third step of hyperblock formation is choosing a set of basic blocks to combine into a hyperblock. Blocks are selected based on two high-level, possibly conflicting goals. First, including more blocks can potentially improve performance by eliminating branches among the included blocks. Second, including too many blocks is likely to result in an overall performance loss due over-saturation of processor resources or increased dependence height. These conflicting goals must be addressed by the block selection algorithm.

Blocks are selected by enumerating execution paths through the region. An execution path is a path of control flow from the entry block to an exit block in the region. A priority is calculated

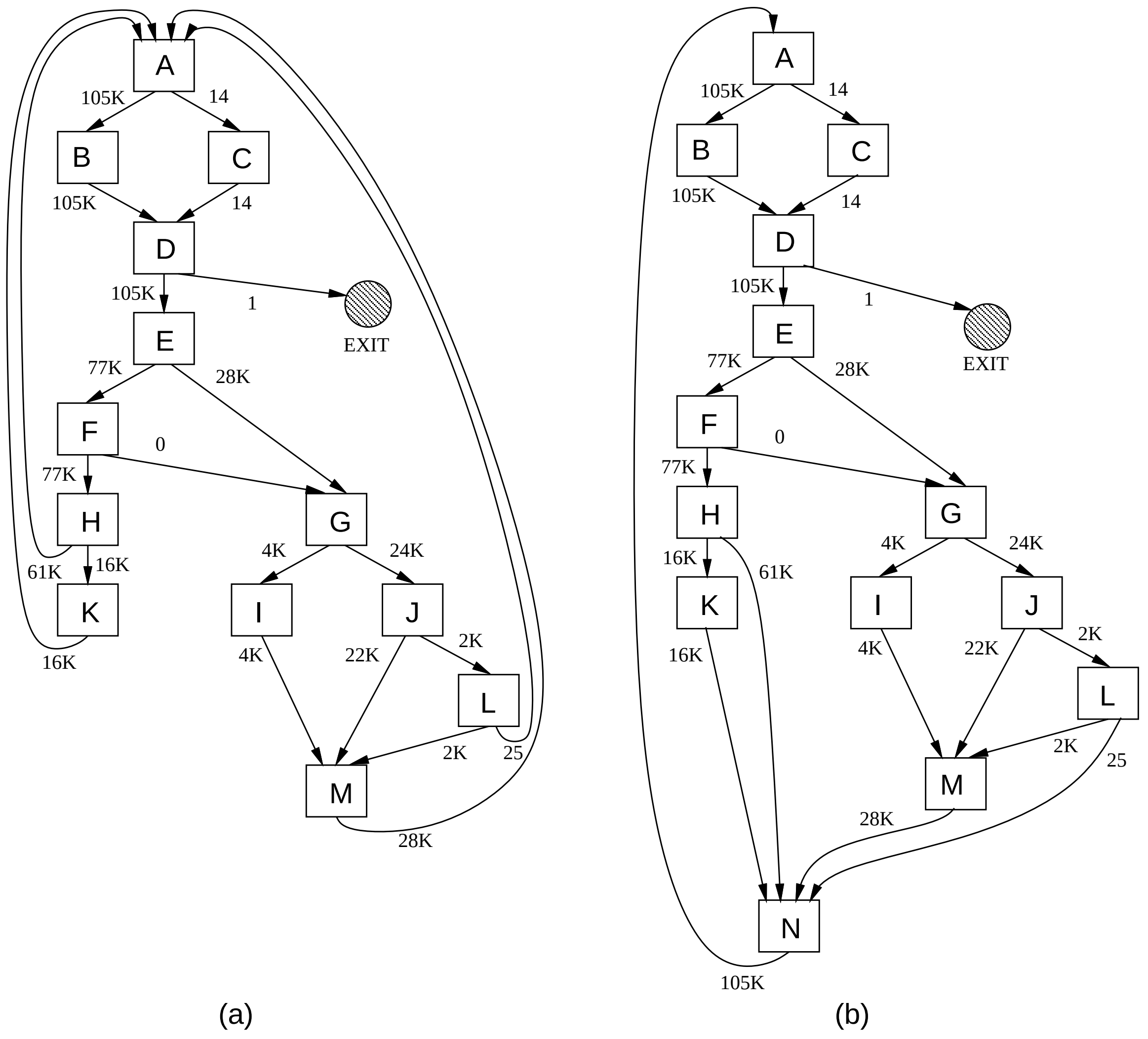


Figure 7.3 Backedge coalescing applied to the inner loop segment of *wc*, (a) before coalescing, (b) after coalescing.

for each path to determine its relative importance. Paths are included from highest priority to lowest priority based upon the estimated available execution resources and the characteristics of the path. The final set of selected blocks is then the union of all blocks along paths chosen for inclusion.

The path priority function is a combination of four elements: path execution frequency, number of instructions on the path, path dependence height, and hazard conditions on the path. Execution frequency is used to give paths with higher execution frequency a higher priority. In general, execution frequency is used to exclude paths of control which are not often executed. Removing infrequent paths eliminates dependence constraints for optimization and scheduling associated with these paths. Also, the demand for resources is reduced by omitting these paths. The number of instructions along a path is used to give higher priority to paths with fewer instructions. Longer paths utilize more machine resources and are likely to reduce the overall performance of the hyperblock if they are combined with shorter paths.

The dependence height of a path is used to give paths with larger dependence height a lower priority. When multiple paths are merged together in a hyperblock, the dependence height of the resultant hyperblock is the maximum across all paths. Therefore, the overall performance of a hyperblock can be reduced by merging a path with a very large relative dependence height. Finally, any hazard conditions that exist along a path are used to give the path lower priority. Hazard conditions include procedure calls and unresolvable memory stores (typically pointer updates). Hazard conditions limit the effectiveness of optimization and scheduling for the entire hyperblock since the compiler must make conservative assumptions regarding the hazards to ensure correctness.

The path priority function is defined more precisely by the following three equations:

$$dep\_ratio_i = 1.0 - (dep\_height_i / \max_{1 \leq j \leq N}(dep\_height_j)) \tag{7.1}$$

$$op\_ratio_i = 1.0 - (num\_ops_i / \max_{1 \leq j \leq N}(num\_ops_j)) \tag{7.2}$$

$$priority_i = (probability_i \times hazard_i) \times (dep\_ratio_i + op\_ratio_i + K) \tag{7.3}$$

Equation (7.1) calculates the ratio of a particular path's dependence height with respect to the path with the largest dependence height in the region. In order to make smaller dependence heights more favorable, this ratio is subtracted from one. Correspondingly, the ratio of the number of operations on a particular path with respect to the largest number of operations along a path in the region is calculated by Equation (7.2). These two equations are used to gauge the height and resource dominance of each path through the region.

The overall priority is calculated by Equation (7.3). The priority is the product of two terms. The first term is the probability the path is traversed scaled by a hazard multiplier. The hazard multiplier is used to reduce the probability of paths that contain a hazardous instruction. Currently, a value of 0.25 is used for any path containing a subroutine call or an unresolvable memory store. For paths containing containing no hazards, a value of 1.0 is used. The second product term is the sum of the previously computed dependence and operation ratios along with a constant term, K. The constant term is used to indicate a base contribution of the path probability. In this manner, a path with the largest dependence height and number of operations still may have a nonzero priority. Currently, the value of K is set to 0.1.

As previously mentioned, after the priorities for all paths are calculated, the paths are sorted in priority order and considered for inclusion from highest to lowest priority. The algorithm used for block selection is presented in Figure 7.4. Paths are included in the hyperblock provided

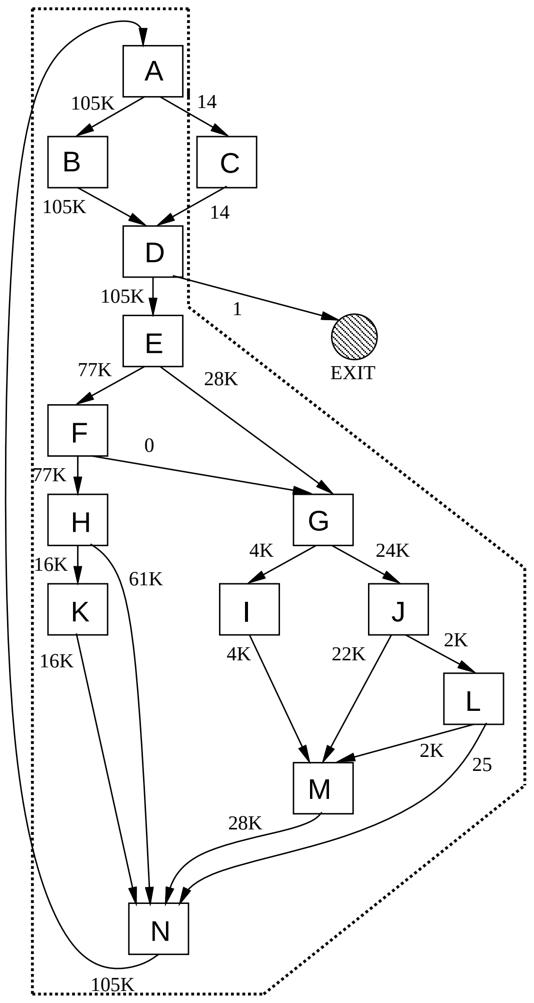
```
/* Predefined variables for block selection */
ISSUE_WIDTH = 1 to 8          /* As specified in the machine description file */
RES_MULTIPLIER = 2
MAX_DEP_GROWTH = 3
MIN_PATH_PRIORITY_RATIO = 0.10

block_selection(region) {
    enumerate all paths in region
    calculate priority of each path
    sort paths from largest to smallest priority
    /* Initialization of loop variables */
    $avail\_resources = ISSUE\_WIDTH \times dep\_height_1 \times RES\_MULTIPLIER$
    $used\_resources = 0$
    $last\_priority = 0.0$
    $sel\_paths = 0$
    for ($i = 1$ to $num\_paths$) {
        /* Check if there enough resources available to include the path */
        if (($num\_ops_i + used\_resources$) > $avail\_resources$) {
            continue
        }
        /* Prevent paths with large relative dependence heights from being included */
        if ($dep\_height_i > (dep\_height_1 \times MAX\_DEP\_GROWTH)$) {
            continue
        }
        /* Do not include paths with a small relative priority to that of the last included path */
        if ($priority_i < (last\_priority \times MIN\_PATH\_PRIORITY\_RATIO)$) {
            continue
        }
        /* Include the path in the hyperblock */
        $sel\_paths = sel\_paths \cup path_i$
        $used\_resources = used\_resources + num\_ops_i$
        $last\_priority = priority_i$
    }
    $sel\_blocks$ = all blocks contained within $sel\_paths$
    return $sel\_blocks$
}
```

Figure 7.4 Block selection algorithm.

that they do not violate any of the following three conditions. First, the additional resources required by a path may not cause the total number of resources required by the hyperblock to exceed the estimated available resources. Second, the dependence height of a path may not exceed the dependence height of the highest priority path (dep_height_1) by more than a predefined fraction. Finally, the priority of a path must be within some fraction of the priority for the last included path. This restriction prevents disparate low priority paths from being included in a hyperblock consisting of high priority paths. The final set of blocks that are actually selected for inclusion are calculated by taking the union of all blocks along the selected paths.

The block selection algorithm utilizes a simplified scheme to model processor resources. Resources are modeled by keeping track of the estimated number of available instruction slots. Currently, instruction slots are not classified by allowable instruction types. Therefore, each instruction under consideration may be placed in any available slot. The available number of instruction slots is calculated by multiplying the issue width of the target processor by the dependence height of the highest priority path. In addition, the number of available resources is increased by a padding factor referred to as the *RES_MULTIPLIER*. A padding factor of 1.0 constrains the selection algorithm to not increase the schedule length of the highest priority path due to resource demands of other paths. In practice, this was found too restrictive. For many cases, increasing the schedule length of the highest priority path by a modest margin is profitable because more paths can be overlapped. For this dissertation, the *RES_MULTIPLIER* was set to 2.0.

The application of the block selection algorithm to the *wc* example is illustrated in Figure 7.5. In the right-hand portion of the figure, the execution paths are enumerated in priority



1. A–B–D–E–F–H–N
2. A–B–D–E–F–H–K–N
3. A–B–D–E–G–J–M–N
4. A–B–D–E–G–J–L–M–N
5. A–B–D–E–G–I–M–N
6. A–B–D–E–G–J–L–N
7. A–B–D
8. A–C–D–E–F–H–N
9. A–C–D–E–F–H–K–N
10. A–C–D–E–G–J–M–N
11. A–C–D–E–G–J–L–M–N
12. A–C–D–E–G–I–M–N
13. A–C–D–E–G–J–L–N
14. A–C–D
15. A–B–D–E–F–G–I–M–N
16. A–B–D–E–F–G–J–M–N
17. A–B–D–E–F–G–J–L–M–N
18. A–B–D–E–F–G–J–L–N
19. A–C–D–E–F–G–I–M–N
20. A–C–D–E–F–G–J–M–N
21. A–C–D–E–F–G–J–L–M–N
22. A–C–D–E–F–G–J–L–N

Figure 7.5 Block selection applied to the inner loop segment of *wc*.

order. This loop region contains 22 unique paths, with the path A-B-D-E-F-H-N having the highest priority. Paths 1-7 are chosen for inclusion by the block selection algorithm. After path 7, the priority value for the remaining paths drops dramatically due to their low execution frequency. Additionally, block C contains a hazardous instruction (a subroutine call), so the priority of all paths which contain block C is further reduced. The blocks which are actually selected for inclusion are then calculated by taking the union of all blocks from the selected paths. The result of block selection is that all blocks with the exception of block C are chosen for the hyperblock. With this strategy, some paths which were not chosen may indeed be included in the hyperblock. For this example, paths 15-18 are actually also selected since all the blocks which lie along those paths are chosen. In reality these paths could be excluded if desired, but little advantage is gained by doing this.

Step 4 - Tail duplication. In order to make the eventual hyperblock be single entry, control flow from nonselected blocks to selected blocks (other than the entry block) must be eliminated. Such paths of control are referred to as *side entry points* into the hyperblock. In the example (Figure 7.5), a side entry point exists from block C to block D. Tail duplication is used to remove all side entry points of a hyperblock. The tail duplication algorithm transforms the control flow graph by first marking all blocks which have side entry points. Then, all selected blocks that may be reached from a marked block without passing through the entry block are also marked. Finally, all the marked blocks are duplicated and the control flow arcs corresponding to the side entry points are adjusted to transfer control to the corresponding duplicate blocks. Note that blocks are duplicated at most one time regardless of the number of side entry points.

The *wc* example after tail duplication is shown in Figure 7.6. The set of blocks which must be duplicated is identified by first marking the target of the side entry point, namely, block D. Then all selected blocks in which control can reach from block D without passing through block A are marked. The set of reachable blocks contains blocks E, F, G, H, I, J, L, M, and N. Tail duplication proceeds by replicating block D and all of the reachable blocks. Lastly, the C-D control arc is adjusted to C-D$'$ to remove the side entrance.

Step 5 - If-conversion. The final phase of hyperblock formation is if-conversion. If-conversion removes all control flow among the blocks selected for the hyperblock using con-

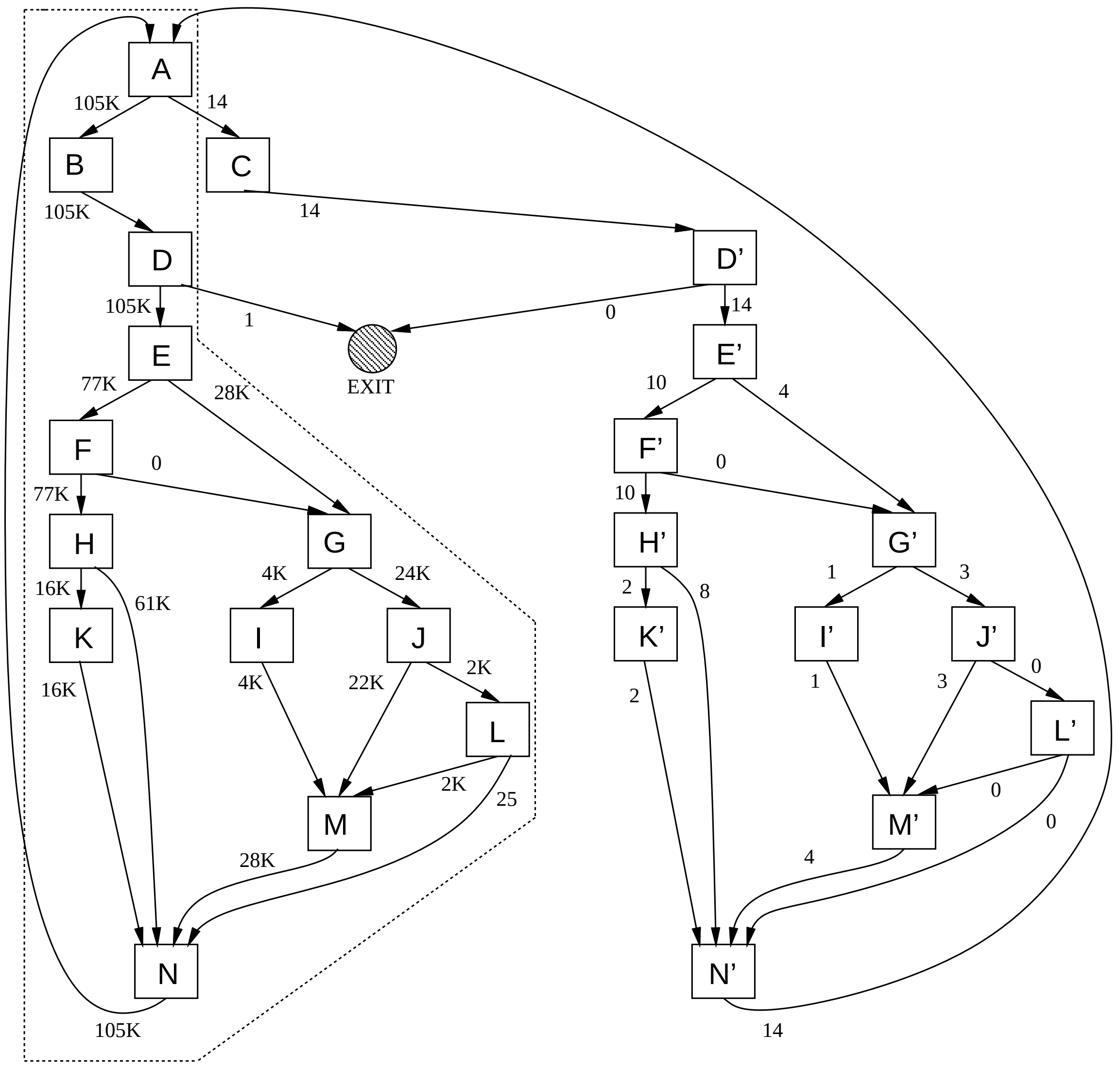
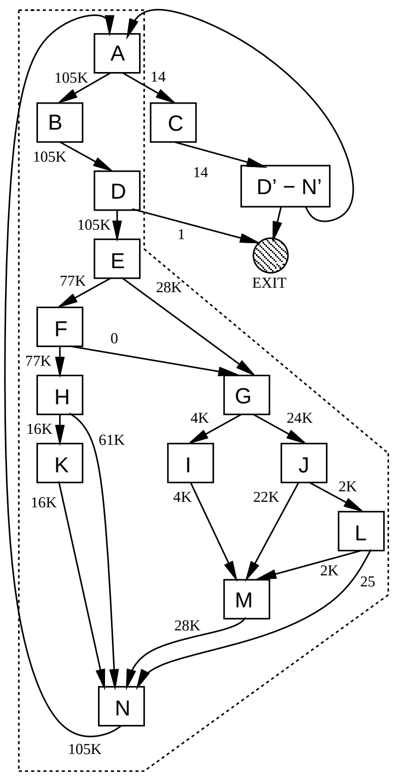


Figure 7.6 Tail duplication applied to the inner loop segment of *wc*.

ditional execution. However, explicit branches remain to handle all control flow which exits the hyperblock. In the current implementation, a variant of the RK if-conversion algorithm is utilized [20]. The if-conversion algorithm first calculates the localized control dependence information among the selected basic blocks [100]. Control dependences are maintained as a set of edges in the control flow graph which determine the execution condition of a particular basic block. The control dependence information is localized because only control flow among the selected blocks is considered. All control dependences resulting from branches not in the region or branches which exit the hyperblock are ignored for the purposes of calculating control dependences. This strategy minimizes the number of control dependences represented with predicates to only those branches which are targeted for elimination.

Once the control dependence information is calculated, one predicate register is assigned to represent each unique set of control dependences. Therefore, all blocks which share a common set of control dependences will be executed under the same predicate. Predicate comparison instructions are inserted into all basic blocks which are the source of the control dependence edges associated with a particular predicate. The predicate compare condition is determined by the branch condition specified by a particular control dependence edge. After the predicate comparison instructions are inserted, all instructions in each selected block, including the newly inserted predicate comparisons, are conditioned under the predicate assigned to their block. Finally, all conditional and unconditional branches from selected blocks to other selected blocks are removed. The predicated code is placed linearly in the final hyperblock using a topological sort of the original hyperblock control flow graph.

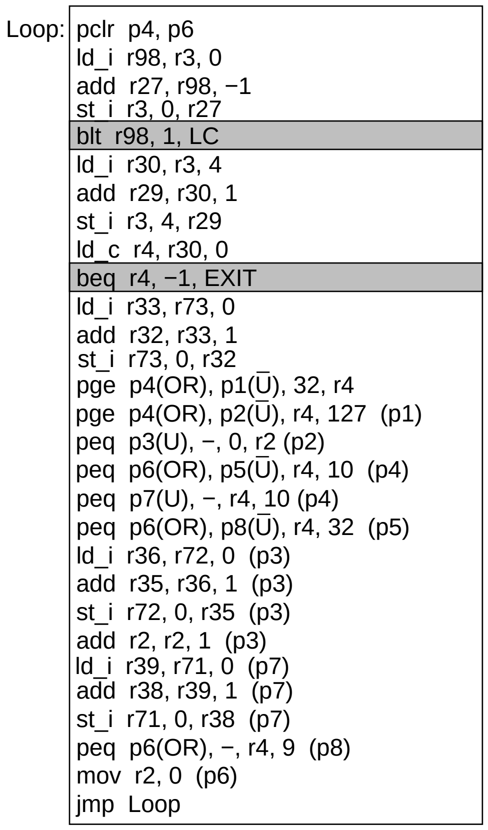
The if-conversion step performed on the *wc* example is illustrated in Figures 7.7 and 7.8. The calculation of the localized control dependence information and the predicate assignment



Control Dependences	Predicate Assignment
A : none	A : null
B : none	B : null
D : none	D : null
E : none	E : null
F : brE	F : p1 (U)
G : $\overline{\text{brE}}$, $\overline{\text{brF}}$	G : p4 (OR)
H : brF	H : p2 (U)
I : brG	I : p7 (U)
J : $\overline{\text{brG}}$	J : p5 (U)
K : brH	K : p3 (U)
L : $\overline{\text{brJ}}$	L : p8 (U)
M : brI, brJ, brL	M : p6 (OR)
N : none	N : null

Figure 7.7 Localized control dependence calculation and predicate assignment for the inner loop segment of *wc*.

are shown in Figure 7.7. Blocks A, B, D, E, and N have no local control dependences. Therefore, these blocks will always be executed if the hyperblock is not exited prematurely through a side exit and do not require predicates. The remaining blocks are control dependent on the edges specified in the figure. Control dependences are denoted by indicating the branch from which they originate. True and complement conditions are used to distinguish the left-hand and right-hand control flow arcs out of a particular block, respectively. For example, the control dependence for block J is $\overline{brG}$, indicating the right-hand edge leaving block G. The example hyperblock contains eight unique sets of control dependences, thus eight predicates are required. The mapping of control dependences to predicates and the assignment of predicates to basic blocks are also shown in Figure 7.7.



```
Loop: pclr  p4, p6
      ld_i  r98, r3, 0
      add  r27, r98, −1
      st_i  r3, 0, r27
      blt  r98, 1, LC
      ld_i  r30, r3, 4
      add  r29, r30, 1
      st_i  r3, 4, r29
      ld_c  r4, r30, 0
      beq  r4, −1, EXIT
      ld_i  r33, r73, 0
      add  r32, r33, 1
      st_i  r73, 0, r32
      pge  p4(OR), p1(Ū), 32, r4
      pge  p4(OR), p2(Ū), r4, 127  (p1)
      peq  p3(U), −, 0, r2 (p2)
      peq  p6(OR), p5(Ū), r4, 10  (p4)
      peq  p7(U), −, r4, 10 (p4)
      peq  p6(OR), p8(Ū), r4, 32  (p5)
      ld_i  r36, r72, 0  (p3)
      add  r35, r36, 1  (p3)
      st_i  r72, 0, r35  (p3)
      add  r2, r2, 1  (p3)
      ld_i  r39, r71, 0  (p7)
      add  r38, r39, 1  (p7)
      st_i  r71, 0, r38  (p7)
      peq  p6(OR), −, r4, 9  (p8)
      mov  r2, 0  (p6)
      jmp  Loop
```

Figure 7.8 Inner loop segment of *wc* after if-conversion.

Unconditional predicates are used for predicates which have a single edge in their control dependence sets. For the example, predicates p1, p2, p3, p5, p7, and p8 are unconditional. On the other hand, OR-type predicates are used for predicates which have multiple edges in their control dependence sets. OR-type predicates are necessary with multiple edges since the predicate should be set to 1 if either edge is traversed. Predicates p4 and p6 must be OR-type in the example. In reality, OR-type predicates could be used exclusively for if-conversion. However, OR-type predicates require explicit clearing for proper use. With unconditional predicates, explicit clearing is not required; thus, they are used whenever possible to reduce the number of necessary clears.

To illustrate the insertion of predicate comparison instructions, consider the calculation of predicate p4, which is the predicate for block G. The control dependence set for block G is $\{\overline{brE}, \overline{brF}\}$; thus, comparison instructions must be placed in blocks which originate the control dependence edges, namely, blocks E and F. From Figure 7.2, the compare conditions are derived from the conditional branches which terminate these blocks. Therefore, both comparisons will utilize a *pge* instruction to correspond with the *bge* instruction. The final code after if-conversion, presented in Figure 7.8, shows the two *pge* instructions which define predicate p4 as OR-type. Note also that OR-type predicates require explicit clearing before they are defined or referenced. Thus, the *pclr* instruction is placed at the top of the hyperblock.

If-conversion is completed by associating instructions in each basic block of the hyperblock with the appropriate predicate and subsequently removing all internal control flow. The predicates of each instruction are derived directly from their original basic block and the predicate assignment given in Figure 7.7. For example, the *ld_i*, *add*, and *st_i* instructions originally in block I, are conditioned under predicate p7 in the final hyperblock code. When control flow is removed, both conditional and unconditional branches are eliminated. In the final hyperblock for the *wc* example, all but three branches are removed. The remaining branches are the two infrequent branches which exit the hyperblock (highlighted in Figure 7.8) and an unconditional loop-back branch at the bottom of the hyperblock.

7.2 Extending Superblock Techniques to Hyperblocks

Hyperblock formation is only the first step in the hyperblock compilation techniques. The hyperblocks created by the formation procedure require subsequent ILP enhancing and exposing transformations to be applied. For example, loop unrolling, register renaming, and induction variable expansion are regularly applied to hyperblock loops. Aggressive scheduling and register allocation are also necessary to realize any of the performance potential. This closely mirrors the scenario of superblock formation and subsequent superblock compilation techniques. To accomplish the necessary compiler support, all of the superblock techniques in the IMPACT compiler are extended to operate on hyperblocks as the basic compilation unit. The major difference between hyperblocks and superblocks is that program control flow is not completely represented by branches in hyperblocks. Rather, it is represented with a combination of branches and predicates. Therefore, the compiler must be adopted to understand the meaning and relationships of predicates in order to efficiently perform transformations in the presence of predicates.

A block diagram of the backend hyperblock compilation path is presented in Figure 7.9. The diagram shows the phase ordering of the backend compilation steps. Predicates are generated early in the backend compilation procedure, during hyperblock formation. After hyperblock formation, all subsequent parts of the compiler operate on predicated code. The information regarding the relationships among predicates is communicated to the compiler via two modules: the predicate hierarchy graph (PHG) and the predicate control flow graph (CFG) generator. The PHG provides information regarding the relationships of predicates with other predicates. For example, it can answer the question are two predicates true under mutually exclusive conditions. The predicate CFG generator constructs a control flow graph that jointly represents the control flow indicated by branches and predicates. The resultant predicate CFG is then used by the dataflow analysis equation solver to provide the necessary predicate-sensitive dataflow information to the rest of the compiler.

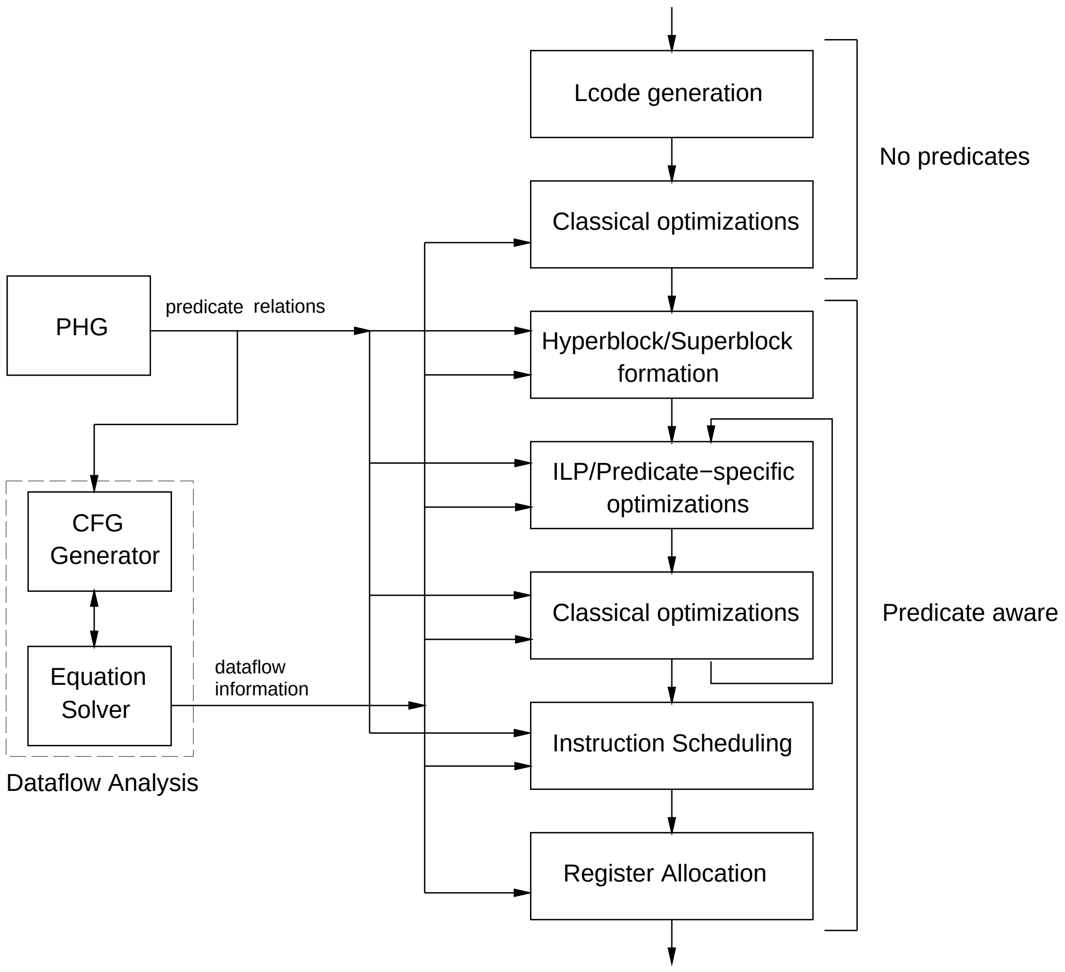


Figure 7.9 Block diagram of the backend compilation path with hyperblocks.

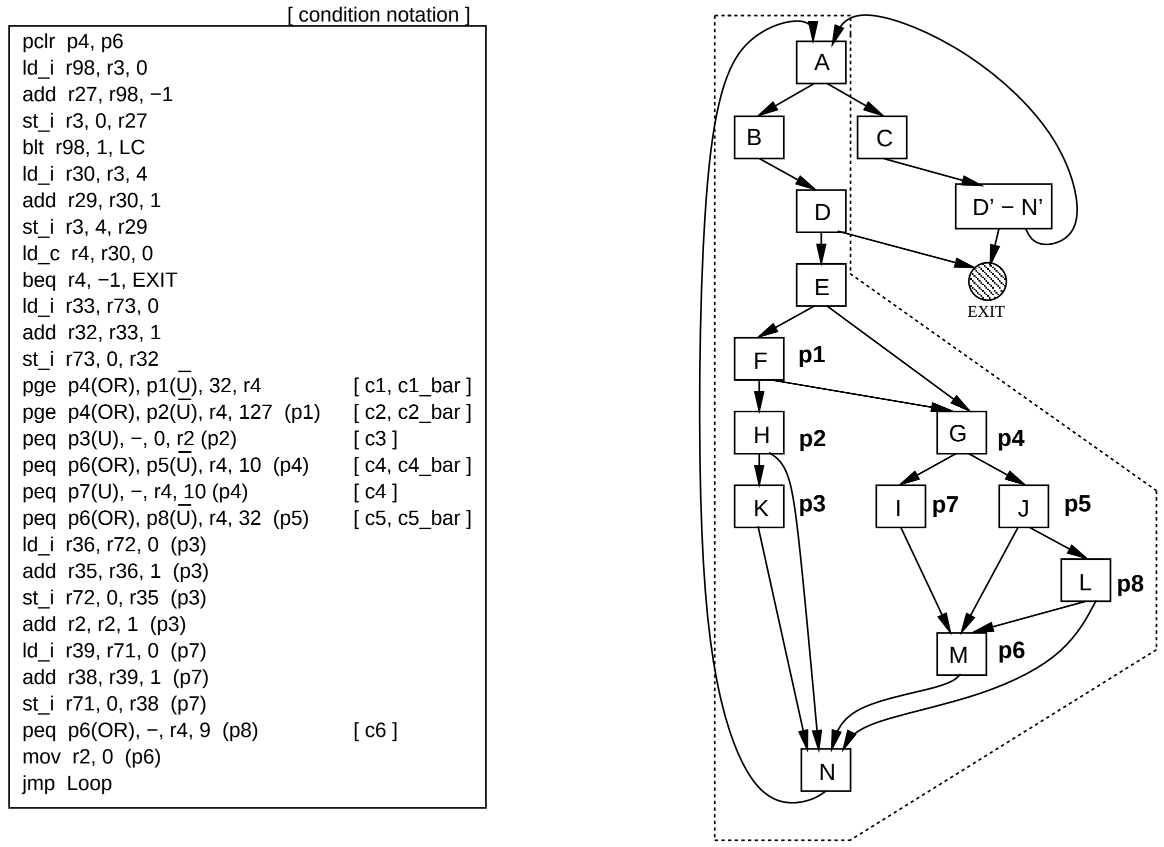
The remainder of this section discusses the details of the PHG and the dataflow analysis using the predicate CFG generator. In addition, the interfaces of these modules into the rest of the compiler are summarized.

7.2.1 Predicate hierarchy graph

The relationships among predicates are derived from a structure referred to as the PHG. A PHG is a directed acyclic graph that represents the Boolean equations used to compute all the predicates in a hyperblock [101]. The PHG is composed of two types of nodes, predicate and condition. Predicate nodes represent the predicates themselves. Therefore, there is a single predicate node in the PHG for each predicate defined in a hyperblock. Condition nodes represent the compare conditions used to compute the predicates. There are condition nodes in the PHG which correspond to each predicate comparison instruction in a hyperblock. Edges in the PHG represent the flow of values used to compute predicates and conditions. Two nodes are connected with an edge when the value specified by one node is used to directly compute the value in the other node. A PHG is generally rooted at the true predicate node, or 'T', since hyperblocks are constructed such that all chains of predicate computations start under the true predicate.

A PHG is constructed by initially creating a predicate node for the true predicate. Each predicate comparison instruction is then examined sequentially from the entry point of the hyperblock. The compare condition specifies up to two condition nodes that must be created. The condition nodes reflect the regular and the complement conditions of two target predicate comparison instructions. Edges are inserted to the new condition nodes from the predicate that is sourced by the predicate comparison. Next, predicate nodes are created for each destination of the predicate comparison if they do not exist already. Edges are then inserted between the appropriate condition nodes that are used to define predicates and the predicate nodes. This process is repeated until nodes and edges have been added to the graph for all predicate comparison instructions in the hyperblock. The final PHG is very regular in structure, containing alternating levels of predicate and condition nodes all starting from the true predicate node.

The PHG construction process is best illustrated with an example. The hyperblock loop from the benchmark *wc* that was used to demonstrate hyperblock formation will be utilized. The assembly code for the final hyperblock is presented in Figure 7.10(a). The predicate comparison



```
                                    [ condition notation ]
pclr  p4, p6
ld_i  r98, r3, 0
add  r27, r98, −1
st_i  r3, 0, r27
blt  r98, 1, LC
ld_i  r30, r3, 4
add  r29, r30, 1
st_i  r3, 4, r29
ld_c  r4, r30, 0
beq  r4, −1, EXIT
ld_i  r33, r73, 0
add  r32, r33, 1
st_i  r73, 0, r32
pge  p4(OR), p1(U), 32, r4              [ c1, c1_bar ]
pge  p4(OR), p2(U), r4, 127  (p1)       [ c2, c2_bar ]
peq  p3(U), −, 0, r2 (p2)               [ c3 ]
peq  p6(OR), p5(U), r4, 10  (p4)        [ c4, c4_bar ]
peq  p7(U), −, r4, 10 (p4)              [ c4 ]
peq  p6(OR), p8(U), r4, 32  (p5)        [ c5, c5_bar ]
ld_i  r36, r72, 0  (p3)
add  r35, r36, 1  (p3)
st_i  r72, 0, r35  (p3)
add  r2, r2, 1  (p3)
ld_i  r39, r71, 0  (p7)
add  r38, r39, 1  (p7)
st_i  r71, 0, r38  (p7)
peq  p6(OR), −, r4, 9  (p8)             [ c6 ]
mov  r2, 0  (p6)
jmp  Loop
```

Figure 7.10 Example hyperblock loop from *wc*, (a) assembly code for the final hyperblock, (b) control flow graph before if-conversion.

instructions are marked with a shorthand notation to represent the computed conditions. This notation, such as *c1* and its complement *c1_bar*, is used for all of the PHG construction examples in this section. To serve as a reference point, the CFG for the hyperblock before if-conversion is shown in Figure 7.10(b). The predicates assigned to each block are marked in the figure.

The PHG that is constructed from the example hyperblock is shown in Figure 7.11. Construction begins by creating a node for the true predicate, T, and inserting it into the graph. The first predicate comparison instruction is then examined. It computes conditions *c1* and *c1_bar* under predicate T. Thus, two condition nodes are created and inserted into the graph.

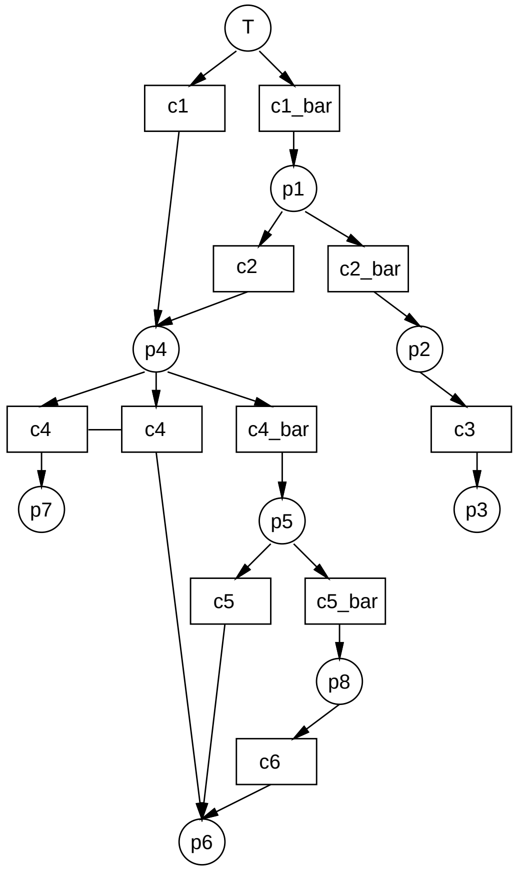


Figure 7.11 Predicate hierarchy graph for example hyperblock loop from *wc*.

In addition, edges are added between predicate T and the two condition nodes. Two predicates are also defined by this predicate comparison instruction, namely, *p4* and *p1*. Since nodes to represent these predicates do not already exist, they are created. An edge is inserted to connect predicate *p4* and condition *c1* because the predicate is set when the compare condition holds. On the other hand, predicate *p1* is a complement type that is computed under the opposite condition of predicate *p4*. Hence, an edge is added between predicate *p1* and condition *c1_bar*.

Construction continues by examining the second predicate comparison instruction. Again, two condition nodes are created and inserted into the graph to represent conditions *c2* and *c2_bar*. Edges are inserted between these condition nodes and predicate *p1* since the predicate comparison is predicated on *p1*. As with the first predicate comparison instruction, this comparison also defines two predicate values, *p4* and *p2*. However, a node already exists for predicate *p4*. Therefore, a new node is created only for predicate *p2*. To complete the procedure for the second predicate comparison instruction, edges are inserted to connect predicates *p4* and *p1* to conditions *c2* and *c2_bar*, respectively. The process continues through the remaining five predicate comparison instructions to construct the PHG presented in the figure. It should be noted that condition *c4* is computed twice in the hyperblock. This results in two identical condition nodes in the PHG. Such identical condition nodes can be later merged if desired. The final PHG distinctly shows the computation chains used to derive each predicate.

The purpose of the PHG is to provide an efficient structure for the compiler to derive relations among the predicates. Three important predicate relations have been identified for this dissertation. The relations are defined as Boolean on a pair of predicates.

(1) Ancestor - A predicate is an ancestor of another predicate if all conditions used to compute the predicate are derived directly or indirectly from the ancestor.

(2) Control path - Two predicates have a control path if there exists at least one set of conditions under which both predicates are jointly true.

(3) Implies - A predicate implies another predicate when the conditions for the first predicate being true guarantees the second predicate will also be true.

The meaning of the first two relations is relatively straightforward. The ancestor relation defines predicates from which other descendant predicates are completely derived. This relation is similar to most ancestor/descendant relations defined on graphs or trees. The one caveat is that an ancestor must be used to derive all conditions for the descendant predicate as opposed to a single condition in the case of a predicate defined with multiple conditions. With the ancestor/descendant relation, the compiler is certain the descendant may be true only when the ancestor is also true. Similarly, if the descendant has a value of 1, then the ancestor must be 1.

The control path relation identifies predicates which have overlapping conditions, so that they may be true at the same time. Essentially, the control path relation is the inverse of mutual exclusion. Two predicates which are mutually exclusive may not both have values of 1 for a given set of conditions. Note that mutually exclusive predicates may both be 0, but not both 1. Instructions conditioned on predicates which do not have a control path relation have no dependences to one another. Thus, the compiler can treat such instructions independently without any difficulties.

The implies relation is not as straightforward and occurs much less frequently in hyperblocks. A predicate implies another predicate when the condition(s) for a predicate being true also ensure that another predicate is true. A descendant predicate seemingly has this relationship with its ancestor. However, the implies relationship is not for predicates whose conditions are derived from another predicate as is the descendant/ancestor relation. Rather, the implies relation is that some condition or set of conditions for determining one predicate are also used for a second predicate. The implies relation generally involves one predicate that is defined if any of multiple conditions are true and a second predicate that is defined under one those conditions. In cases where the second predicate is known to be true, the first predicate is implied to be true.

Algorithms to compute each of the three relationships using the PHG are presented in Figures 7.12 – 7.14. The first algorithm computes the ancestor relationship among two predicates,

```
/* Determine if $p1$ is an ancestor of $p2$ */
compute_ancestor($p1$, $p2$) {
    $root$ = root node of the PHG
    $node1$ = predicate node representing $p1$
    $node2$ = predicate node representing $p2$
    for each path through the PHG from $root$ to $node2$, $cur\_path$ {
        if ($cur\_path$ does not contain $node1$)
            return (0)
    }
    return (1)
}
```

Figure 7.12 Algorithm to compute the ancestor relationship between two predicates.

p1 and *p2*. A value of 1 is returned if predicate *p1* is an ancestor of predicate *p2*. Otherwise, a value of 0 is returned. The ancestor relationship can be efficiently expressed as a dominator relationship in the PHG. A predicate is an ancestor of another predicate if its predicate node dominates the other predicate node. In a graph with a unique entry node, a node X is defined to dominate another node Y if all paths starting from the entry node to node Y go through node X [52]. Therefore, in the PHG, predicate *p1* is an ancestor of predicate *p2* if all paths in the PHG starting with predicate T leading to predicate *p2* pass through predicate *p1*. The algorithm in Figure 7.12 computes exactly this relationship.

Using this algorithm on the example PHG in Figure 7.11, the ancestors of predicate *p4* are predicates T and *p4*. Each predicate is always an ancestor of itself. Additionally, the root node, predicate T, is an ancestor of all predicates. For predicate *p4*, no other predicates satisfy the dominance constraints. Predicate *p1* occurs on one path from the root node to predicate *p4*. However, there is a path where it does not occur, so it is not an ancestor. As another example, the ancestors of predicate *p5* are predicates T, *p4* and *p5*.

```
/* Determine if there is a control path between $p1$ and $p2$ */
compute_control_path($p1$, $p2$) {
    $node1$ = predicate node representing $p1$
    $node2$ = predicate node representing $p2$
    visit $node1$ and all its predecessor nodes, marking visited nodes with $flag1$
    visit $node2$ and all its predecessor nodes, marking visited nodes with $flag2$
    let $merge\_nodes$ be the set of nodes which are encountered first along all
            paths starting from $node2$ with both $flag1$ and $flag2$ set
    for each predicate node in $merge\_nodes$, $cur\_merge$ {
        $mutually\_ex = 0$
        for each pair of edges leaving $cur\_merge$, $edgeX$ and $edgeY$ {
            let $condX$ = node pointed to by $edgeX$
            let $condY$ = node pointed to by $edgeY$
            if ((($condX.visit == flag1$) && ($condY.visit == flag2$)) ||
                    (($condX.visit == flag2$) && ($condY.visit == flag1$))) {
                $mutually\_ex$ = complement_conditions($condX$, $condY$)
                if (! $mutually\_ex$) return (1)
            }
        }
        if (! $mutually\_ex$) return (1)
    }
    return (0)
}
```

Figure 7.13 Algorithm to compute the control path relationship between two predicates.

The second algorithm, shown in Figure 7.13, computes the control path relationship among two predicates. The algorithm returns a value of 1 if the two predicates can be simultaneously true. Otherwise, a value of 0 is returned indicating the predicates are mutually exclusive. The control path relationship is calculated by traversing all paths backwards from each of the predicate nodes, *p1* and *p2*, to the root of the PHG. The set of nodes where paths from predicates *p1* and *p2* first intersect (*merge_nodes*) is then constructed. For each of the *merge_nodes*, the edges leaving the merge in the forward direction are analyzed. Predicates *p1* and *p2* are mutually exclusive if all pairs of forward leaving edges from the merge that were traversed during different backward traversals, lead to complementary conditions. Essentially, there may be no overlap of the conditions leading to predicates *p1* and *p2* at the *merge_nodes*. In the cases in which the edges lead to overlapping conditions or there is only a single edge leaving the merge, the two predicates are not mutually exclusive. Hence, they have a control path between them.

The calculation of the control path relation can be equivalently viewed as forming the Boolean expression for each predicate. If the logical AND of the two Boolean expressions can be simplified to 0, the predicates are mutually exclusive. Otherwise, there is a control path between the two predicates. Using the control path algorithm on the example PHG (Figure 7.11), the predicates with a control path to predicate *p4* are *T*, *p1*, *p4*, *p5*, *p6*, *p7* and *p8*. The remaining predicates, namely, *p2* and *p3*, are mutually exclusive with predicate *p4*. This is clear from the PHG because only predicate *p2* and its descendants are calculated under complementary conditions, namely, condition *c2_bar*. The control path relationship is also obvious if the control flow graph for the hyperblock before if-conversion is examined. In Figure 7.10(b), it is clear that there is no path of control from predicate *p4* (block G) to either predicates *p2* (block H) or *p3* (block K) without exiting the hyperblock.

As another example, the predicates with a control path to predicate *p5* are predicates *T*, *p1*, *p4*, *p5*, *p6*, and *p8*. This predicate has the same control path relationship as predicate *p4* with one exception. For this case, predicate *p7* is also mutually exclusive with predicate *p5*.

The final algorithm shown in Figure 7.14 computes the implies relationship between two predicates, *p1* and *p2*. A value of 1 is returned if predicate *p1* implies predicate *p2*. Otherwise, a value of 0 is returned. The implies relation is calculated using an algorithm similar to the previous control path algorithm. The algorithm examines all the conditions which are used to compute predicate *p1*. In general, if a duplicate condition node which directly computes predicate *p2* can be found for each condition used to compute predicate *p1*, the implies relation is established. This is accomplished by traversing backwards all paths starting from each

```
/* Determine if $p1$ implies $p2$ is true */
compute_implies($p1$, $p2$) {
    $node1$ = predicate node representing $p1$
    $node2$ = predicate node representing $p2$
    for each predecessor of node1, $prev\_node$ {
        visit $prev\_node$ and all its predecessor nodes, marking visited nodes with $flag1$
        visit $node2$ and all its predecessor nodes, marking visited nodes with $flag2$
        let $merge\_nodes$ be the set of nodes which are encountered first along all
                paths starting from $node2$ with both $flag1$ and $flag2$ set
        $implies = 0$
        for each predicate node in $merge\_nodes$, $cur\_merge$ {
            for each edge leaving $cur\_merge$ that leads to a node with $flag1$ set, $edgeX$ {
                $implies = 0$
                for each edge leaving $cur\_merge$ that leads to a node with $flag2$ set, $edgeY$ {
                    let $condX$ = node pointed to by $edgeX$
                    let $condY$ = node pointed to by $edgeY$
                    let $predY$ = successor predicate node to $condY$
                    if ((same_conditions($condX$, $condY$)) && ($predY == node2$)) {
                        $implies = 1$
                        break
                    }
                }
                if (! $implies$) return (0)
            }
            if ($implies$) break
        }
        if (! $implies$) return (0)
    }
    return (1)
}
```

Figure 7.14 Algorithm to compute the implies relationship between two predicates.

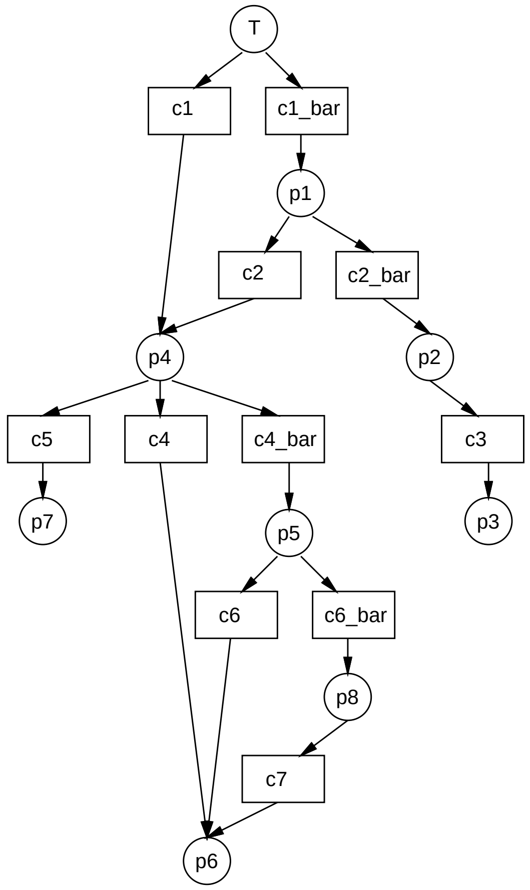
condition for predicate *p1* and predicate *p2* itself to the root of the PHG. The set of nodes (*merge_nodes*) that is the first intersection of these backward traversals is then constructed.

For each of the *merge_nodes*, the edges and target nodes are analyzed to determine if an implies relation exists. Predicate *p1* will imply predicate *p2* if, for each edge visited during the predicate *p1* traversal, there is a corresponding edge visited during the predicate *p2* traversal which leads to an identical condition node. In addition, the predicate computed by the condition node on the predicate *p2* traversal must be predicate *p2*. The algorithm is conservative by its formulation in that the corresponding duplicate condition for predicate *p2* must directly compute predicate *p2* for the implies relation to be detected. A more sophisticated algorithm is required to capture more general implies relations.

In the example PHG shown in Figure 7.11, a single implies relation exists. Predicate *p7* being true implies that predicate *p6* will be true. This relation holds because predicate *p6* is an OR-type predicate which is computed on two conditions. One of these conditions (condition *c4*) is also used to compute a different predicate, namely, predicate *p7*. In all cases, if predicate *p7* is true, predicate *p6* will also be true. Note that the opposite implies relation, predicate *p6* implies predicate *p7*, does not hold. Predicate *p6* may be true if either conditions *c4* or *c4_bar* are true, whereas, condition *c4* must hold for predicate *p7* to be true.

The final PHG for the example hyperblock loop from *wc* with all predicate relations enumerated is presented in Figure 7.15. For each predicate, those predicates which have control path, ancestor, and implies relations are shown.

One obvious alternative strategy to utilizing the PHG would be to analyze the CFG just before if-conversion to construct predicate relations. This information could then be maintained throughout the compiler backend to supply the predicate relations. Many of the necessary analyses are well-understood in the CFG area and could be accomplished in a straightforward manner. This approach was considered and not chosen for several important reasons. First, all compiler transformations that may affect the relations among predicates would have to incrementally update the predicate relation information. Second, any compiler transformation which introduces new predicates would also have to postulate the relations of the new predicates with those existing predicates. Finally, the CFG prior to if-conversion may not always be



(a)

p1	control: T, p1, p2, p3, p4, p5, p6, p7, p8
	ancestor: T, p1
	implies: none
p2	control: T, p1, p2, p3
	ancestor: T, p1, p2
	implies: none
p3	control: T, p1, p2, p3
	ancestor: T, p1, p2, p3
	implies: none
p4	control: T, p1, p4, p5, p6, p7, p8
	ancestor: T, p4
	implies: none
p5	control: T, p1, p4, p5, p6, p8
	ancestor: T, p4, p5
	implies: none
p6	control: T, p1, p4, p5, p6, p7, p8
	ancestor: T, p4, p6
	implies: none
p7	control: T, p1, p4, p6, p7
	ancestor: T, p4, p7
	implies: p6
p8	control: T, p1, p4, p5, p6, p8
	ancestor: T, p4, p5, p8
	implies: none

(b)

Figure 7.15 Predicate hierarchy graph for example hyperblock loop from *wc*, (a) graph itself, (b) enumeration of all predicate relations.

available for a given compilation framework, such as binary translation. The PHG approach overcomes these problems by deriving the predicate relations directly from the hyperblock code.

7.2.2 Dataflow analysis using the predicate CFG generator

The second mechanism for providing predicate relationship information to the compiler is with a module referred to as the predicate CFG generator. The predicate CFG generator constructs a CFG for a hyperblock that jointly represents the control flow produced by branches as well as the predicates. The implicit control flow that is derived from certain predicates evaluating to true and others evaluating to false is made explicit in the predicate CFG. The resultant graph can then be analyzed to obtain predicate sensitive information without considering the predicates of instructions. The major use of the predicate CFG is to serve as the underlying structure for dataflow analysis. With this approach, standard dataflow analysis techniques are utilized to analyze hyperblocks.

A predicate CFG is constructed by utilizing the concept of predicate covering. Conceptually, a predicate evaluates to true under some set of conditions. The predicate is covered by a group of predicates whose collective conditions subsume those of the original predicate. Hence, the predicate may not evaluate to true unless one or more of the covering predicates are true.

Using the concept of predicate covering, an instruction-level graph that represents the possible flows of instruction execution through the hyperblock may be constructed. Instructions are considered executed in this discussion only if their predicate evaluates to true. Hence, this instruction-level graph, the predicate CFG, can be used to enumerate all the possible instruction execution sequences in a hyperblock. Simplisticly, every instruction in a hyperblock could be chained to each of its successors. The resultant predicate CFG is correct, but is highly conservative because it contains many execution sequences which cannot occur. A much more precise predicate CFG may be constructed using predicate covering. The set of instructions that can be executed immediately after a particular instruction are the first set of the sequential successors whose predicates form a predicate covering. Other instructions may be subsequently executed, but they will be executed after one or more of the immediate successors. The set of immediate successors for each instruction define the connection points for the predicate CFG.

```
/* Defn: A predicate node is covered if any of the following conditions hold:
    1) It is marked visited
    2) All its predecessor condition nodes are visited
    3) Complementary successor condition nodes are visited */

/* Mark node and all nodes which are implicitly covered in the PHG */
mark_covered_nodes(node, target_node) {
    if (node.visit == 1) return
    node.visit = 1
    if (node == target_node) return
    /* Predicate node: visit all predecessor and successor condition nodes of node */
    if (node is a predicate node) {
        for each predecessor of node, pred_node
            mark_covered_nodes(pred_node, target_node)
        for each successor of node, succ_node
            mark_covered_nodes(succ_node, target_node)
    }
    /* Condition node: visit all predecessor and successor predicate nodes of node if they are
        covered, visit sibling condition nodes of node which compute the same condition */
    else {
        pred_node = predecessor predicate node of node
        if ((pred_node.visit == 0) && (pred_node is covered))
            mark_covered_nodes(pred_node, target_node)
        for each successor condition node of pred_node, succ_node
            if ((succ_node.visit == 0) && (same_condition(succ_node, pred_node)))
                mark_covered_nodes(succ_node, target_node)
        succ_node = successor predicate node of node
        if ((succ_node.visit == 0) && (succ_node is covered))
            mark_covered_nodes(succ_node, target_node)
    }
}
```

Figure 7.16 Algorithm for marking the covered nodes in a predicate hierarchy graph.

Predicate covering is defined more precisely in terms of the predicate nodes in a PHG. A predicate is covered if any of the following are true: the predicate is specified as covered; all the predicate's predecessor condition nodes are covered; or two complementary successor condition nodes of the predicate are covered. Essentially, a predicate node is covered if there are a set of nodes in the PHG which collectively dominate or post dominate it.

The two algorithms that are used by the compiler to construct the predicate CFG for a hyperblock are presented in Figures 7.16 and 7.17. The primary algorithm is that given in Figure 7.17. The *visit* flag is used to specify the nodes in the PHG that are currently covered. The goal is to sequentially consider each instruction in the hyperblock from the target instruction, *instr*, until a set of instructions that establish a predicate covering is found. The "for" loop in the algorithm performs a sequential scan of the instructions starting at the target instruction. During the scan, any instruction conditioned on the same predicate or an ancestor predicate automatically covers the predicate of *instr*, so the search is terminated. Predicate covering is established by definition for an instruction with the same predicate. Similarly, an ancestor predicate always covers its descendants by recursively applying the definition of predicate covering. For these cases, the *succ_set* is updated with the current instruction and returned as the set of immediate successors. Note that any instructions already in *succ_set* remain in the set as valid immediate successors.

A second case which automatically terminates the search is that the current instruction's predicate is implied by the target instruction's predicate. With such an implies relation, the current instruction's predicate is computed on a superset of the conditions used to compute the target instruction's predicate. Therefore, if the implied predicate is covered, any predicate computed on a subset of conditions must also be covered. Overall, the three special cases (same, ancestor, or implied predicate) are handled by the general algorithm to be discussed. They are singled out as special cases for performance reasons. When none of the termination scenarios occur, the instruction must pass two tests before it is recorded as an immediate successor. First, the current instruction's predicate must have a control path to the target instruction's predicate. If the two predicates are mutually exclusive, the current instruction will not be a

```
/* Compute a set of instructions which are the immediate successors of instr */
compute_successors(hyperblock, instr) {
    succ_set = ∅
    pred = instr.predicate
    pred_node = PHG node corresponding to pred
    if ((instr is an unconditional branch) && (pred == true))
        return (∅)
    reset visit flag for all nodes in the PHG for hyperblock
    for each instruction sequentially after instr, cur_instr {
        cur_pred = cur_instr.predicate
        cur_pred_node = PHG node corresponding to cur_pred
        if ((cur_pred == pred) || (cur_pred is an ancestor of pred) ||
                (pred implies curr_pred)) {
            succ_set += cur_instr
            return (succ_set)
        }
        else if (cur_pred and pred are not on control path)
            continue
        else if (cur_pred_node.visit == 1)
            continue
        succ_set += cur_instr
        mark_covered_nodes(cur_pred_node, pred_node)
        /* All successors have been found when pred_node is covered */
        if (pred_node.visit == 1)
            return (succ_set)
    }
    succ_set += -1          /* add fall through path of hyperblock as a successor */
    return (succ_set)
}
```

Figure 7.17 Algorithm to compute the immediate successors for an instruction in a hyperblock.

successor. The second case for which the current instruction is not an immediate successor occurs when its predicate is already covered by the predicate of a prior instruction. In this case, because the current instruction is a successor but not an immediate successor, it need not be considered.

At this point in the algorithm, the current instruction is indeed an immediate successor. The remaining step is to mark the node in the PHG corresponding to the current instruction's predicate as covered. Then, the state change is propagated through the PHG to mark any additional nodes that have become covered as a result of the current instruction's predicate being covered. The algorithm *mark_covered_nodes*, presented in Figure 7.16, performs these actions. Covering is propagated by first setting the *visit* flag for the predicate node corresponding to the current instruction's predicate. Then, all the neighboring nodes are recursively examined to determine if further covering conditions are established. The recursion terminates when either the target node (predicate of the target instruction) is reached or the node itself is already covered.

The actions taken by the recursive visit of all neighboring nodes are broken down by the node type. For a predicate node, the search is propagated to all predecessor and successor condition nodes. For a condition node, its predecessor, successor, and sibling nodes are examined to determine if a predicate covering has been established. This is accomplished by simply applying the nontrivial covering definition; namely, a predicate node is covered if either all its predecessors are covered or complementary successors are covered. Any node which is newly covered expands the search to all its neighbors. At the end of the *mark_covered_nodes* algorithm, all predicates covered by the current set of immediate successors are marked as visited.

Execution next resumes in the *compute_successors* algorithm (Figure 7.17). All immediate successors have been found if the target instruction's predicate is covered. The loop iterates until this condition is reached. In the case where the hyperblock ends before the target instruction's predicate is covered, implicit control flow out of the bottom of the hyperblock may occur. Hence, the immediate successor list is augmented with an entry for the fall through path.

The application of the *compute_successors* algorithm to each instruction in a hyperblock provides the connection points for an instruction-level CFG that represents the implicit control flow among the instructions introduced by the predicates. Several mechanical steps are now

```
i1:   pclr  p4, p6
i2:   ld_i  r98, r3, 0
i3:   add  r27, r98, −1
i4:   st_i  r3, 0, r27
i5:   blt  r98, 1, LC
i6:   ld_i  r30, r3, 4
i7:   add  r29, r30, 1
i8:   st_i  r3, 4, r29
i9:   ld_c  r4, r30, 0
i10:  beq  r4, −1, EXIT
i11:  ld_i  r33, r73, 0
i12:  add  r32, r33, 1
i13:  st_i  r73, 0, r32
i14:  pge  p4(OR), p1(Ū), 32, r4
i15:  pge  p4(OR), p2(Ū), r4, 127  (p1)
i16:  peq  p3(U), −, 0, r2 (p2)
i17:  peq  p6(OR), p5(Ū), r4, 10  (p4)
i18:  peq  p7(U), −, r4, 10 (p4)
i19:  peq  p6(OR), p8(Ū), r4, 32  (p5)
i20:  ld_i  r36, r72, 0  (p3)
i21:  add  r35, r36, 1  (p3)
i22:  st_i  r72, 0, r35  (p3)
i23:  add  r2, r2, 1  (p3)
i24:  ld_i  r39, r71, 0  (p7)
i25:  add  r38, r39, 1  (p7)
i26:  st_i  r71, 0, r38  (p7)
i27:  peq  p6(OR), −, r4, 9  (p8)
i28:  mov  r2, 0  (p6)
i29:  jmp  Loop
```

(a)

Instruction	Immediate successor(s)
i1:	i2
i2:	i3
i3:	i4
i4:	i5
i5:	i6
i6:	i7
i7:	i8
i8:	i9
i9:	i10
i10:	i11
i11:	i12
i12:	i13
i13:	i14
i14:	i15, i17
i15:	i16, i17
i16:	i20, i29
i17:	i18
i18:	i19, i24
i19:	i27, i28
i20:	i21
i21:	i22
i22:	i23
i23:	i29
i24:	i25
i25:	i26
i26:	i28
i27:	i28, i29
i28:	i29
i29:	−

(b)

Figure 7.18 Example of the computation of immediate successors from *wc*, (a) assembly code for the hyperblock, (b) immediate successors for each instruction.

required to convert the instruction-level CFG to the desired predicate CFG. First, the control flow edges for the branches are added into the instruction-level CFG so that it contains both predicate and branch information. Next, the instruction-level graph is partitioned into basic blocks [52]. This step is not absolutely required as an instruction-level CFG could be utilized by a dataflow analyzer. However, this step is extremely important for efficiency to reduce the number of nodes in the graph. Finally, the CFG for the hyperblock is connected with the equivalent CFGs generated for the rest of the blocks in the function. The result is a complete CFG for a function body that represents both branch and predicate control flow.

The process of generating a predicate CFG is illustrated using the continuing example hyperblock from the benchmark *wc*. The computation of the immediate successors for each instruction is presented in Figure 7.18. In Figure 7.18(a), the assembly code for the hyperblock is shown again. Each instruction is marked with a tag, such as *i10*, for reference throughout the example. The application of the algorithm *compute_successors* to each instruction in the hyperblock produces the immediate successors given in Figure 7.18(b). The successors for the first 13 instructions are simply the next subsequent instructions since all the instructions are predicated under true. Note that at this point, the control flow associated with the branches is not considered. Therefore, a branch, such as instruction *i5*, has just instruction *i6* as a successor.

The first instruction with nontrivial successors is instruction *i14*. The algorithm *compute_successors* proceeds by sequentially considering instructions subsequent to instruction *i14*. The goal is to find the first set of instructions whose predicates cover the predicate of instruction *14*, namely, predicate true. The first instruction, *i15*, is conditioned on predicate *p1*. Predicate *p1* has none of the following relations with predicate true: identical, ancestor, or implies. Therefore, it does not satisfy any of the special cases which immediately terminate the search. Predicate *p1* also has a control path to predicate true and is not already covered, so instruction *i15* is indeed an immediate successor to instruction *i14*. The next step of the process is to utilize the *mark_covered_nodes* algorithm to mark predicate *p1* as covered and to propagate this state to neighboring nodes. The result of this step is that predicates *p1*, *p2*, and *p3* are marked as covered. This is clear from the PHG for the hyperblock shown in Figure 7.15(a), in which the cover state is propagated to all the descendants of predicate *p1*.

At this point, the overall goal of finding the set of instructions whose predicates cover predicate true has not yet been achieved. Therefore, the search continues to the next instruction, namely, instruction *i16*. This instruction is conditioned on predicate *p2* which is already marked as covered. Therefore, the loop is just iterated without modifying the *succ_set* or the PHG visit flags. The next sequential instruction is instruction *i17*, which is conditioned on predicate *p4*. Again, this predicate does not satisfy any of the special cases to terminate the search, so the algorithm continues. Predicate *p4* has a control path to predicate true, and it is not already covered. Hence, instruction *i17* is added to the immediate successor list. As in the previous case, the next step of the process is to invoke the *mark_covered_nodes* algorithm to mark predicate *p4* as covered and to propagate this state to neighboring nodes.

The current state of the PHG is that predicates *p1*, *p2*, and *p3* are covered. By marking predicate *p4* as covered, conditions *c1* and *c1_bar* are also covered. This in turn leads to predicate true becoming covered since two of its successor nodes which compute complementary conditions are covered. The overall result is the first set of instructions whose predicates cover predicate true is found. Thus, the *compute_successors* algorithm terminates and returns $\{i15, i17\}$ as the immediate successors for instruction *i14*. The same procedure is applied to the remaining instructions in the hyperblock to derive the immediate successors given in Figure 7.18(b).

The immediate successors for each instruction in the hyperblock provide the connection points to build an instruction-level CFG that represents all the implicit control flow between the instructions caused by the predicates. The resulting graph is shown in Figure 7.19(a). Instructions *i1* through *i12* are summarized in the first node for space reasons since they just have sequential control flow to the subsequent instruction. Although this hyperblock is

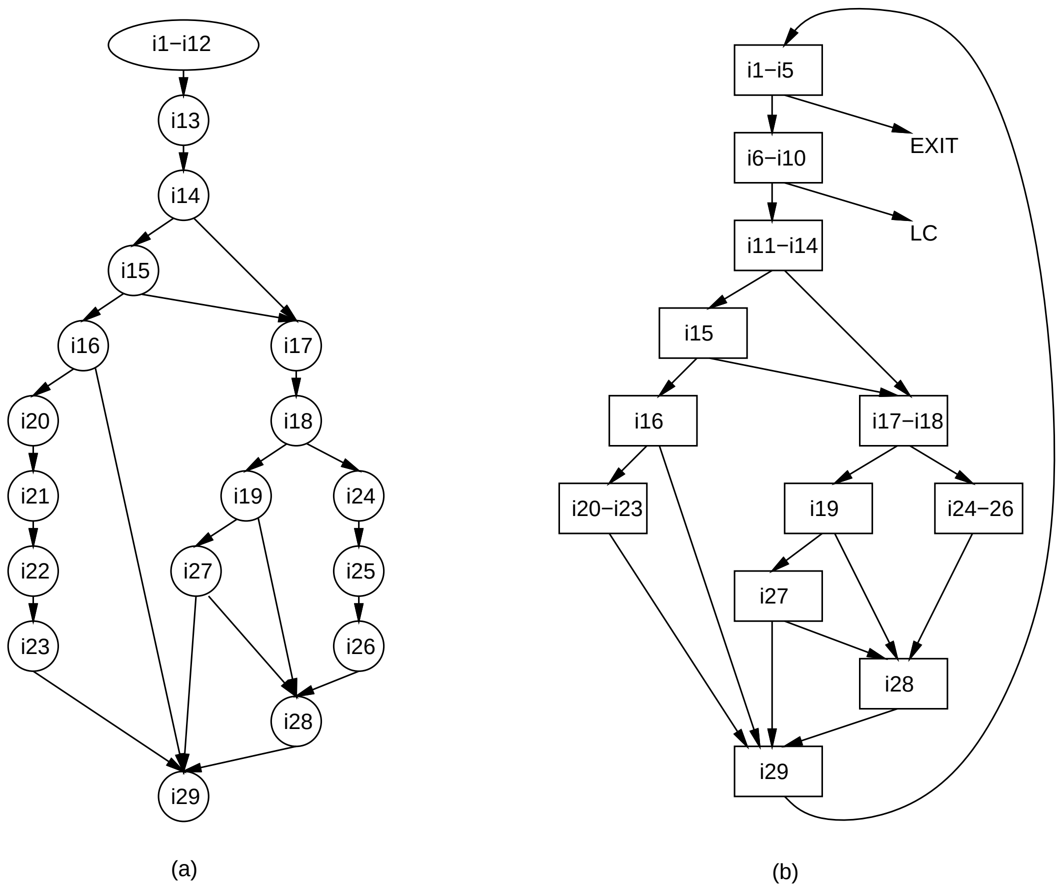


Figure 7.19 Example of predicate CFG generation from *wc*, (a) instruction-level CFG representing implicit control flow for predicates, (b) final predicate CFG for the hyperblock.

relatively small, the implicit control flow is rather complex. The next step of the procedure is to insert additional edges into the graph for the explicit control flow of the branches. The hyperblock has three branches, instructions *i5*, *i10* and *i29*. Therefore, an additional edge to represent the taken direction of each branch is added to the graph.

The final step is to partition the instruction-level graph into basic blocks to reduce the number of nodes the subsequent analyses will have to consider. The basic blocks are recognized from the instruction-level graph as simply the largest group of sequential instructions without any control flow splits or merges. For example, instructions *i20*, *i21*, *i22*, and *i23* form a basic block. The resultant predicate CFG for the example hyperblock is shown in Figure 7.19(b). Each node in the graph is a basic block with the specified instructions contained within it. Comparing the predicate CFG for this hyperblock with the original CFG before if-conversion (Figure 7.10(b)) shows that the graphs are indeed identical. In general, a correct predicate CFG may be derived for any hyperblock. However, the degree to which it matches the original CFG before if-conversion depends heavily on the compiler transformations applied after if-conversion.

The use of the predicate CFG approach has several advantages and disadvantages. The major advantage of the approach is that the introduction of predicates into the compiler does not force compiler analyses to be rewritten in a predicate cognizant form. Rather, conventional techniques can be applied to hyperblocks and to produce predicate-sensitive information. This is likely to be important both in terms of implementation cost as well as compile time. The predicate CFG allows the use of conventional techniques by logically separating predicates and their relations from the actual compiler analyses. It is the responsibility of the predicate CFG generator to produce a control flow graph which contains both predicate as well as branch control flow information. Subsequent analyses then operate on this graph without regard for the predicates of instructions.

The weakness of this approach is that the resultant predicate CFG is conservative under certain circumstances. As a consequence, the analysis results derived can also be conservative. Note that this does not mean that incorrect results are obtained by analyzing the predicate CFG. Rather, the analysis results may be not as precise as they could be which may cause the compiler to miss optimization, scheduling, or register allocation opportunities. The predicate CFG becomes conservative because certain hyperblock code sequences may artificially force the control flow graph to merge and re-split. By merging and re-splitting, correctness is maintained in that the possible implicit control flow is accurately represented. As a consequence though, additional control flow paths are added to the graph which cannot really occur. Thus, the predicate CFG is conservative because more control flow paths are represented than really exist. The problem arises when instructions with less constrained predicates are intermixed among instructions with more constrained predicates. This primarily occurs after instruction scheduling.

The problem is illustrated by the example in Figure 7.20. The example is a simple if-then-else statement which is fully if-converted to form a hyperblock. Figure 7.20(a) shows the original CFG before if-conversion. One code sequence for the hyperblock is given in Figure 7.20(b). For this sequence, there is no instruction intermixing that causes any difficulties. As a result, an accurate predicate CFG is generated, as shown in Figure 7.20(c). Note that the predicate CFG generator properly handles intermixing of predicates *p1* and *p2* as instruction *i6* occurs between two instructions conditioned on predicate *p1*. The problem arises in the example when an instruction conditioned on predicate True is intermixed with instructions conditioned on predicates *p1* and *p2*. The hyperblock code sequence in Figure 7.20(d) shows this case with instruction *i7* placed between instructions *i6* and *i4*.

The predicate CFG constructed for the second code sequence is presented in Figure 7.20(e). As shown, control flow must merge at instruction *i7*, since it is the only successor of both instructions *i3* and *i6*. Then, control flow is re-split into mutually exclusive paths to account for the next two instructions. The resultant predicate CFG contains four possible paths of control. However, there are really only two possible paths as indicated by the original CFG, making the graph conservative. By having extra paths of control, compiler analyses, such as dataflow analysis, blindly account for these paths which produce conservative results.

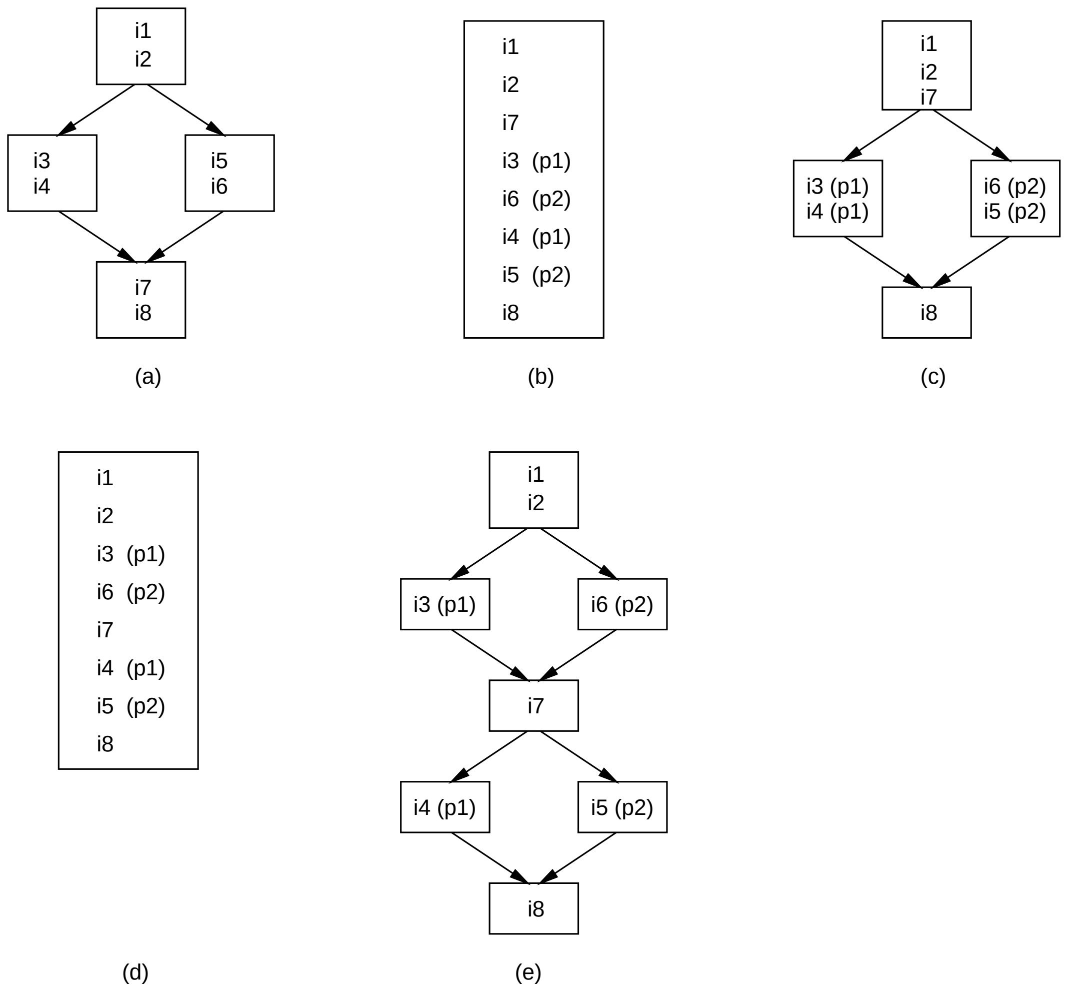


Figure 7.20 Example of an accurate and a conservative predicate CFG, (a) original CFG, (b) first example code sequence for the hyperblock, (c) accurate predicate CFG for the first hyperblock, (d) second example code sequence for the hyperblock, (e) conservative predicate CFG for the second hyperblock.

There are several possible strategies to achieve more accurate results. First, using the predicate CFG approach, some principles of scheduling with reverse if-conversion could be utilized [102]. In particular, replicating instructions with less constrained predicates in the predicate CFG would eliminate the need to merge control flow. For the previous example, instruction *i7* could be duplicated with one copy serving as the sole successor of instruction *i3* and the other serving as the sole successor of instruction *i6*. As a result, an accurate predicate CFG could be obtained even with intermixed predicates. Another strategy differs completely from the predicate CFG approach. Several researchers are exploring the area of directly analyzing predicated code [103],[104]. For this approach, the compiler analyses are modified to understand predicates and their relations. The accuracy of the analysis is only limited by the accuracy of the predicate relations.

7.2.3 Use of predicate information by the compiler backend

The backend modules of the IMPACT compiler have been enhanced to operate on hyperblocks using a combination of the predicate information provided by the PHG and the predicate CFG. The compiler modules affected by hyperblocks include the classical optimizer, ILP optimizer, instruction scheduler, and register allocator. A combination of predicate-sensitive dataflow information and direct predicate relation information is used by the modules to efficiently transform predicated code. A brief summary of the ways in which each class of modules utilizes the predicate information is provided in this section.

Classical/ILP optimization: The optimization modules pose the most difficulties in the process of extending the compiler backend components to operate on hyperblocks. One approach for the optimizations is to exclusively utilize the predicate CFG. In this manner, all hyperblocks would be converted into a graph of basic blocks which could then be optimized using standard techniques. However, there are two problems associated with this approach. First, the majority of optimizations which are local to a hyperblock cast as global optimizations in the predicate CFG. This is a serious problem for compilation speed in compilers such as IMPACT where global transformations are significantly more complex to perform than local transformations. The second reason is the structure of the IMPACT ILP optimizer. Most of the ILP optimizations operate on a single superblock or superblock loop as was presented in Section 3.3. Extending the ILP optimizations to operate on hyperblocks as the basic unit is much more natural than extending them to be global transformations.

The approach chosen is to extend all optimizations to directly operate on hyperblocks as the lowest level structure. Hence, hyperblocks are defined as the basic unit for local transformations. Global transformations are defined to be transformations among instructions in different hyperblocks. The classic and ILP optimizers utilize all of the predicate information that is provided by the PHG and the predicate CFG to various degrees. The predicate CFG is used to construct global dataflow analysis information to establish the correctness requirements at the hyperblock boundaries. Hyperblock transformations may not be performed if they violate the dataflow conditions when the hyperblock is exited. This restriction is the same regardless of the underlying structure for local optimization.

Within a hyperblock, the optimizers utilize the three predicate relations provided by the PHG. The ancestor and implies relations are used to establish a dominance relation between two predicated instructions in a hyperblock. Most classical optimizations, such as local copy propagation, are applied among two instructions in which the first instruction dominates the second [52]. In a basic block or a superblock, dominance trivially holds as the preceding instruction always dominates a subsequent instruction. This is true because basic blocks and superblocks are single entry blocks with linear control flow.

Hyperblocks are also single entry, but the predicates disrupt the simple ordering constraints for dominance. However, using the PHG, the dominance condition can be established by determining if either the ancestor or the implies relations hold among the predicates of the target instructions. If an instruction, A, precedes another instruction, B, A dominates B if either the predicate for A is an ancestor of the predicate for B or the predicate for A is implied by the predicate for B. Essentially, these rules are used to identify the cases in which an instruction is executed under the same or a more restrictive set of conditions of another instruction. Hence, whenever instruction B is executed, instruction A is guaranteed to have previously executed.

The control path relation provided by the PHG is also utilized by the optimizations. Optimization opportunities are generally lost when a particular register or expression is overwritten or killed. For optimizing memory references, the problem is more serious because any intervening memory instruction which potentially writes to a particular address takes away an optimization opportunity. In a basic block or superblock, all intervening instructions must be considered as potential optimization hazards, whereas with hyperblocks, the problem is less clear. Only instructions conditioned under predicates which have a control path to the instructions being transformed need to be considered as potential optimization hazards. Any instructions which are conditioned under mutually exclusive predicates can be ignored. Hence, the hyperblock optimizations use the control path relation to ignore instructions which are conditioned under mutually exclusive predicates as potential optimization hazards. Clearly, incorrect transformations would not be performed if the control path relation was not considered. However, a large fraction of the hyperblock optimization opportunities is lost if the relation is not utilized.

```
A:  mov   r1,r2 (p1)
B:  add   r2,r3,r4 (p2)
C:  ld_i  r5,r1,0 (p3)
```

Figure 7.21 Example use of the predicate relations to perform local copy propagation.

The use of the predicate relations to perform local copy propagation on a hyperblock is illustrated in Figure 7.21. For this example, predicate *p1* is assumed to be an ancestor of both predicates *p2* and *p3*. In addition, predicates *p2* and *p3* are mutually exclusive. The goal of copy propagation is to forward propagate the source operand of copy instructions into subsequent instructions which use the destination operand of the copy. For the example in Figure 7.21, a potential opportunity for a local copy propagation occurs between instructions A and C. However, the correctness of the transformation is not clear because of the predicates of the instructions.

A real opportunity is identified in the example because the predicate for instruction A is an ancestor of the predicate for instruction C. Hence, whenever instruction C is executed, the compiler knows that instruction A is previously executed. The optimization opportunity, though, is lost if there is an intervening instruction which over-writes the value in register *r2*. In this example, instruction B seemingly does this. But, the predicates for instructions B and C are mutually exclusive. Thus, the potential hazard does not affect the copy propagation, and it can be safely performed.

Instruction scheduling: Extending the superblock scheduler to operate on hyperblocks can be accomplished in a relatively straightforward manner. The use of predicate information in the hyperblock scheduling framework is localized to one step, dependence graph construction. After the dependence graph is in place, the remainder of the superblock scheduling process is

```
A: ld_i   r1,r2,r3 (p2)
B: add    r4,r1,4 (p2)
C: ld_i   r1,r5,0 (p3)
D: mul    r6,r1,r7 (p3)
```

Figure 7.22 Example use of the control path relation for hyperblock scheduling.

not altered for hyperblocks. The dependence graph construction utilizes two forms of predicate information. First, the predicate CFG provides predicate-sensitive dataflow information at the hyperblock exit points. As with superblock scheduling (see Section 3.4), instructions may not be moved above an exit branch if they define a register which is in the *LIVE-OUT* set of a branch. Hence, a dependence edge is placed between the defining instruction and the branch to ensure the proper ordering.

The second use of predicate information for dependence graph construction is to avoid unnecessary dependences. In particular, the control path relation provided by the PHG is utilized for this purpose. Two instructions which are conditioned under predicates that do not have a control path relation (mutually exclusive) may never both be executed. Therefore, there should be no ordering constraints among instructions conditioned by mutually exclusive predicates. This is true even if the instructions define and use common registers or memory locations. As a result, the dependence graph construction process makes extensive use of the control path relation to only put edges between instructions which have a control path relation between their predicates.

The code sequence shown in Figure 7.22 illustrates the point. As with the optimization example, assume that predicates *p2* and *p3* are mutually exclusive. Using a naive dependence construction procedure, all four instructions in Figure 7.22 are sequentially linked with dependences. There is at minimum a flow dependence between instructions A and B, an anti-dependence between instructions B and C, and a flow dependence between instructions C and D. In addition, depending on the processor model and the instruction latencies, there is also an output dependence between instructions A and C along with a flow dependence between instructions A and D.

Clearly, if predicates *p2* and *p3* are mutually exclusive, there should be no dependences between instructions conditioned by predicate *p2* and predicate *p3*. Using a dependence construction procedure that makes use of the control path relation, the number of dependences in the example is reduced to two, namely, flow dependences between instructions A and B along with instructions C and D. The remainder of the scheduling process takes full advantage of the reduced dependences to achieve a higher degree of overlap among instructions which are conditioned under mutually exclusive predicates.

Register allocation: The register allocator makes exclusive use of the predicate CFG to perform allocation of predicated code. Register live ranges are constructed by utilizing the predicate CFG for a function body which contains hyperblocks. The predicate CFG is a graph that represents control flow for both branches as well as predicates. Hence, traditional global live range construction techniques for a basic block level graph can be applied to the predicate CFG to derive live ranges which are predicate-sensitive. In essence, live range construction for register allocation is treated as just another global dataflow problem in the compiler. With predicate-sensitive live ranges constructed, the remaining register allocation process is unaltered by the presence of hyperblocks.

7.3 Predicate-Specific Optimizations

Up to this point in this chapter, the compiler support for predicated execution has focused on forming hyperblocks and extending superblock techniques to operate on hyperblocks. There are also many opportunities for new compiler optimizations targeted directly at improving the performance of predicated code. These new optimization techniques improve the efficiency of predicated code as well as performing new transformations that are made possible with conditional execution support. For this dissertation, a set of four important predicate-specific optimizations have been identified. The optimizations are predicate promotion, branch combining, predicated loop peeling, and instruction merging.

7.3.1 Predicate promotion

Speculative execution is an important source of ILP for superscalar and VLIW processors regardless of whether predicated execution is provided. By speculating instructions, a compiler can execute instructions before their condition for execution is completely known. With superblock compilation support, speculative execution was shown to be a major source of performance improvement. The ability to move instructions above branches which they depend on gave the scheduler substantially more freedom to achieve a compact schedule. With hyperblock compilation support, speculation comes in two forms. First, instructions can be moved above exit branches in the hyperblock. This form of speculation is the same as in the superblock domain. The second form of speculation in hyperblocks occurs in the predicate domain and is referred to as predicate promotion.

Predicate promotion advances the predicate of an instruction to an ancestor predicate [21]. The ancestor predicate is less constrained than the original predicate, meaning that it is computed using fewer conditions. As a result, the promoted instruction is executed under fewer conditions than the original program specified, making it a speculative instruction. The major advantage of predicate promotion is reducing the dependence height of the transformed hyperblocks. The critical dependence chains in hyperblocks frequently involve instructions awaiting the computation of their predicate. Subsequent predicate computations in turn await these instructions. With predicate promotion, dependences between predicate comparisons and predicated instructions are broken. The dependence is completely broken when the predicate of an instruction is advanced to True. Otherwise, the dependence height is lessened by connecting predicated instructions to predicate values that are available earlier. The overall result is that more compact schedules can be achieved through reduced dependence height and additional code motion freedom.

Three types of predicate promotion have been identified to handle the various opportunities to advance instruction predicates. The first type is the most trivial form of promotion and is aptly referred to as simple predicate promotion. Simple predicate promotion is utilized for instructions whose predicates are computed by a single predicate comparison instruction. In addition, no modifications to surrounding instructions nor insertion of new instructions are allowed by this transformation. An algorithm for simple predicate promotion is presented in Figure 7.23. The first four conditions in the algorithm identify potential promotion candidates. Condition 2 ensures that an instruction is only eligible for promotion if it writes its results into a destination register. As a result, instructions such as branches and stores, are not eligible for predicate promotion. The major fact that must hold to perform simple predicate promotion is expressed by conditions (5) and (6) in Figure 7.23. These conditions ensure that there is no useful value in the destination register of the candidate, *op(x)*, that would be overwritten

```
simple_predicate_promotion($hyperblock$) {
    for each instruction, $op(x)$, in $hyperblock$ {
        if all of the following conditions are true:
            1. $op(x)$ is predicated
            2. $op(x)$ has a destination register
            3. $op(x)$ has a speculative version
            4. there is a unique $op(y)$ lexically before $op(x)$ such that $dest(y) = pred(x)$
            5. $dest(x)$ is not live at $op(y)$
            6. $dest(j) \neq dest(x)$ in { $op(j), j = y + 1 \ldots x - 1$ such that there is a path
                    of control between $op(j)$ and $op(y)$ }
            7. it is profitable to promote $op(x)$
        then promote $op(x)$:
            1. set $pred(x) = pred(y)$
    }
}
```

Figure 7.23 Algorithm for simple predicate promotion.

if the candidate is promoted. Therefore, the candidate instruction's predicate may be safely advanced one level to its immediate ancestor.

Conditions (1) - (6) identify opportunities for a legal simple predicate promotion. The remaining condition which must hold is that there is some profit associated with the promotion. The costs associated with simple predicate promotion are exactly those of speculation in the control flow domain. The nominal speculation costs are primarily increased execution count with possible increases in register pressure and cache misses. In addition, a speculation model, such as sentinel speculation, may impose additional costs for promoting potentially excepting instructions. However, the general speculation model is assumed throughout this discussion, so there are no additional speculation costs. With the low overhead of simple predicate promotion, the profitability check is assumed to always hold in the current implementation. Thus, simple predicate promotion is applied to all cases which meet the legality requirements. In addition, simple predicate promotion is iteratively applied to continually advance instruction predicates until they either reach true or one of the conditions is not met.

```
multidef_predicate_promotion($hyperblock$) {
    for each instruction, $op(x)$, in $hyperblock$ {
        if all the following conditions are true:
            1. $op(x)$ is predicated
            2. $op(x)$ has a destination register
            3. $op(x)$ has a speculative version
            4. there exists more than one $op(y)$ lexically before $op(x)$ such that $dest(y) = pred(x)$
            5. $dest(x)$ is not live at the first instruction in $hyperblock$, $op(1)$
            6. $dest(j) \neq dest(x)$ in $\{ op(j), j = 1 \ldots x-1 \}$
            7. it is profitable to promote $op(x)$
        then promote op(x):
            set $pred(x) = True$
    }
}
```

Figure 7.24 Algorithm for multidefinition predicate promotion.

The second type of predicate promotion is very similar to simple predicate promotion, but is utilized for predicates defined by two or more instructions. Again, the transformation is restricted to perform no modifications to surrounding instructions nor insertion of new instructions. An algorithm for this type of promotion, referred to as multidefinition predicate promotion, is given in Figure 7.24. Multidefinition predicate promotion can be viewed as simultaneously performing simple predicate promotions of a candidate instruction to each of those instructions which compute its predicate. The difficulty, though, is choosing the appropriate predicate in which to advance the candidate instruction. To simplify the process, the true predicate is chosen as the fixed target for multidefinition predicate promotion. With this approach, the same conditions as for simple predicate promotion are used. Conditions (5) and (6) utilize the first instruction in the hyperblock as the reference point to ensure that no useful value in the destination register of the candidate is overwritten if the candidate is promoted to true. Clearly, this algorithm is conservative, but in practice it is found to work rather well.

```
renaming_predicate_promotion($hyperblock$) {
    for each instruction, $op(x)$, in the hyperblock {
        if all the following conditions are true:
            1. $op(x)$ is predicated
            2. $op(x)$ has a destination register
            3. $op(x)$ has a speculative version
            4. $op(x)$ cannot be promoted by simple predicate promotion
            5. it is profitable to promote $op(x)$
        then rename and promote $op(x)$:
            $new\_reg$ = new virtual register
            for each instruction, $op(y)$, lexically after $op(x)$ {
                if ($src(y) = dest(x)$) and ($op(x)$ is the only definition of $dest(x)$ to reach $op(y)$)
                    $src(y) = new\_reg$
            }
            $dest(x) = new\_reg$
            add new move instruction, $op(z)$, immediately following $op(x)$ to perform:
                    original $dest(x) = new\_reg$
            $pred(z) = pred(x)$
            $pred(x) = True$
    }
}
```

Figure 7.25 Algorithm for renaming predicate promotion.

The final type of predicate promotion is referred to as renaming predicate promotion. Renaming predicate promotion is introduced to overcome a major limitation of both simple and multidefinition predicate promotions. This limitation is that instructions cannot be promoted whenever their destination register is live along alternate control paths (violate either conditions 5 or 6 in Figures 7.23 or 7.24). Predicate promotion could be performed, though, if the destination register of the instruction is renamed to a new temporary register. In this manner, the live register value would not be corrupted by the promoted instruction. However, as a side effect of renaming the destination register, an additional copy instruction may be needed to ensure that subsequent instructions use the proper values.

An algorithm to perform renaming predicate promotion is presented in Figure 7.25. The conditions for renaming predicate promotion are much weaker than either of the other types of promotion. In general, all instructions which have a speculative version can be legally promoted. With this broad applicability and the need to insert a copy instruction, the profitability function becomes more important. A promotion is useful if it directly or indirectly leads to a more compact schedule for the hyperblock. However, in the IMPACT compiler all predicate promotion occurs before scheduling during the optimization phase, so an estimation of profitability is required.

The current profit function determines that a renaming predicate promotion should occur if the candidate has at least one flow dependence edge to another instruction in the hyperblock. However, there are two exceptions to this rule. First, if the candidate is a copy instruction itself, the promotion is not profitable. For this case, the compiler would just be replacing a predicated copy by a promoted copy and another predicated copy. Second, if the candidate overwrites one of its source operands, such as an increment instruction, the renaming promotion is not profitable. The reason for this exception is that other compiler transformations such as induction or accumulator expansion are much more effective for removing dependences on these instructions.

Renaming predicate promotion is accomplished in Figure 7.25 by creating a new virtual register, *new_reg*. All instructions subsequent to the candidate which are guaranteed to use its destination are modified to use the new register. This substitution allows the dependent instructions to be potentially hoisted along with the candidate during scheduling. After renaming the destination of the candidate to the new register, a predicate copy is inserted after the candidate. The copy conditionally moves the contents of the new register into the original destination of the candidate conditioned under the candidate's original predicate. Finally, the candidate is promoted to true.

To illustrate the application of predicate promotion, the example in Figure 7.26 is presented. The example is the most important loop segment for the benchmark *qsort*. The assembly code after hyperblock formation is given in Figure 7.26(a). The loop contains a single if-then-else statement which has been if-converted. In addition, the hyperblock contains two branches, the first of which exits the loop, and the second is the loop back branch. The schedule for the hyperblock loop on a processor with no resource constraints is shown in Figure 7.26(b). Note that all instructions are assumed to have a latency of one cycle except loads which have a two-cycle latency. A major problem in the hyperblock is that the predicated loads (instructions 4 and 8) cannot issue until cycle 3 because this is the earliest time their predicates are available. The delaying of the loads in turn causes subsequent instructions which depend on the loads to also be delayed.

A clear opportunity for predicate promotion exists in this example. By advancing the predicate of both instructions 4 and 8, the critical dependence chain in the hyperblock can be reduced. The hyperblock loop after predicate promotion is presented in Figure 7.26(c). The first load, instruction 4, can be promoted with a simple predicate promotion. The value in *r6* is not live at the point of instruction 4; therefore, the predicate for instruction 4 is just advanced. The new predicate of instruction 4 is the predicate of the instruction which computes *p126*, namely, the predicate of instruction 3 which is true.

The second load, instruction 8, cannot be transformed with simple predicate promotion. At instruction 8, there is a live value in *r6* that would be corrupted if the predicate of instruction 8 is simply advanced to true. As a result, renaming predicate promotion is required. To accomplish renaming predicate promotion, the destination of instruction 8 is renamed to a new register, *r60*, and the predicate is advanced to true. In addition, a copy instruction is inserted

```
 1  LA:  ld_i   r20, r14, r101
 2       ld_i   r23, r2, r102
 3       pge    p126(U), p127(Ū), r20, r23
 4  LB:  ld_i   r6, r123, 0  (p126)
 5       add    r123, r123, 8  (p126)
 6       add    r9, r9, 1  (p126)
 7       add    r101, r101, 8  (p126)
 8  LC:  ld_i   r6, r124, 0  (p127)
 9       add    r124, r124, 8  (p127)
10       add    r8, r8, 1  (r127)
11       add    r102, r102, 8  (p127)
12  LD:  st_i   r114, 0, r23
13       st_i   r114, 4, r6
14       add    r7, r7, 1
15       add    r114, r114, 8
16       bge    r9, r3, EXIT
17  LE:  blt    r8, r1, LA
```

(a)

```
 1  LA:  ld_i  r20, r14, r101
 2       ld_i  r23, r2, r102
 3       pge   p126(U), p127(Ū), r20, r23
 4  LB:  ld_i  r6, r123, 0
 5       add   r123, r123, 8  (p126)
 6       add   r9, r9, 1  (p126)
 7       add   r101, r101, 8  (p126)
 8  LC:  ld_i  r60, r124, 0
 8'      mov   r6, r60  (p127)
 9       add   r124, r124, 8  (p127)
10       add   r8, r8, 1  (p127)
11       add   r102, r102, 8  (p127)
12  LD:  st_i  r114, 0, r23
13       st_i  r114, 4, r6
14       add   r7, r7, 1
15       add   r114, r114, 8
16       bge   r9, r3, EXIT
17  LE:  blt   r8, r1, LA
```

(c)

cycle	issued instructions
0	1,2
1	−
2	3
3	4, 5, 6, 7, 8, 9, 10, 11
4	12, 14
5	13, 15, 16
6	17

(b)

cycle	issued instructions
0	1,2, 4, 8
1	−
2	3
3	5, 6, 7, 8', 9, 10, 11
4	12, 13, 14, 15, 16
5	17

(d)

Figure 7.26 Example of predicate promotion from *qsort*, (a) assembly code after hyperblock formation, (b) schedule for hyperblock, (c) assembly code after predicate promotion, (d) schedule for hyperblock after predicate promotion.

which moves the new register back to the original destination of instruction 8, *r6*, conditioned under the original predicate of instruction 8, *p127*. Subsequent uses of *r6* are guaranteed to receive the proper value by inserting the copy.

The resulting schedule after predicate promotion is shown in Figure 7.26(d). The overall schedule length has been reduced by one cycle. This improvement was accomplished because instructions 4 and 8 can be issued in cycle 0 after promotion. This compares with issuing them in cycle 3 in the original code. As a result, instruction 13 and the subsequent branches can all be executed a cycle earlier. Although this example seemingly only shows a modest performance increase, the actual performance gain with predicate promotion is substantial in the final hyperblock. The final hyperblock is unrolled eight times. With predicate promotion, the schedule length is reduced from 44 to 33 cycles, a 33% improvement. The reduction in dependence height and the increased code motion freedom enable the scheduler to achieve a significantly more compact schedule.

7.3.2 Branch combining

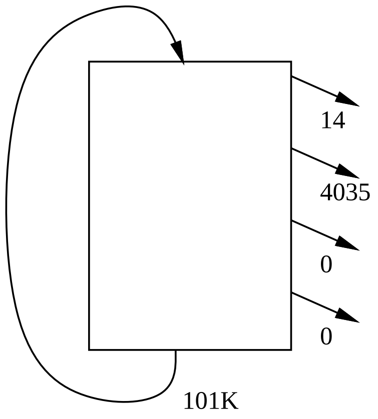
A common problem arising in hyperblocks is that they contain a large number of infrequently taken branches which exit the hyperblock. With limited branch resources, these exit branches often become the performance bottleneck. The exit branches are to handle execution sequences which transfer control flow to basic blocks which were not selected for inclusion in the hyperblock. These basic blocks typically correspond to handling infrequent execution scenarios, such as special cases, boundary conditions, and invalid input. In the *wc* example presented earlier in this chapter (see Figure 7.8), the hyperblock contained two exit branches. These exit branches handle the special cases of refilling the input buffer and detecting the end of the input file. In many cases, code segments contain a large number of these infrequent execution scenarios. Thus, the corresponding hyperblocks contain a large number of exit branches.

An example of such a hyperblock is the loop segment from the benchmark *grep* presented in Figure 7.27. The code segment consists of a loop body, where each iteration contains two memory, two ALU, and five branch instructions. The first four branches are exit branches which are taken very infrequently, as indicated by the execution frequencies in Figure 7.27(b). A small loop such as this is generally unrolled to increase the available ILP. In this example, the iterations of the loop are completely independent from each other. Thus, a high degree of instruction overlap can potentially be achieved. However, with four exit branches per iteration and few other instructions, the branch execution bandwidth to sustain four or eight instructions per cycle is quite large. For processors with limited branch resources, the resultant performance will likely be determined by the resource constraints. In this example unrolled twice and ignoring the loop backedge, the unrolled loop contains eight branches. For a processor which can execute at most one branch per cycle, the minimal schedule length for this hyperblock is $8\ branches \times 1\ cycle/branch$, or 8 cycles. With only 18 instructions in the loop body, the maximal performance that can be achieved is 2.25 instructions per cycle.

In this example, if-conversion alone was not sufficient for eliminating branches from the code. If-conversion failed because the cost of eliminating the branches was too large due to the instructions which would have to be included in the hyperblock. For these cases, the compiler can employ a transformation referred to as branch combining to eliminate exit branches from the hyperblock. Branch combining replaces a group of exit branches by a corresponding group of predicate define instructions. All of the predicate defines write into the same predicate register using the OR-type semantics. As a result, the resultant predicate will be set to 1 if any

```
1  A:  bge   r1, r5, EXIT1
2      ld_c  r3, r1, 0
3      beq   r3, 10, EXIT2
4      beq   r3, 0, EXIT3
5      bge   r2, r6, EXIT4
6      st_c  r2, 0, r3
7      add   r1, r1, 1
8      add   r2, r2, 1
9      jmp   A
```

(a)



(b)

Figure 7.27 Loop segment from *grep*, (a) assembly code, (b) weighted control flow graph.

of the exit branches were to be taken. Not exiting the hyperblock is the most common case, so the predicate will be false.

Branch combining is illustrated in Figure 7.28. Each of the exit branches, instructions 1, 3, 4, 5, 7, 9, 10, and 11 in Figure 7.28(a), is replaced by a corresponding predicate define instruction in Figure 7.28(b) based on the same compare condition. All predicate define instructions target the same predicate register, *p1*. The predicate is initially cleared, then each predicate define instruction sets *p1* if the corresponding exit branch would have taken in the original code. A single, combined exit branch (instruction 16) is then inserted which is taken whenever any of the exit branches were taken. The correct exiting condition is achieved by creating an unconditional branch predicated on *p1*. In cases where *p1* is false, the remainder of the unrolled loop is executed and the next iteration is invoked. In cases where *p1* is true and an exit branch was indeed taken, instruction 12 transfers control to the block labeled Decode. In this block, exit branches are re-executed in their original order to determine the branch which was originally taken. Since the conditions of multiple exit branches could be true, the first

```
 1  A:  bge   r1, r5, EXIT1
 2      ld_c  r3, r1, −1
 3      beq   r3, 10, EXIT2
 4      beq   r3, 0, EXIT3        Iter 1
 5      bge   r2, r6, EXIT4
 6      st_c  r2, −1, r3
 7      bge   r1, r7, EXIT5
 8      ld_c  r4, r1, 0
 9      beq   r4, 10, EXIT6
10      beq   r4, 0, EXIT7        Iter 2
11      bge   r2, r8, EXIT8
12      st_c  r2, 0, r4
13      add   r1, r1, 2
14      add   r2, r2, 2
15      jmp   A
```

(a)

```
 0   A:  pclr  p1
 1’      pge   p1(OR), r1, r5
 2       ld_c  r3, r1, −1
 3’      peq   p1(OR), r3, 10          Iter 1
 4’      peq   p1(OR), r3, 0
 5’      pge   p1(OR), r2, r6
 7’      pge   p1(OR), r1, r7
 8       ld_c  r4, r1, 0
 9’      peq   p1(OR), r4, 10          Iter 2
10’      peq   p1(OR), r4, 0
11’      pge   p1(OR), r2, r8
16       jmp   Decode         (p1)
 6’      st_c  r2, −1, r3
12       st_c  r2, 0, r4
13       add   r1, r1, 2
14       add   r2, r2, 2
15       jmp   A

     Decode:
 1       bge   r1, r5, EXIT1
 3       beq   r3, 10, EXIT2
 4       beq   r3, 0, EXIT3
 5       bge   r2, r6, EXIT4
 6       st_c  r2, −1, r3
 7       bge   r1, r7, EXIT5
 9       beq   r4, 10, EXIT6
10       beq   r4, 0, EXIT7
11       jmp   EXIT8
```

(b)

Figure 7.28 Example of branch combining from *grep*, (a) assembly code after unrolling twice, (b) assembly code after branch combining.

such branch has to be determined since that branch would have been taken in the original code sequence.

An important issue with branch combining is correct handling of instructions located between branches which are eliminated. In the original code, these instructions will not be executed if a previous exit branch is taken to transfer control out of the hyperblock. But in the transformed code, these instructions will be executed regardless because the actual control transfer out of the hyperblock does not occur until after the last exit branch. Instructions between combined branches are essentially speculated. However, there are often instructions which cannot be speculated between eliminated branches, such as stores.

To handle the nonspeculative instructions properly, two things are done. First, instructions which cannot be speculated are moved below the combined exit branch in the hyperblock. Second, they are replicated in the decode block and placed in their original position with respect to the exit branches. In the example in Figure 7.28(b), the first store, instruction 6, is handled in this manner. By positioning the store as such, it is guaranteed to execute exactly the same number of times as it did in the original code sequence. For most instructions which may be speculative, such as loads or arithmetic instructions, such transformations are unnecessary. Stores and instructions whose destination register is live along a prior exit branch are the most common instructions in the nonspeculative category.

A second issue with branch combining is the heuristics of when to apply the transformation and which branches should be combined. This profitability question arises because branch combining can create highly inefficient hyperblocks when applied blindly. The major negative of branch combining is an extra level of branch indirection that is added whenever an exit branch is taken. For the case in which an exit branch is taken, the combined exit branch is first executed to transfer control to the decode block. Then, the actual branches are executed to determine the first exit condition which holds and transfers control to the correct exit point. Furthermore, the actual cycle at which the exit occurs with branch combining is typically much later than without branch combining. This occurs because the combined exit cannot be executed until all exit branch conditions are calculated. With a frequently taken exit or a group of low probability exits which collectively account to a significant exit frequency, performance will suffer.

To overcome these potential problems, the branch combining algorithm utilizes two thresholds to control the application of the transformation. First, individual branches are candidates for branch combining only if the fraction of time they are taken is less than a maximum fraction. This threshold rules out all but infrequently taken exit branches. The second threshold is a ceiling on the overall taken frequency of the combined branches. This threshold limits the number of consecutive branches which are combined by the sum of the taken probabilities. By limiting the number of branches which are combined, a balance of overhead when a hyperblock exit occurs and branch reduction in the main hyperblock can be maintained. In the current implementation, the individual branch threshold is set to 0.10 and the combined threshold to 0.25.

Overall, for an issue-8 processor which can execute at most one branch per cycle, the execution time of the example loop from *grep* is reduced from 584K cycles to 106K cycles with branch combining. This large performance increase is primarily due to the drastic reduction in dynamic branches for this loop. With branch combining, the number of dynamic branches in the hyperblock drops from 343K to to 21K.

7.3.3 Predicated loop peeling

A fundamental limitation of if-conversion is the inability to transform a cyclic control flow graph. Hyperblocks can be loops themselves by having an exit branch which targets the entry block of the hyperblock. This was the case for the *wc* example presented in Section 7.1. But, no internal branches which form cycles are allowed in regions identified for hyperblock formation. For many cases, this limitation is not a problem since the compiler generally wants to optimize and schedule loops as a single entity. Transformations, such as loop unrolling and register renaming, enable high levels of ILP to be extracted from hyperblock loops by overlapping the execution of independent iterations. One important exception is the loops which do not iterate frequently. For these loops, there are not enough iterations executed to achieve the desired level of ILP.

An effective approach is to overlap the execution of infrequently iterated loops with the code surrounding the loop. In this manner, the loop execution can be overlapped with more instructions to increase the ILP. Predicated loop peeling is introduced to enable such overlap. Loop peeling itself is a traditional transformation which unravels several iterations from the beginning or end of a loop. Typically, loop peeling is utilized by compilers to treat loops which have specialized code that is executed for a particular iteration. For example, some variable may be initialized in the first iteration and referenced by subsequent iterations. By peeling off the first iteration, the compiler can eliminate the special case code from the body of the loop.

Predicated loop peeling is an extension of the traditional transformation. With predicated loop peeling, the first several iterations of an infrequently iterated loop are peeled off. In general, the loop is peeled enough times so that the majority of invocations for the loop just execute the peeled code. The peeled iterations are conditioned by iteration predicates to enable execution of the proper number of iterations. Any peeled iterations which are not required for execution are just nullified by the predicate hardware. The peeled iterations appear as purely acyclic code; thus, they are merged with basic blocks before and after the loop to form a single hyperblock. A copy of the original loop body, referred to as the recovery loop, is also maintained to handle invocations of the peeled loop which iterate more times than the peel amount. A branch is placed after the last peeled iteration to test if more iterations are required. In the cases where additional iterations are required, control flow is transferred out of the hyperblock to the recovery loop to ensure correctness.

One of the major advantages of predicated loop peeling is the ability to exploit outer loop ILP with hyperblock techniques. For many cases, the code surrounding the peeled loop is itself another loop. By peeling the inner loop and thus converting it into sequential code, a hyperblock loop can be formed for the outer loop. The hyperblock consists of the selected blocks from the outer loop as well as the peeled iterations of the inner loop. This loop can then be effectively transformed with inner loop techniques, such as loop unrolling and register renaming, to enable high levels of ILP to be extracted from the outer loop. The example presented later in this section illustrates the performance potential of this approach.

Loop peeling of this form does not strictly require predicated execution to be performed. The compiler could perform the transformation purely in the control flow domain. However, two important advantages of the transformation are not observed with this approach. First, predicates allow the compiler to merge the peeled iterations and the surrounding code into a single hyperblock structure. After this point, an unmodified hyperblock optimizer and scheduler take full advantage of the resultant peeling, whereas without predicated execution, a complex control flow graph consisting of the peeled loop and the surrounding code remains which must be optimized and scheduled in some manner for ILP. A second advantage of utilizing predicates for loop peeling is the ability to remove the mispredictions associated with loop back branches. For a loop which does not iterate frequently, the loop back branch is a major source of mispredictions. With predicated loop peeling, the compiler converts the loop back branches for each peeled iteration into predicate comparisons. The defined predicates are then used to properly condition subsequent instructions. When peeling is performed purely in the control flow domain, no branches are eliminated. Thus, the peeled code does not reduce the number of mispredictions.

The major tradeoff involved with predicated loop peeling is that of coverage versus instruction overhead. On the one hand, the loop should be peeled enough times so that execution rarely requires more than the peeled iterations. Whenever more iterations are required, performance is lost because the peeled hyperblock is exited to enter the recovery loop. On the other hand, each time the loop is peeled, more instructions are inserted into the hyperblock. The processor resources will become over-saturated when the loop is peeled too many times. The loop peeling and hyperblock formation algorithms utilize these tradeoffs to identify opportunities. The details of the peeling heuristics and the algorithms for transformation are not presented here. The interested reader is referred to [105] for a complete description of predicated loop peeling.

To illustrate the application and effectiveness of predicated loop peeling, the example presented in Figure 7.29 is utilized. The figure shows a nested loop segment from the function *elim_lowering* in *008.espresso*. This loop segment is among the most frequently executed in the benchmark. The assembly code and weighted control flow graph for the loop segment are shown in Figure 7.30. The entire loop nest has the seven basic blocks shown, with the inner loop consisting of blocks C and D. From Figure 7.30, two important facts can be determined. First, the inner loop does not iterate frequently; on average, the loop body is executed 2.6 times. Second, the inner loop contains only nine instructions, and is dominated by a five-cycle dependence chain. With the combination of these features, very little ILP can be extracted from the inner loop by itself. The compiler achieves an average of approximately two instructions

```
            /* OUTER LOOP */
            for (p = CC−>data, last = p + CC−>count * CC−>wsize;
LG:              p < last;
                 p += CC−>wsize)
               {
LA:                if (( p[0] & (0x2000) ))
                      {
LB:                      register int i_ = ( p[0] & 0x03FF );
                         /* INNER LOOP */
                         do {
LC:                            if ( p[i_] & ~r[i_])  break;
LD:                         } while(−−i_ > 0);
LE:                      if ( i_ != 0 )
                            goto false1;
                         continue;
                         false1:
LF:                      CC−>active_count−−,  ( p[0] &= ~(0x2000));
                      }
               }
```

Figure 7.29 Example loop segment from the function *elim_lowering* in *008.espresso*.

per cycle by unrolling and transforming the inner loop. The inner loop will therefore be peeled to enable the compiler to overlap the inner loop with the surrounding outer loop instructions.

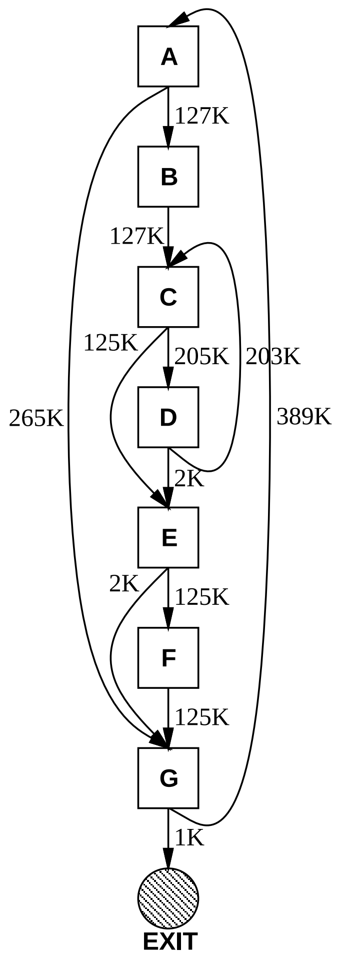
The predicated loop peeling transformation in conjunction with hyperblock formation is illustrated in Figure 7.31. For this example, the inner loop is peeled twice. The figure shows the assembly code and control flow graph for the resultant hyperblock. The entire loop nest is combined into a single hyperblock, consisting of blocks from the outer loop and the peeled inner loop. After the second peeled iteration is the branch (instruction 25) which checks if there are additional iterations. If there were more than two iterations required for the inner loop, the branch is taken and the recovery loop, the block labeled "RECOV", is entered. This code segment contains a copy of the inner loop to execute the remaining iterations. In addition, the code subsequent to the inner loop which was merged into the hyperblock (blocks E, F, and G) is tail duplicated.

```
LA:  ld_i  r98, r1, 0
     and   r99, r98, 8192
     beq   r99, 0, LG
LB:  and   r11, r98, 1023
     lsl   r115, r11, 2
     add   r124, r115, r1
     add   r125, r115, r2
LC:  ld_i  r56, r124, 0
     add   r124, r124, −4
     ld_i  r58, r125, 0
     add   r125, r125, −4
     xor   r59, −1, r58
     and   r60, r56, r59
     bne   r60, 0, LE
LD:  add   r11, r11, −1
     bgt   r11, 0, LC
LE:  beq   r11, 0, LG
LF:  add   r137, r137, −1
     and   r64, r98, −8193
     st_i  r1, 0, r64
LG:  add   r1, r1, r101
     blt   r1, r3, LA
```

Register contents:

```
r1 = p                 mem(r124, 0) = p[i_]
r11 = i_               mem(r125, 0) = r[i_]
r101 = CC−>wsize       r137 = CC−>active_count
```

(a)



(b)

Figure 7.30 Example loop segment from *008.espresso*, (a) original assembly code, (b) weighted control flow graph.

```
 1  LA:  ld_i   r98, r1, 0
 2       and    r99, r98, 8192
 3       pne    p130(U), −, r99, 0
 4       and    r11, r98, 1023        (p130)
 5       lsl    r115, r11, 2          (p130)
 6       add    r124, r1, −4          (p130)
 7       add    r125, r2, −4          (p130)─┐
 8       ld_i   r56, r115, r1         (p130) │
 9       add    r124, r124, r115      (p130) │
10       ld_i   r58, r115, r2         (p130) │
11       add    r125, r125, r115      (p130) │
12       xor    r59, −1, r58          (p130) │  PEEL 1
13       and    r60, r56, r59         (p130) │
14       peq    p134(U), −, r60, 0    (p130) │
15       pgt    p135(U), −, r11, 1    (p134) │
16       add    r11, r11, −1          (p134)─┤
17       ld_i   r56, r124, 0          (p135) │
18       add    r124, r124, −4        (p135) │
19       ld_i   r58, r125, 0          (p135) │
20       add    r125, r125, −4        (p135) │
21       xor    r59, −1, r58          (p135) │  PEEL 2
22       and    r60, r56, r59         (p135) │
23       peq    p136(U), −, r60, 0    (p135) │
24       add    r11, r11, −1          (p136) │
25       bgt    r11, 0, RECOV         (p136)─┘
26       pne    p133(U), −, r11, 0    (p130)
27       add    r137, r137, −1        (p133)
28       and    r64, r98, −8193       (p133)
29       st_i   r1, 0, r64            (p133)
30       add    r1, r1, r101
31       blt    r1, r3, LA
```

Predicate association:

p130 ~ B, Peel 1 − C, E	p136 ~ Peel 2 − D
p134 ~ Peel 1 − D	p133 ~ F
p135 ~ Peel 2 − C	

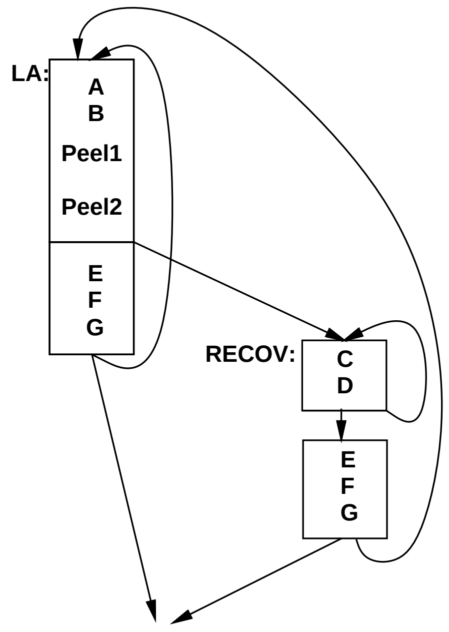


Figure 7.31 Example loop segment from *008.espresso* after hyperblock formation with predicated loop peeling; the inner loop is peeled twice.

The assembly code for the hyperblock shows the generation and use of predicates by the transformation. The iteration predicates for the peeled loop are predicates *p130* and *p135*. These predicates determine whether each iteration is executed or not. The first iteration is executed whenever block B is executed; therefore, the first iteration predicate is assigned the same predicate as block B, namely, predicate *p130*. The second iteration is executed whenever the loop back branch would have taken in the original loop. For this example, instruction 15 computes the loop back condition and stores the result in predicate *p135*. The second iteration is then conditioned based on this predicate. The inner loop also has embedded control flow, namely, the branch at the bottom of block C. This branch is eliminated with standard if-conversion and introduces the second set of predicates in each peeled iteration, predicates *p134* and *p136*. The resultant structure looks much like a simple hyperblock loop with a single exit branch formed from an inner loop region.

The hyperblock in Figure 7.31 is still very sequential and is dominated by data dependences. To increase the ILP, the compiler applies the full set of hyperblock ILP transformations to the resultant structure. The compiler is able to treat the outer loop, peeled inner loop combination as a conventional hyperblock loop since it is not structurally different. More specifically, the loop is first unrolled, with subsequent register renaming, induction variable expansion, and accumulator variable expansion applied to the hyperblock. Additionally, predicate promotion is aggressively applied because it is an extremely important ILP enhancing transformation with peeled loops. The instructions in each peeled iteration are often limited by a chain of predicate computation instructions. In order to allow the scheduler to overlap the execution of these instructions with previous instructions, the predicate dependences are broken with promotion.

A portion of the final hyperblock after ILP transformations is presented in Figure 7.32. For space reasons, the figure shows just a single iteration of the outer loop, where in actuality the outer loop is unrolled six times. The loop iteration contains one less instruction than the loop prior to transformation. The major differences are the renaming and expansion transformations along with the predicate promotion that have occurred to reduce the dependence height. Promoted instructions are indicated by an "$*$" in the predicate field for each instruction. From the figure, six of the nine instructions in each peeled iteration are promoted. The most important advantage of the renaming and promotion transformations is that the input operands for the

```
LA:  ld_i   r346, r157, 0                     * = Promoted Instruction
     and    r347, r346, 8192
     pne    p130(U), −, r347, 0
     and    r11, r346, 1023        *
     lsl    r348, r11, 2           *
     add    r349, r157, −4         *
     add    r350, r2, −4           *
     ld_i   r351, r348, r157       *       ┐
     add    r124, r349, r348       *       │
     ld_i   r352, r348, r2         *       │
     add    r125, r350, r348       *       │
     xor    r353, −1, r352         *       │ PEEL 1
     and    r354, r351, r353       *       │
     peq    p134(U), −, r354, 0    (p130)  │
     pgt    p135(U), −, r11, 1     (p134)  │
     add    r11, r11, −1           (p134)  ┤
     ld_i   r355, r349, r348       *       │
     add    r124, r124, −4         (p130)* │
     ld_i   r356, r350, r348       *       │
     add    r125, r125, −4         (p130)* │
     xor    r357, −1, r356         *       │ PEEL 2
     and    r358, r355, r357       *       │
     peq    p136(U), −, r358, 0    (p135)  │
     add    r11, r11, −1           (p136)  │
     bgt    r11, 0, RECOV          (p136)  ┘
     pne    p133(U), −, r11, 0     (p130)
     add    r161, r161, −1         (p133)
     and    r359, r346, −8193      *
     st_i   r157, 0, r359          (p133)
     blt    r1, r3, LA
```

Figure 7.32 Example loop segment from *008.espresso* after hyperblock formation with predicated loop peeling and ILP optimizations. This figure shows just one iteration of the outer loop. In the actual compiled code, the outer loop is unrolled six times to achieve performance.

Performance Summary (issue−8, 1 branch)

	No Peeling	With Peeling
Execution time:	6.70M cycles	2.46M cycles
Executed IPC:	1.83	6.59
Dynamic branches:	2.96M	884K
Dynamic mispredictions:	182K	16K

Figure 7.33 Performance summary for example loop segment from *008.espresso*.

predicate computations themselves can be scheduled earlier. As a result, the dependence chain threading through the predicate computations is substantially reduced.

The ILP transformations also allow the scheduler to achieve a high degree of instruction overlap across outer loop iterations. The outer loop does not have any cross iteration memory dependences in this example. Therefore, there is nothing to limit iteration overlap besides register dependences which the compiler can effectively eliminate with transformations. The overall result is a large increase in performance for this code segment with predicated loop peeling.

A summary of the performance for this loop nest on an issue-8 processor capable of executing at most 1 branch per cycle is given in Figure 7.33. The methodology utilized to generate these data is the same as that used in the overall evaluation of predicated execution in Chapter 8. The figure compares the best the compiler can achieve for the example with and without loop peeling. The performance change is drastic, the cycle count is reduced by nearly a factor of three. In correspondence, the average IPC is increased from 1.83 to 6.59. Clearly, predicated loop peeling has enabled the compiler to increase the ILP and translate the parallelism into overall performance improvement. A large portion of the performance improvement is due to the reductions in dynamic branches and dynamic mispredictions. From Figure 7.33, these are reduced by over three and ten times with peeling.

7.3.4 Instruction merging

The final predicate specific optimization discussed in this dissertation is referred to as instruction merging. Instruction merging differs from the previous optimizations in that it is primarily oriented towards improving the efficiency of the predicated code rather than increasing the ILP. Instruction merging combines two instructions in a hyperblock with complementary predicates into a single instruction which will execute under the union of the conditions. As a result, the number of instructions in a hyperblock is reduced. This technique achieves some of the same effects of partial redundancy elimination [53],[54],[55]. However, it uses the hyperblock boundaries to identify the paths along which redundancies are sought. Instruction merging is most effective for reducing the number of instructions for resource limited instruction classes, such as branches or stores. By merging these instructions, more compact schedules can often be achieved by reducing the resource pressure in a hyperblock.

To accomplish instruction merging, instructions that have the same source and destination operands along with equivalent opcodes are first identified. Then, if the same values for these operands reach both instructions, candidates for instruction merging are identified. The next step is to ascertain whether one of the instructions can be promoted to a new predicate which allows the other instruction to be eliminated. The new predicate that is required is the logical OR of the candidate instruction predicates. To simplify matters, only two scenarios are considered in the current implementation. First, the candidate predicates are mutually exclusive with a common ancestor predicate. In this case, the common ancestor predicate is used as the new predicate. The second scenario is that one of the candidate predicates is an ancestor of the other. For this case, the ancestor predicate itself serves as the new predicate.

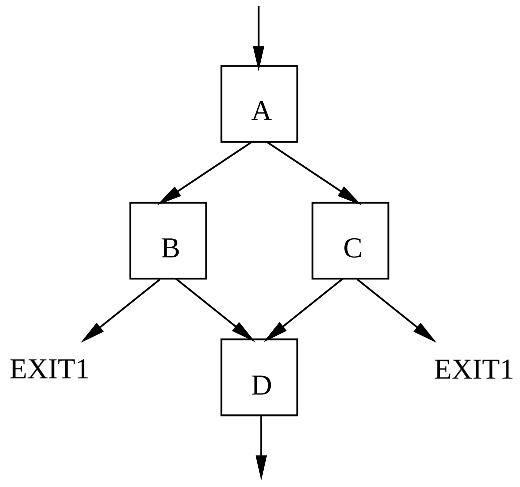
The application of instruction merging is illustrated by the example presented in Figure 7.34. This example is not directly from one of the benchmark programs, but is artificially created to resemble some of the more common application instances. The original assembly code and control flow graph before any hyperblock transformations are shown in Figures 7.34(a) and 7.34(b). The example is an acyclic code region with an if-then-else statement. On each side of the if-then-else are branches which exit the region. Figure 7.34(c) shows the code after hyperblock formation. All blocks are selected for the hyperblock; thus, the if-then-else branch is removed via if-conversion. The highlighted instructions show identical computations under mutually exclusive predicates, which are the candidates for instruction merging.

The code segment after instruction merging is shown in Figure 7.34(d). The transformation combines instructions 4 and 9 into instruction 4. In a somewhat different manner, instructions 7 and 12 are combined into instruction 12. Considering first the instruction merging of the load instructions, the second load is made redundant by promoting the first load to true. After promotion, the value of *r6* computed by instruction 4 can be utilized by all subsequent uses of *r6* under any predicate. Thus, the re-computation of *r6* by instruction 9 is unnecessary.

The second instruction merging cannot be accomplished in the same manner. Since the instructions are branches, promoting the first branch may cause the hyperblock to be prematurely exited. This can be seen by considering the case where *p31* is 1 and the hyperblock exit branch is taken. For this case, the hyperblock would be exited without executing instructions 10 and 11. To overcome this problem, the instruction merging transformation is reversed for

```
 1   LA:  ld_i   r20, r1, 0
 2        ld_i   r23, r2, 0
 3        blt    r20, r23, LC
 4   LB:  ld_i   r6, r4, 0
 5        add    r7, r6, 1
 6        st_i   r5, 0, r7
 7        beq    r6, 0, EXIT1
 8        jmp    LD
 9   LC:  ld_i   r6, r4, 0
10        sub    r6, r6, r20
11        st_i   r1, 0, r6
12        beq    r6, 0, EXIT1
13   LD:  add    r1, r1, 1
```

(a)



(b)

```
 1   LA:  ld_i   r20, r1, 0
 2        ld_i   r23, r2, 0
 3        pge    p30(U), p31(Ū), r20, r23
 4        ld_i   r6, r4, 0  (p30)
 5        add    r7, r6, 1  (p30)
 6        st_i   r5, 0, r7  (p30)
 7        beq    r6, 0, EXIT1  (p30)
 9        ld_i   r6, r4, 0  (p31)
10        sub    r6, r6, r20  (p31)
11        st_i   r1, 0, r6  (p31)
12        beq    r6, 0, EXIT1  (p31)
13        add    r1, r1, 1
```

(c)

```
 1   LA:  ld_i   r20, r1, 0
 2        ld_i   r23, r2, 0
 3        pge    p30(U), p31(Ū), r20, r23
 4        ld_i   r6, r4, 0
 5        add    r7, r6, 1  (p30)
 6        st_i   r5, 0, r7  (p30)
10        sub    r6, r6, r20  (p31)
11        st_i   r1, 0, r6  (p31)
12        beq    r6, 0, EXIT1
13        add    r1, r1, 1
```

(d)

Figure 7.34 Example of instruction merging, (a) original assembly code, (b) original control flow graph, (c) assembly code after hyperblock formation, (d) assembly code after instruction merging.

branches. The second branch is promoted allowing the first branch to be eliminated. By doing this, all instructions which require execution are physically located before the actual branch.

Overall, instruction merging has enabled two instructions to be eliminated in the example hyperblock. Most importantly, one of the instructions is a branch which can be very important for processors with limited branch resources. In general, the performance gains achieved with instruction merging are very small when compared with the other predicate specific optimizations. The importance of this transformation is expected to rise as the processors considered have more difficult resource constraints. In addition, if the inverse transformation, instruction splitting, is aggressively applied by the compiler, instruction merging may become extremely important to fuse predicated instructions back together.

CHAPTER 8

EXPERIMENTAL EVALUATION OF PREDICATED EXECUTION

The effectiveness of predicated execution using the hyperblock compilation techniques is evaluated in this chapter. Extensions to the methodology for the speculative execution experiments in Chapter 5 are first discussed. The experimental results are then presented. They include the effects of predicated execution on the overall performance, the instruction stream contents, the importance of speculation, and the instruction/data cache behavior.

8.1 Experimental Methodology

The experiments presented in this chapter utilize the same methodology as the previous experiments with several extensions. The extensions are focused on the additional support required for predicated execution. The two primary components of the experimental methodology that are affected by predicated execution are the processor model and the emulation capabilities.

8.1.1 Processor model

The processor model described in Section 5.1.3 is extended to support predicated execution. The extensions are those instruction set and microarchitecture extensions presented in Sections 6.2.2 and 6.2.3. The instruction set of the target processor is expanded to contain predicate comparison instructions, predicate clear/set instructions, and predicate save/restore instructions. To the microarchitecture, a nullification mechanism at the decode/issue pipeline stage is added along with a 64-entry predicate register file. The latencies of the predicate comparison and predicate clear/set instructions are assumed to have the same latency as the integer ALU instructions, 1 cycle. The predicate save/restore instructions are assumed to have the same latency as the conventional memory load or store instructions (Table 5.6). Finally, no restrictions are assumed on the combination of predicated or predicate defining instructions that may be issued each cycle.

8.1.2 Emulation of predicated execution

Emulation of predicated execution is achieved using the bit manipulation and conditional nullification capabilities of the HP PA-RISC processor. The 64 1-bit predicate registers are emulated by reserving two of the callee-saved integer registers and accessing them as 64 1-bit registers. Figure 8.1 shows an example of the PA-RISC assembly instructions used to emulate a sequence of predicated code. The predicated code segment (Figure 8.1(a)) is the code sequence from Figure 6.5(c). In this example, predicate registers *p1*, *p2*, and *p3* have been assigned bits 1,2, and 3 of general register *%r3*, respectively. Also, the values a, b, c, and d have been allocated to general registers *%r23*, *%r24*, *%r25*, and *%r26*, respectively.

Emulation of a predicate clear instruction (1) is achieved by using the deposit immediate PA-RISC instruction to write a 0 into the appropriate bit position of the general register. In this case, *p1* is assigned bit 1 of register *%r3*; therefore, that is cleared.

The instruction sequence required to emulate a predicate define instruction is dependent upon the predicate types of the destination predicate registers. Consider predicate define instruction (2) in Figure 8.1(a). This instruction is defining predicate register *p1* as OR-type and

```
(1) pclr   p1                            DEPI         0,1,1,%r3

(2) peq    p1(OR), p2(U̅), a,0            DEPI         0,2,1,%r3
                                         COMCLR,=     %r0,%r23,%r0
                                         DEPI,TR      1,2,1,%r3
                                         DEPI         1,1,1,%r3

(3) peq    p1(OR), p3(U̅), b,0 (p2)       DEPI         0,3,1,%r3
                                         BB,>=,N      %r3,2,$_ex_pred_0
                                         COMCLR,=     %r0,%r24,%r0
                                         DEPI,TR      1,3,1,%r3
                                         DEPI         1,1,1,%r3
                                      $_ex_pred_0

(4) add    c,c,1 (p3)                    EXTRU,EV     %r3,3,1,%r0
                                         ADDI         1,%r25,%r25

(5) add    d,d,1 (p1)                    EXTRU,EV     %r3,1,1,%r0
                                         ADDI         1,%r26,%r26

              (a)                                  (b)
```

Figure 8.1 An example of predicated execution emulation, (a) target processor assembly code, (b) HP PA-RISC assembly code.

p2 as unconditional complement. This combination requires a sequence of four instructions as given in Figure 8.1(b). The first instruction in the PA-RISC code sequence places a 0 in bit 2 of register *%r3*, thereby clearing *p2*. The second instruction then performs the comparison. The conditional nullification capabilities are used to execute either the third or the fourth instruction. If the contents of *%r23* are 0, the comparison nullifies the third instruction and only the fourth instruction is executed, writing a 1 to bit 1 of *%r3* (*p1*). Otherwise, the third instruction will be executed, writing a 1 to bit 2 of *%r3* (*p2*). Additionally the third instruction unconditionally nullifies the execution of the next instruction, so the fourth instruction is not executed. The correct functionality of the predicate define instruction is achieved by this code sequence. In the case where the variable *a* is zero, *p1* is set to 1 and *p2* is set to 0. Under the opposite condition, *p1* is not modified and *p2* is set to 1.

Instruction (3) in Figure 8.1 is an example of a predicated predicate define instruction. The assembly code sequence is similar to that of instruction (2), with the addition of a branch instruction to handle the predicate. Other predicated instructions are emulated by extracting the bit from one of the reserved registers that corresponds to the predicate for that instruction. The value of that bit is used to conditionally execute the predicated instruction. For example, instruction (4) in Figure 8.1(a) is predicated on *p3*. Thus, bit 3 is extracted from *%r3* using the bit extract instruction, and the value extracted is used to conditionally nullify the increment of *c*. In the case where the extracted bit is 0, the subsequent instruction is nullified. Otherwise, the subsequent instruction is allowed to execute normally. The proper semantics of predicated instructions are therefore realized.

For a detailed description of the functionality of the instructions used in Figure 8.1(b), the reader is referred to the HP PA-RISC Instruction Set Manual [75].

8.2 Results

As with Section 5.2, performance improvement is presented throughout this section using a speedup calculation. Speedup is computed by dividing the total execution cycles of the base configuration by the total execution cycles of the evaluated configuration. The major difference with the results in Section 5.2 is that the base configuration is changed from an issue-1 processor with basic block compilation support to an issue-1 processor with restricted speculation and superblock compilation support. Overall, this reduces the absolute speedups reported by eliminating the code efficiency improvements achieved through expanding the compiler's scope from basic blocks to superblocks. Again, the same cache models are assumed for both configurations in the speedup calculation. Thus, for those experiments that utilize perfect caches, the execution cycles for the base configuration are derived using a perfect cache, whereas for those experiments that use a finite cache, the execution cycles for the base configuration are derived using the same sized finite cache.

8.2.1 Overall performance with predicated execution

The performance improvement achieved with predicated execution using hyperblock compilation techniques is presented in Figures 8.2 - 8.4. The figures show results for processor issue rates of 2, 4 and 8, respectively. Each processor is able to issue at most one branch per cycle. The figures show the speedup achieved over the base processor for two configurations. The first configuration, "Superblock General," has architectural support for general speculation and utilizes superblock compilation support. This represents the highest performance configuration discussed in Chapter 5. The second configuration, "Hyperblock General," has architectural support for both general speculation and predicated execution and utilizes hyperblock compilation support. For all the data presented in these figures, perfect caches are assumed.

From Figure 8.2, little performance improvement is achieved with predicated execution for an issue-2 processor. In fact, for many of the benchmarks, performance loss is observed. This behavior is a composite of two factors. First, the combination of superblock compilation support and general speculation is highly effective for an issue-2 processor. There are few idle processor resources to take advantage of with increased ILP. As a result, there is little opportunity to increase performance for an issue-2 processor. The second factor is that the hyperblock techniques tend to be aggressive and assume a wide issue processor when applying

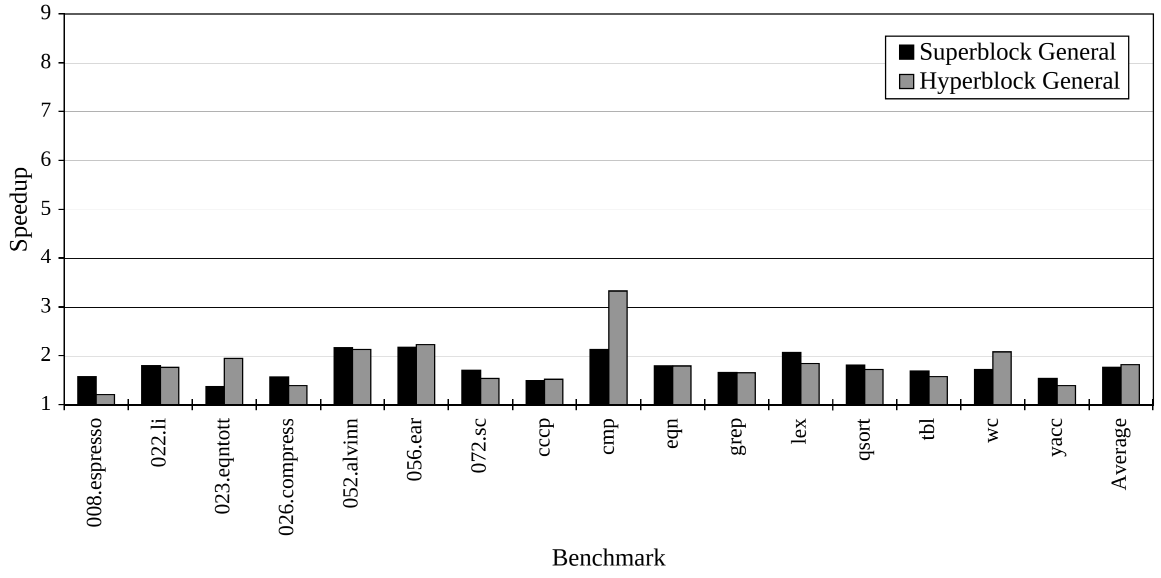


Figure 8.2 Performance improvement achieved with predicated execution for an issue-2, one-branch processor.

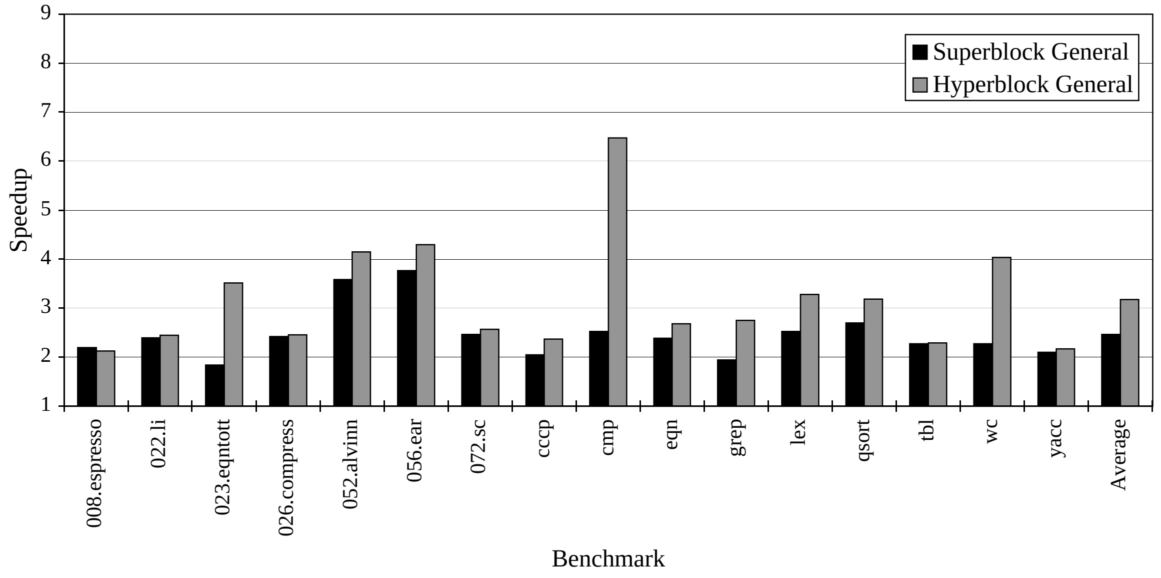


Figure 8.3 Performance improvement achieved with predicated execution for an issue-4, one-branch processor.

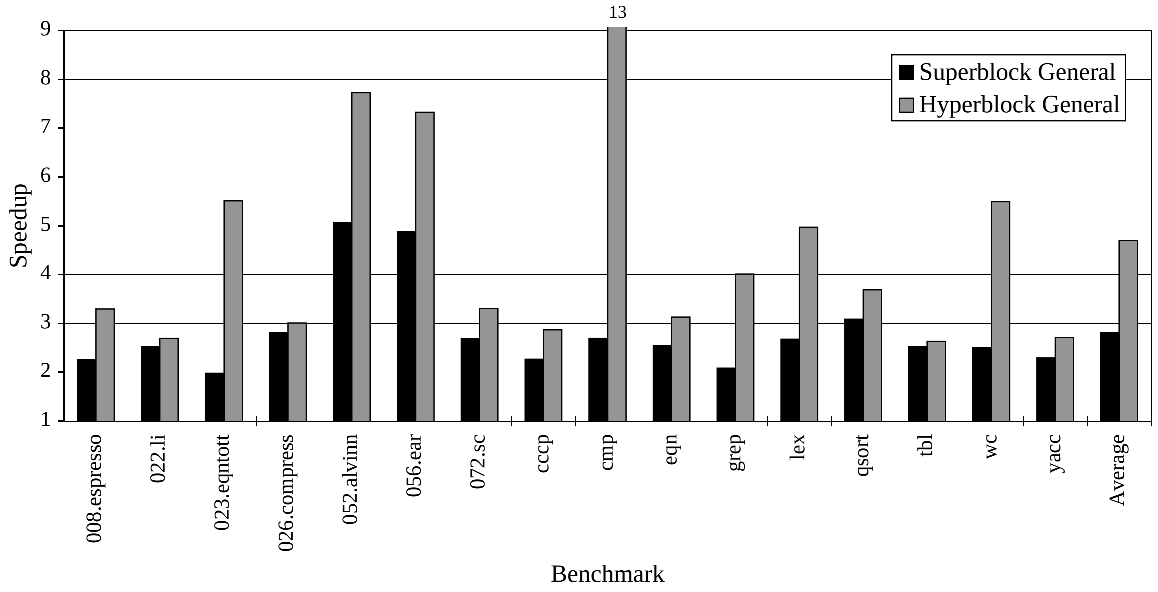


Figure 8.4 Performance improvement achieved with predicated execution for an issue-8, one-branch processor.

transformations. For processors with insufficient instruction execution bandwidth, processor resources tend to be over-saturated in the hyperblocks. The effect is an increase in the execution time of the program.

With an issue-4 processor (Figure 8.3), more consistent performance gains are observed with predicated execution. The most branch intensive programs tend to show the largest improvements, including *023.eqntott*, *cmp*, *grep*, and *wc*. The ability of the compiler to eliminate branches to reduce contention for the branch resources and to eliminate branch mispredictions to get rid of the misprediction penalties are the main reasons for the improvement. For example, consider the benchmark *cmp*. Recall from Table 5.12, that 59% of the dynamic instructions in this program are branches after superblock optimizations. On an issue-4 processor capable of executing 1 branch per cycle, a performance bound of approximately 2 instructions per cycle is established by the resource constraints. However, with predicated execution, many of these branches can be eliminated via transformations such as if-conversion and branch combining. The overall results are the elimination of the branch resource bottleneck and a large increase in speedup due to the available ILP.

Performance gains, however, are still somewhat limited for many of the benchmarks with an issue-4 processor. This trend occurs because the issue-4 processor is still suffering from one of the problems of the issue-2 processor. Namely, the hyperblock techniques are overly aggressive and tend to assume a wide issue (8-12 issue) processor when applying transformations. Thus, processor resources tend to become over-saturated in some hyperblocks. A good example of this occurs for *008.espresso*. In this benchmark, loop peeling is performed a large number of times. The loop peeling and subsequent hyperblock optimizations increase the ILP substantially. But, loop peeling creates multiple predicated copies of innermost loop bodies and results in too many instructions for an issue-4 processor to absorb. The result is a significant increase in the schedule lengths for some of the most frequently executed hyperblocks, which causes the net overall performance loss.

The performance results achieved by increasing the processor issue width to 8 are presented in Figure 8.4. For an issue-8 processor, the hyperblock compilation techniques are highly effective at increasing performance. Overall, the average speedup across the benchmarks of Hyperblock General is 68% above that of Superblock General. There are several major reasons for the improved performance of Hyperblock General. First, the number of processor resources is large enough to eliminate most of the over-saturation problems previously discussed. Thus, the processor is able to take full advantage of the increased ILP provided by the hyperblock techniques. The most notable example is the benchmark *008.espresso*. For both issue-2 and issue-4 processors, performance loss is observed with Hyperblock General. However, for the issue-8 processor a 46% performance increase over that for Superblock General is achieved. The hyperblock loop peeling optimization becomes highly effective with enough processor resources for this benchmark.

The second reason for the improved performance of Hyperblock General is that the branch resource limitations and branch misprediction penalties become an obvious problem for superblock code at issue-8. Branch intensive programs are limited to a maximum 2-4 instructions per cycle on average with only one branch unit. Similarly, each cycle lost due to a branch misprediction is really a loss of 8 potential instructions that might have been issued. As a result, decreases in both branch instructions as well as branch mispredictions translate directly into improved performance. Benchmarks limited by branch resource contention include *cmp*, *grep*, and *lex*. All of these benchmarks benefit significantly from branch combining to remove a large fraction of the dynamic branches. Benchmarks limited by branch mispredictions include *023.eqntott*, *qsort* and *wc*. The use of if-conversion to form hyperblocks removes a large fraction of the highly mispredicted branches from these benchmarks. The branch and misprediction effects are examined in more detail in Section 8.2.3.

The final reason for the improved performance of Hyperblock General is the ability of the compiler to overlap the execution of multiple paths of control. For many of the benchmarks, the branches are not heavily biased; thus, single path techniques, such as superblocks, are inherently limited. The limitations become more of a problem as the demand for ILP increases with expanding processor issue width. With predicated execution, the compiler employs if-conversion to selectively overlap the execution of multiple paths of control. As a result, the compiler exposes ILP along multiple paths of control to the hardware. Performance is improved for all the benchmarks with this increased ILP.

8.2.2 Branch resource effects

The results from the previous section assume that the processor can issue a maximum of one branch per cycle. This assumption is made due to the difficulties associated with designing processors which execute multiple branches per cycle. Multiple branches require significant additional pipeline complexity as well as dealing with difficult branch predictor design issues. Thus, future generation ILP processors will likely have limited branch handling capabilities. However, it is important to evaluate the effectiveness of predicated execution with fewer branch constraints to more deeply understand its uses and limitations. The overall performance of predicated execution with multiple branches per cycle is examined in Figures 8.5 and 8.6. The experiments are performed assuming perfect caches.

Figure 8.5 shows the performance improvement with predicated execution for an issue-8 processor with the number of branches per cycle increased to 2. The use of predicated execution and hyperblock compilation techniques with this configuration still shows substantial performance improvements for most of the benchmarks. Overall, the average speedup across the benchmarks with Hyperblock General is 34% higher than that of Superblock General. Despite the additional branch resource, the compiler is able to derive many of the same advantages as the issue-8, one-branch case. Namely, the ability to reduce the number of branches and overlap the execution of multiple paths of control allow the compiler to significantly enhance the ILP. In addition, the reduction in branch mispredictions because of the removal of unbiased branches decreases the cycles associated with handling the mispredictions. The combination of the increased ILP and the reduction in penalty cycles leads to the performance improvement.

The overall increase with Hyperblock General though is noticeably smaller than the previous data for an issue-8, one-branch processor. Comparing these results with the earlier results

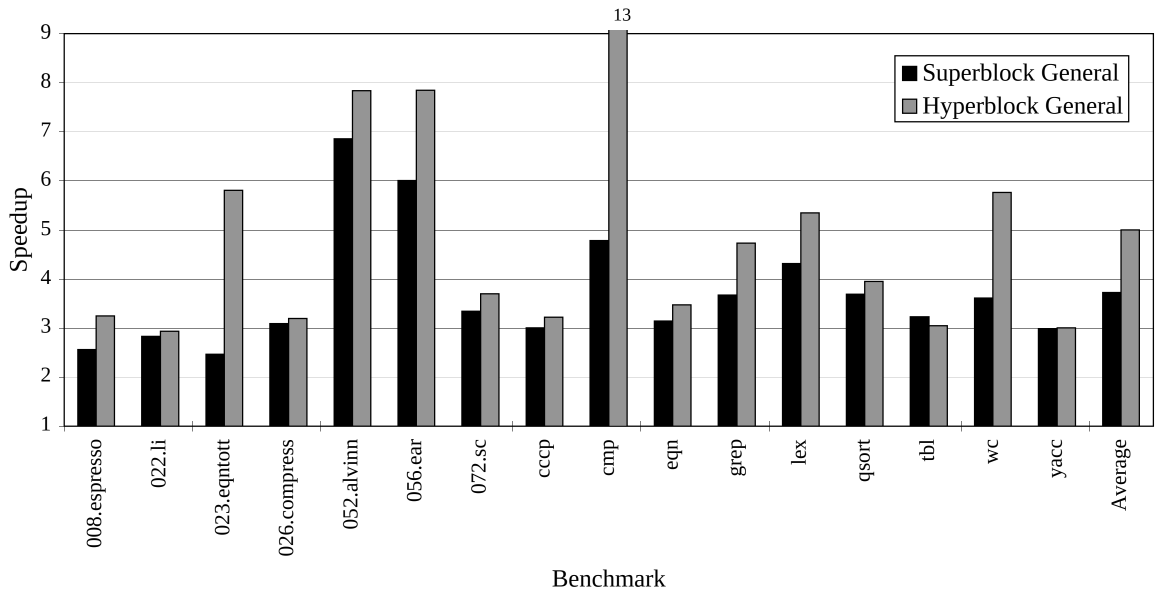


Figure 8.5 Performance improvement achieved with predicated execution for an issue-8, two-branch processor.

(Figure 8.4) shows that the performances of both Superblock General and Hyperblock General are increased with the additional branch resource. However, the performance of Superblock General is increased by a significantly larger margin. This result could be anticipated. The hyperblock code has a large fraction of the branches eliminated. Therefore, it does not benefit significantly from an additional branch resource. However, the superblock code is highly branch intensive. Thus, it benefits greatly from the additional branch resource.

The performance effects of completely removing the limitation on the number of branches per cycle is presented in Figure 8.6. For this figure, the issue-8 processor can issue any combination of 8 instructions each cycle. On the surface, the effectiveness of predicated execution and hyperblock compilation techniques has become less clear. For several of the benchmarks, including *grep*, *lex*, *tbl*, and *yacc*, Superblock General noticeably outperforms Hyperblock General. For other benchmarks, including *008.espresso*, *023.eqntott*, *056.ear*, and *wc*, Hyperblock Gen-

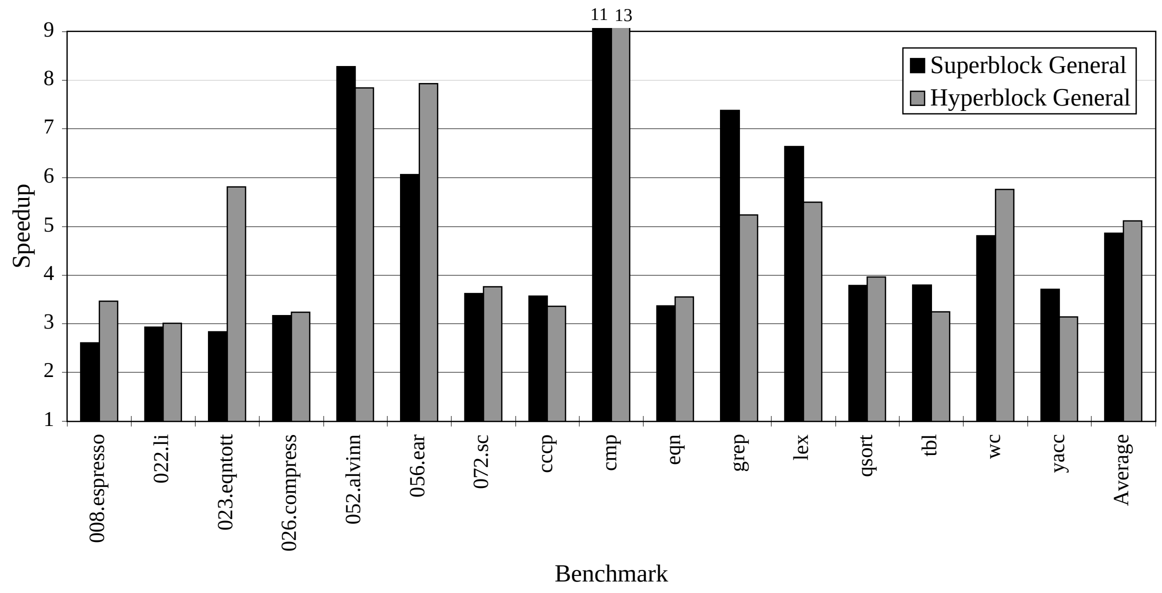


Figure 8.6 Performance improvement achieved with predicated execution for an issue-8 processor with no constraints on branches.

eral shows large performance improvements. Closer examination of the benchmarks provides an important insight into the performance of Hyperblock General, namely that the performance improvement gained from predicated execution can be broken into two categories.

The first category is the performance effect that is achieved by reducing the number of branches in the code. For the issue-8, one-branch processor, this category contributes substantially to the overall performance. However, as the number of branch resources is increased to the issue width of the processor, the contribution of this category goes to zero. In fact, it becomes a detriment to the overall performance. Eliminating branches via hyperblock techniques has a nonzero cost; thus, if there is no performance advantage gained by applying the transformations, a net performance loss is observed. This is the case for benchmarks such as *grep*, *lex*, and *tbl*. These benchmarks are primarily limited by the frequency of branches, so they have little chance for improvement without any branch resource limitations.

The other category of performance improvements achieved with predicated execution is that gained by overlapping the execution of multiple control paths and by eliminating branch mispredictions. Regardless of the number of branch resources, predicated execution offers performance advantages in these areas. Several of the benchmarks that show this distinctly are *008.espresso*, *023.eqntott*, *056.ear*, and *wc*. These benchmarks share the common quality of having unbiased branches in the code that is most frequently executed. The hyperblock compilation techniques are able to utilize predicated execution to efficiently overcome the limitations of these unbiased branches. The overall result is the substantial performance improvement regardless of the number of branch resources.

8.2.3 Instruction stream effects

To better understand the overall performance results presented in the previous two sections, some of the important effects that predicated execution and hyperblock techniques have on the instruction stream are investigated in this section. All experiments in this section use an issue-8 processor capable of issuing at most 1 branch per cycle. In addition, perfect caches are assumed.

The effect of predicated execution on the dynamic instruction count is presented in Table 8.1. The table contains the number of dynamic instructions executed for two configurations: Superblock General and Hyperblock General. The ratio of the Hyperblock General instruction count with respect to the Superblock General instruction count is provided in parentheses. The Hyperblock General data includes instructions which are predicate nullified. Intuitively, one would expect the Hyperblock General instruction count to be significantly higher than for the Superblock General because hyperblock formation combines instructions from multiple paths

Table 8.1 Effect of predicated execution on the dynamic instruction count for an issue-8 processor.

Benchmark	Superblock General	Hyperblock General
008.espresso	488M	644M (1.32)
022.li	32M	33M (1.04)
023.eqntott	1030M	892M (0.87)
026.compress	90M	108M (1.20)
052.alvinn	3575M	3604M (1.01)
056.ear	11273M	11273M (1.00)
072.sc	116M	122M (1.05)
cccp	3731K	3923K (1.05)
cmp	932K	921K (0.99)
eqn	45M	45M (1.00)
grep	1282K	1647K (1.28)
lex	35M	46M (1.29)
qsort	48M	52M (1.09)
tbl	2566K	2934K (1.14)
wc	1493K	1526K (1.02)
yacc	43M	51M (1.18)
Average	-	- (1.10)

into a single path. Therefore, more instructions are executed to avoid branches that determine the particular control path that requires execution.

Table 8.1 surprisingly shows only modest increases in the instruction count. The largest increase is 32% for *008.espresso* and the overall average across the benchmarks is 10%. For two of the benchmarks, *023.eqntott* and *cmp*, the dynamic instruction count is actually smaller for Hyperblock General. Examination of the benchmarks in detail reveals that the major reason for this behavior is the competing effects of increased instruction count because of overlapping multiple paths and reduced instruction count because of less speculation. With superblock compilation support, the compiler aggressively speculates instructions across many branches to achieve a compact schedule. Each branch above which an instruction is speculated increases the instruction's execution count. The problem becomes magnified with the tail duplication that is performed during superblock formation. Tail duplication creates additional copies of instructions. With aggressive speculation, the compiler can place multiple instances of the same instruction along particular paths of control. When these paths are traversed, the instruction is not only executed more times because it is speculated, it may also be executed a multiplicative number of times due to the duplication.

On the other hand, with predicated execution, the compiler speculates much less often and does not incur the instruction count increase associated with speculation. Many of the branches have been removed; thus, there are fewer speculation opportunities. In addition, the ILP is higher with predicate support; therefore, the compiler speculates less to achieve a packed schedule. These effects tend to cancel out some or all of the effects of the conventional instruction count increases incurred with predicated execution. The overall result observed for most of the benchmarks is a modest net instruction count increase. For two of the benchmarks, *023.eqntott* and *cmp*, the reduction in instruction count from reduced speculation exceeds any increase due to overlapping multiple paths. Thus, the net decrease is the result.

The effect of predicated execution on the dynamic number of branches is presented in Table 8.2. As in the previous table, this table contains the dynamic number of branches executed for Superblock General and Hyperblock General. The ratio of the Hyperblock General branch count with respect to the Superblock General branch count is provided in parentheses. The table clearly shows the effectiveness of using predicated execution to reduce the number of branches in the instruction stream. For ten of the sixteen benchmarks measured, over half of the branches are removed. The benchmark with the largest number of branches removed is *cmp*. For this benchmark, only 5% of the dynamic branches remain with predicated execution.

Table 8.2 Effect of predicated execution on the dynamic branch instruction count for an issue-8 processor.

Benchmark	Superblock General	Hyperblock General
008.espresso	73M	35M (0.48)
022.li	7442K	6085K (0.82)
023.eqntott	306M	52M (0.17)
026.compress	12M	9055K (0.74)
052.alvinn	462M	73M (0.16)
056.ear	1538M	442M (0.29)
072.sc	22M	11M (0.48)
cccp	920K	534K (0.58)
cmp	530K	26K (0.05)
eqn	7694K	4510K (0.59)
grep	662K	171K (0.26)
lex	14M	3021K (0.21)
qsort	8670K	6104K (0.70)
tbl	609K	416K (0.68)
wc	478K	223K (0.47)
yacc	11M	5848K (0.49)
Average	-	- (0.45)

The reductions in the number of branches translate directly into improved performance for processors with limited branch execution capabilities. The branch resource limitations place an upper bound on the instruction throughput of these processors. However, with the large reductions in the number of branches, the resource limitations are removed and performance increases.

Table 8.3 reports the effect of predicated execution on the number of dynamic branch mispredictions and the misprediction rate. As with the previous two tables, data for Superblock General and Hyperblock General are reported. The ratio of mispredictions with Hyperblock General to Superblock General is also given in parentheses. From the table, the compiler is able to consistently utilize predicated execution to eliminate branches and the mispredictions asso-

Table 8.3 Effect of predicated execution on the dynamic branch misprediction count for an issue-8 processor.

	Superblock General		Hyperblock General	
Benchmark	Mispredictions	Miss rate	Mispredictions	Miss rate
008.espresso	3480K	4.76	1512K (0.43)	4.27
022.li	764K	10.27	685K (0.90)	11.26
023.eqntott	43M	14.12	6500K (0.15)	12.30
026.compress	1336K	10.91	855K (0.64)	9.44
052.alvinn	1112K	0.24	992K (0.89)	1.34
056.ear	64M	4.20	15M (0.23)	3.43
072.sc	1278K	5.56	844K (0.66)	7.61
cccp	65K	7.14	64K (0.99)	12.11
cmp	4395	0.83	31 (0.01)	0.12
eqn	587K	7.64	497K (0.85)	11.04
grep	9660	1.46	20K (2.08)	11.73
lex	229K	1.62	195K (0.85)	6.46
qsort	1321K	15.24	654K (0.50)	10.72
tbl	39K	6.40	37K (0.96)	8.96
wc	32K	6.85	57 (0.00)	0.03
yacc	502K	4.23	438K (0.87)	7.49
Average	-	6.34	- (0.69)	7.39

ciated with those branches. The fraction of mispredictions eliminated, however, varies widely across the benchmarks. Two of the benchmarks, *cmp*, and *wc*, show drastic improvements with virtually all of their mispredictions eliminated. Four other benchmarks (*008.espresso*, *023.eqntott*, *056.ear*, and *qsort*) have over half of their mispredictions eliminated. For these benchmarks, the hyperblock compilation techniques are consistently able to identify and target the unbiased branches for removal. Utilizing if-conversion and loop peeling, the compiler is highly successful at reducing the number of mispredictions.

The compiler is less successful at reducing mispredictions in other benchmarks. For programs such as *022.li* and *cccp*, there are a large number of mispredictions without predicated execution support. However, the hyperblock techniques are unsuccessful at removing enough of the highly mispredicted branches. For other programs such as *grep* and *lex*, the primary bottleneck is not branch mispredictions, but rather the branch instructions themselves. Therefore, the compiler focuses its work on eliminating branches from the instruction stream, even heavily biased ones. The result is a large drop in the dynamic branches (Table 8.2) and little effect on the branch mispredictions.

The benchmark *grep* actually has the reverse effect of increasing the number of mispredictions with hyperblocks due to the focused effort to reduce the number of dynamic branches. More specifically, *grep* has a large number of opportunities to eliminate branches with the branch combining optimization. Branch combining essentially introduces an additional level of control flow to allow a large number of branches to be moved outside a hyperblock and thus be executed fewer times. The negative of this optimization is that both the first-level branch and the second-level branches can be mispredicted. For most benchmarks, the small misprediction increases from branch combining are hidden by improvements elsewhere in the program. With *grep*, there are no significant opportunities for misprediction improvement; therefore, a net increase from branch combining is observed.

An important piece of instruction stream data to examine is the number of instructions between branches or the distance between branches. These data represent how often the processor is faced with a control transfer decision versus executing sequential memory and arithmetic instructions. The effect of predicated execution on the distance between branches and mispredicted branches is summarized in Table 8.4. For both configurations, Superblock General and Hyperblock General, the number of instructions between branches/mispredictions is averaged across the execution of the benchmarks. The ratios of both the branch and misprediction averages for Hyperblock General with respect to Superblock General are given in parentheses.

Table 8.4 Effect of predicated execution on the average distance between branches and mispredictions for an issue-8 processor.

	Superblock General		Hyperblock General	
Benchmark	Branches	MP Branches	Branches	MP Branches
008.espresso	5.7	139.4	17.2 (3.0)	394.1 (2.8)
022.li	3.3	41.2	4.5 (1.4)	47.9 (1.2)
023.eqntott	2.4	22.8	15.9 (6.7)	136.1 (6.0)
026.compress	6.4	66.6	10.8 (1.7)	123.9 (1.9)
052.alvinn	6.7	3215.7	48.0 (7.1)	3644.9 (1.1)
056.ear	6.3	173.8	24.5 (3.9)	743.9 (4.3)
072.sc	4.1	90.1	10.1 (2.5)	144.2 (1.6)
cccp	3.1	55.8	6.3 (2.1)	59.6 (1.1)
cmp	0.8	211.1	33.5 (44.1)	29730.9 (140.9)
eqn	4.9	76.7	9.1 (1.9)	90.9 (1.2)
grep	0.9	131.8	8.6 (9.2)	81.0 (0.6)
lex	1.5	155.8	14.3 (9.3)	236.1 (1.5)
qsort	4.6	34.7	7.6 (1.7)	76.6 (2.2)
tbl	3.2	64.8	6.0 (1.9)	77.6 (1.2)
wc	2.1	44.6	5.8 (2.7)	26775.7 (600.5)
yacc	2.6	84.4	7.6 (2.9)	114.1 (1.4)
Average	3.7	288.1	14.4 (6.4)	3904.9 (48.1)

The table shows the large increases in the distance between branches that are achieved with predicated execution. The most control intensive application is the benchmark *cmp*, with an average of 0.8 instruction between branches for Superblock General. With Hyperblock General, the distance is drastically increased to 33.5 instructions. Other benchmarks do not show this enormous growth, but still show rather substantial increases. The average across all the benchmarks grows from 3.7 instructions with Superblock General to 14.4 instructions with Hyperblock General. These data are extremely important for estimating the branch handling capabilities needed to sustain an ILP processor. Without predicated execution, an additional branch is required for every three to four instruction slots the issue width is increased in order to sustain the execution throughput. With predicated execution, an additional branch is only required for every 14-15 instruction slots the issue width is increased. This reduction in the branch bandwidth requirements makes the design of wide issue ILP processors much more feasible.

Table 8.4 shows similar trends for the distance between branch mispredictions. With the exception of *grep*, the distance between mispredictions increases across all the benchmarks with Hyperblock General. These increases are extremely important since the distance between mispredictions is a direct measure of the work that a processor may accomplish between successive misprediction repairs. The two benchmarks which stand out the most are *cmp* and *wc*. For these benchmarks, the growth in misprediction distance is enormous. This result could be expected based on the branch misprediction data presented earlier in Table 8.3. The number of branch mispredictions is reduced to nearly zero with predicated execution; thus, any distances between these mispredictions will be extremely large. It should be noted that the average reported in the table is significantly skewed by these two benchmarks. The one anomaly is the benchmark *grep*, which has a smaller distance between mispredictions with predicated execution. This behavior is a direct result of the increase in the number of mispredictions caused by branch combining as described previously.

An interesting way to examine the effects of predicated execution on the branch and misprediction distances is to compare the distributions of the distances. The distributions for Superblock General and Hyperblock General which plot the distance between branches and mispredictions are presented in Figures 8.7 and 8.8, respectively. The instruction distances are broken down into 11 distinct groups with the ranges specified on the horizontal axes. The vertical axes specify the fraction of branch intervals which contain the given number of instructions. Within each benchmark, the fractions are weighted by the dynamic execution frequency. Then,

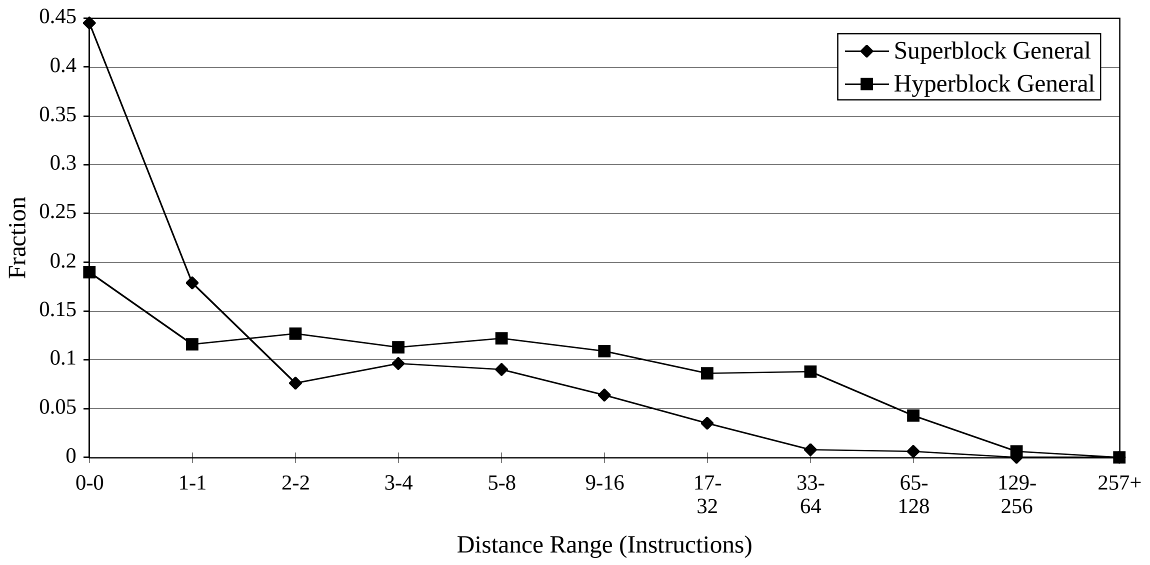


Figure 8.7 Effect of predicated execution on the distribution of the distance between branches for an issue-8 processor.

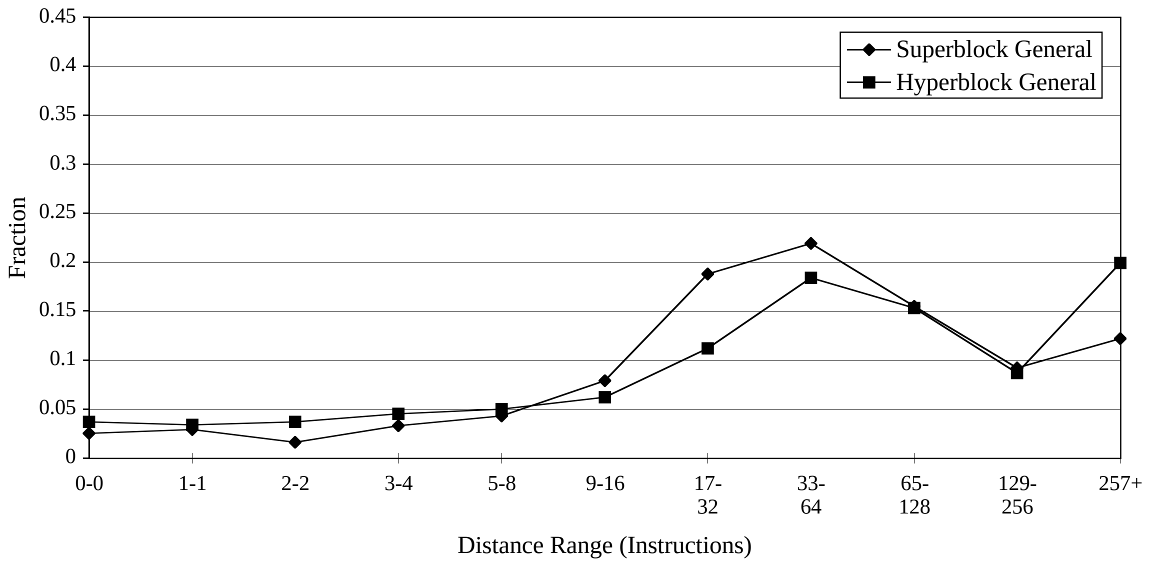


Figure 8.8 Effect of predicated execution on the distribution of the distance between branch mispredictions for an issue-8 processor.

these values are normalized and averaged across all 16 benchmarks to achieve the final data point. Each benchmark contributes equally to the final distribution. An important caveat to keep in mind when examining these figures is that there are a different number of intervals for each configuration. For the most part, the Hyperblock General configuration has fewer branches and mispredictions; thus, the number of intervals is smaller.

Figure 8.7 clearly shows the increase in the observed distance between branches. The Hyperblock General distribution is noticeably shifted towards the larger distance ranges. The largest difference between the two distributions occurs for the 0 instruction range, which corresponds to back-to-back branches in the instruction stream. Hyperblock compilation techniques are therefore successful at eliminating a significant fraction of the back-to-back branches from the benchmarks. The effect of predicated execution is not as clear for the misprediction distance distributions in Figure 8.8. The results are somewhat hidden because of the large increases that occur in the "257+" group that do not show up in the figure. Not only does the number of intervals that fall into this range increase, but the actual distances in this range also grow substantially. An interesting note is that predicated execution has the smallest effects on the mispredictions that occur in closest proximity. Obviously, the branches responsible for these mispredictions are not amenable to the current set of hyperblock compilation techniques.

8.2.4 Speculative execution effects

Although predicated execution is used to eliminate a large fraction of the branches, speculative execution is still extensively used within the hyperblock compilation framework. Speculative execution is accomplished in two ways with hyperblocks. First as with superblocks, speculative execution increases the code motion freedom during optimization and scheduling by allowing instructions to move across control dependent branches. Hyperblock techniques selectively eliminate branches; thus, there are a significant number of branches that remain in the code to provide ample speculation opportunities. The second manner in which speculation is accomplished in predicated code is by predicate promotion. As was described in Chapter 7, predicate promotion advances an instruction's predicate to a less constrained predicate to allow greater code motion freedom. As a result, the instruction is executed more often than was specified by the original program semantics. Hence, the instruction is speculated. The effectiveness of speculative execution on predicated code along with the relative importance of predicated and speculative execution are investigated in this section.

The contributions to overall performance of speculative and predicated execution are compared in Figure 8.9 for an issue-8, one-branch processor. The comparison is achieved by considering separately the performance of code with neither speculation nor predication (SB-R), speculation support alone (SB-G), predication support alone (HB-R), and support for both (HB-G). It should be noted that all data presented in this chapter up to this point for hyperblocks use both speculation and predication. From the figure, the relative contributions of speculation and predication vary widely across the benchmarks. For benchmarks such as *022.li* and *tbl*, the majority of the overall performance is achieved with speculation alone. The use of predication alone yields poor performance, and the combination of speculation/predication achieves little over speculation alone. These benchmarks are inherently not limited by control flow, but rather by data, memory, and control dependences. Speculation provides some improvement by removing control dependences, but the remaining data and memory dependences are the major bottlenecks.

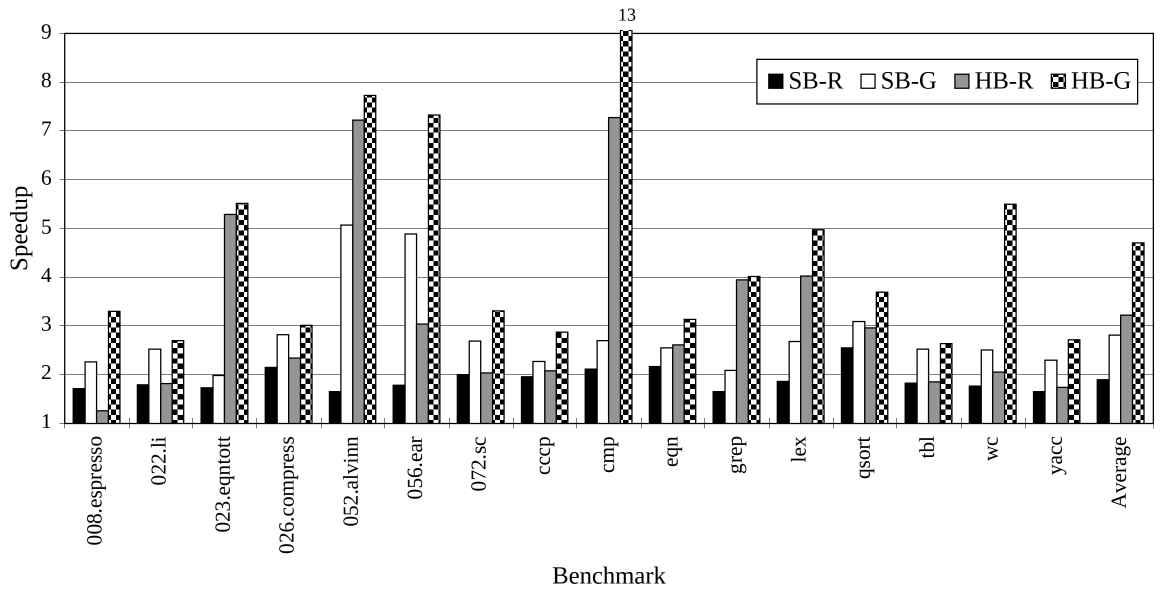


Figure 8.9 Performance comparison of predicated execution and speculative execution for an issue-8, one-branch processor.

At the other extreme, there are several benchmarks in which the majority of the overall performance is achieved with predication alone. Benchmarks exhibiting this behavior include *023.eqntott* and *grep*. These benchmarks are extremely limited by branches, either by branch resource limitations, branch mispredictions, or the lack of dominant execution paths. By eliminating branches and overlapping the execution of multiple paths, these problems are overcome. These benchmarks are also characterized by a high level of ILP. Thus, when the branch problems are overcome, large performance improvements are immediately observed.

The predominant behavior, though, is not indicated by either of these extreme positions. The majority of benchmarks requires the combination of predication and speculation to achieve the maximal performance. Either predication or speculation alone yields substantially lower performance than the combination. Two benchmarks that illustrate this point clearly are *008.espresso* and *wc*. These benchmarks achieve little performance improvement with either

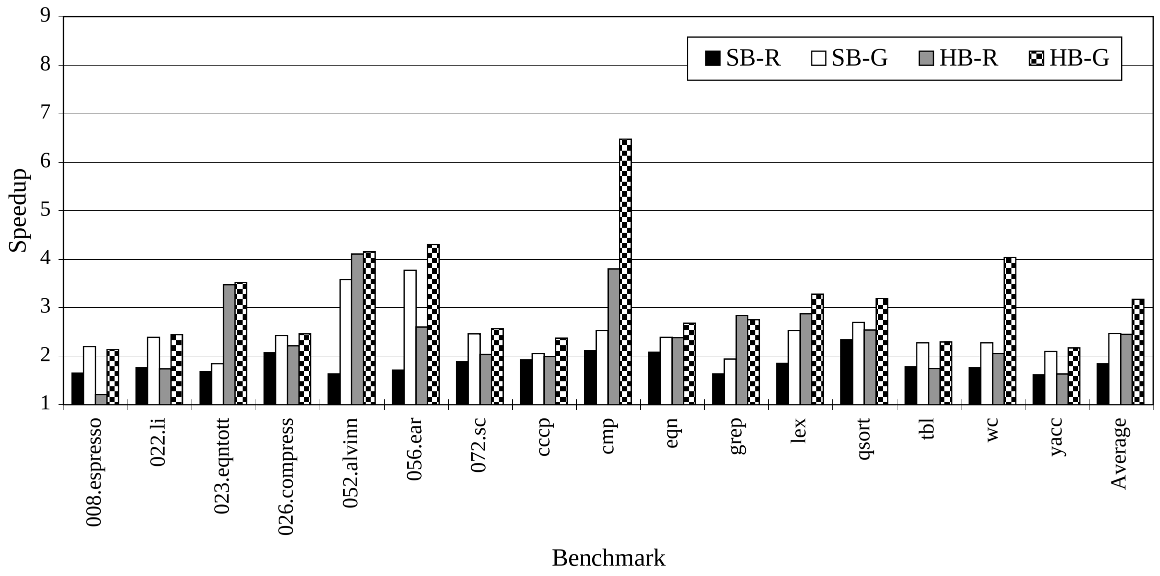


Figure 8.10 Performance comparison of predicated execution and speculative execution for an issue-4, one-branch processor.

speculation or predication alone. However, the combination of the two yields substantial performance gains. The performance gains are actually higher than the individual speedups added together. The major reason for this behavior is that most of the benchmarks are limited by both control dependences as well as branch instructions themselves. Removing one problem enables some performance gain, but the gain is limited because the other problem is quickly exposed and becomes the performance bottleneck. With support for speculation and predication, both problems can be efficiently overcome, which allows a large increase in ILP.

The previous experiment is repeated in Figure 8.10 for an issue-4, one-branch processor. As with the previous experiment, the combination of speculation and predication support provides significant performance improvement over either support alone. The limitations imposed by both control dependences as well as branches must be jointly overcome to achieve the maximal performance level. The most significant difference from the issue-8 experiment is the reduced effectiveness of predicated execution. The relative gains of the predication alone and speculation/predication are noticeably less for several of the benchmarks with the issue-4 processor. This behavior is due to the over-saturation problems of the predicated code for narrower issue processors that were discussed earlier. As a result, the contribution of speculation to the overall performance is higher with the reduced issue rate used in this experiment. Benchmarks, such as *008.espresso*, *056.ear*, and *072.sc*, illustrate this point. For the issue-8 processor, they achieved significant improvements over speculation alone with both speculation and predication support, whereas with the issue-4 processor, most of the performance is achieved with speculation alone.

A large fraction of the speculation in hyperblocks is achieved with predicate promotion. Promoted instructions execute under a less constrained predicate than they were originally assigned. Instructions are promoted completely when their predicate is advanced to true. Table 8.5 provides several statistics on the application of predicate promotion for the benchmarks. The first data column specifies the dynamic number of predicated instructions before any predicate promotion is applied. The second and third columns specify the dynamic number of instructions whose predicates have been promoted and completely promoted, respectively. Finally, the dynamic number of instructions that remain predicated after promotion is complete is shown in the last column. Note that the third and fourth columns should total the number of originally predicated instructions. The ratios of each column with respect to the first data column are specified in parentheses.

From Table 8.5, the fraction of promoted predicated instructions varies widely across the benchmarks. For six of the benchmarks, very little predicate promotion is applied. Four of these benchmarks have absolutely zero instructions predicate promoted and the remaining two benchmarks have less than a 5% promotion rate. The major reason for the small amount

Table 8.5 Predicate promotion statistics.

Benchmark	Original Predicated	Promoted	Promoted to T	Remain Predicated
008.espresso	452M	298M (0.66)	266M (0.59)	186M (0.41)
022.li	6524K	2237K (0.34)	2237K (0.34)	4287K (0.66)
023.eqntott	194M	3656K (0.02)	3648K (0.02)	191M (0.98)
026.compress	22M	3215K (0.14)	3215K (0.14)	19M (0.86)
052.alvinn	29M	0 (0.00)	0 (0.00)	29M (1.00)
056.ear	1144M	238M (0.21)	238M (0.21)	906M (0.79)
072.sc	31M	10M (0.32)	10M (0.32)	21M (0.68)
cccp	872K	165K (0.19)	165K (0.19)	706K (0.81)
cmp	118K	0 (0.00)	0 (0.00)	118K (1.00)
eqn	2669K	764K (0.29)	764K (0.29)	1905K (0.71)
grep	35K	0 (0.00)	0 (0.00)	35K (1.00)
lex	18M	730K (0.04)	694K (0.04)	17M (0.96)
qsort	20M	4336K (0.22)	4336K (0.22)	15M (0.78)
tbl	686K	332K (0.48)	327K (0.48)	358K (0.52)
wc	1051K	0 (0.00)	0 (0.00)	1051K (1.00)
yacc	18M	7398K (0.40)	7398K (0.40)	11M (0.60)
Average	-	- (0.21)	- (0.20)	- (0.80)

of promotion in these benchmarks is the lack of opportunity for profitable promotions. The predicate computations in these benchmarks are generally not along the critical dependence chains in the hyperblocks. Therefore, there is little to be gained in terms of dependence height reduction by removing dependences to the predicate computation instructions through predicate promotion.

The remainder of the benchmarks has a significant fraction of the predicated instructions promoted. The largest degree of promotion occurs for *008.espresso* in which 66% of the predicated instructions are promoted. For these benchmarks, the predicate computations frequently do occur on the critical dependence chains. As a result, a large fraction of the predicated instructions are stuck waiting for the predicates to be computed. By performing predicate promotion, the dependences on the predicate computations are relaxed, which increases the

ILP in the code. One important case in which predicate promotion is applied frequently occurs in hyperblocks with peeled loops. The peeled iterations are predicated through a long chain of predicate computations to ensure that only the required number of iterations is executed. To achieve a compact schedule, predicate promotion is essential to break some of the dependences on the predicate computation. By promoting a sufficient number of the instructions, a compact schedule is achieved for the peeled loops and surrounding code. The large fraction of hyperblocks with peeled loops in *008.espresso* is the primary reason for the high promotion rate observed in this benchmark.

Table 8.5 also shows that most promoted instructions have their predicate advanced to the true predicate by the transformation. This behavior primarily occurs by design of the current predicate promotion algorithms. Instructions are carefully selected as candidates for promotion. But after that, a greedy heuristic is used to guide the degree of promotion. The motivation for this heuristic is that dependences are maximally relaxed by advancing an instruction's predicate as far as possible. When the predicate is advanced to true, the instruction itself is completely independent of any predicate computations. The net result is that almost all promotions are complete.

8.2.5 Cache effects

Up to this point in the evaluation of predicated execution and hyperblock compilation techniques, a perfect cache model has been assumed. In this section, this restriction is removed to study the effects of finite instruction and data cache models on performance.

The performance loss incurred going from perfect caches to 64K caches is presented in Figure 8.11. In the figure, the execution cycles for both perfect and finite caches are normalized

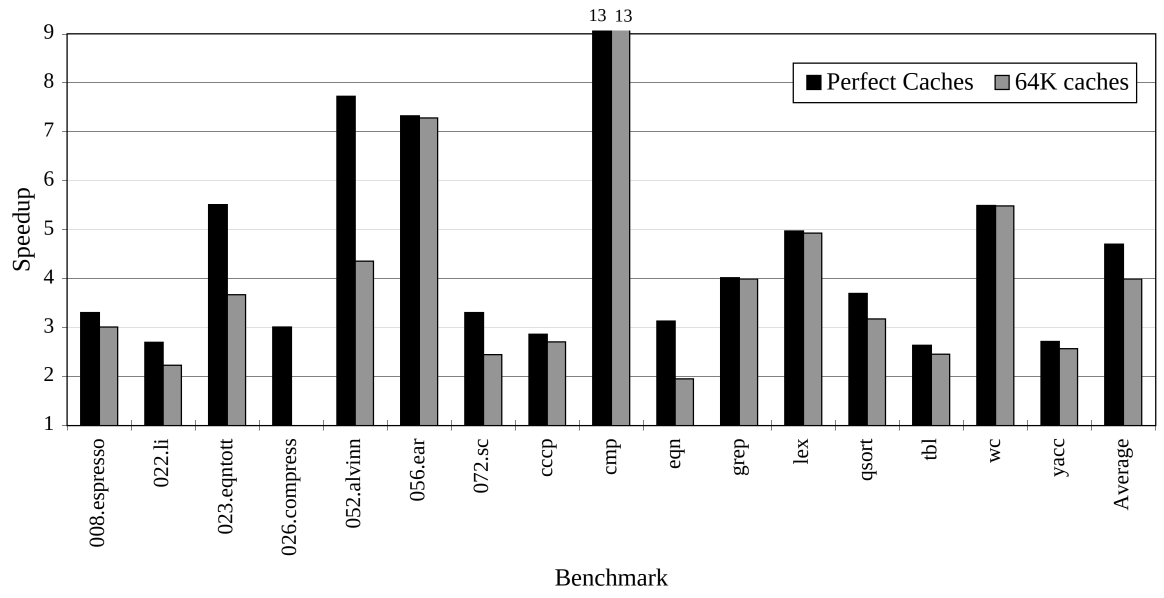


Figure 8.11 Performance comparison of perfect cache model and the 64K cache model for an issue-8 processor with predicated execution support.

with respect to the base configuration with perfect caches. For many of the benchmarks, only a small performance loss is incurred with the finite caches. The instruction and data working sets for these benchmarks have little difficulty fitting into 64K caches. The same conclusion was made for the earlier superblock experiment (see Figure 5.10). Therefore, the hyperblock transformations have not significantly altered this fact. This result is very positive because it shows that the majority of the performance gains achieved with predicated execution and hyperblock compilation techniques is not subsequently erased with a finite cache model.

For several of the benchmarks though, the performance loss is rather large with finite caches. The most obvious example is the benchmark *026.compress*. For this benchmark, the performance gain over the base processor is completely eliminated with finite caches. This behavior is a combination of factors. First, the data working set of *026.compress* is extremely large and causes thrashing in a 64K direct mapped cache. This was indicated by the large performance loss in the previous superblock experiment. But, the problem becomes noticeably worse with hyperblocks, which brings up the the second factor, namely, a large number of additional data cache misses are introduced by the hyperblock compilation techniques that cause the additional performance drop. The remaining benchmarks that suffer noticeable performance losses with predicated execution do so for the same basic reasons. For benchmarks with consistent performance losses in both the superblock and hyperblock experiments, such as *052.alvinn*, the inherent working set(s) are rather large which cause a substantial number of stalls due to cache misses. Other benchmarks such as *023.eqntott* and *eqn* have significant increases in either instruction and/or data cache misses, which leads to the larger speedup loss that is observed with hyperblocks.

The effects of predicated execution and hyperblock compilation techniques on the caches are examined in more detail in Table 8.6. In the table, the number of instruction and data cache misses are reported for the Superblock General and Hyperblock General configurations. The ratios of Hyperblock General to Superblock General for the instruction and data cache entries are shown in parentheses. To magnify the effects, a 4K instruction cache and perfect data cache are used for the instruction cache evaluation. In correspondence, a perfect instruction cache and a 4K data cache are used for the data cache evaluation. For both experiments, an issue-8, one-branch processor is assumed.

Looking first at the data cache results, the predominant trend is an increase in data cache misses with predicated execution. The primary cause of this behavior is the additional load misses introduced by hyperblock formation and subsequent predicate promotion. The hyperblock formation procedure combines multiple paths of control into a single block of predicated instructions using if-conversion. Loads from various paths are then typically predicate pro-

Table 8.6 Effect of predicated execution on the instruction and data caches for an issue-8 processor.

	Icache Misses		Dcache Misses	
Benchmark	Superblock	Hyperblock	Superblock	Hyperblock
008.espresso	2691K	1170K (0.44)	7148K	7203K (1.01)
022.li	1114K	894K (0.80)	1003K	1020K (1.02)
023.eqntott	604K	997K (1.65)	11M	11M (1.02)
026.compress	1856K	2630 (0.00)	4964K	6656K (1.34)
052.alvinn	334K	395K (1.18)	72M	69M (0.96)
056.ear	33M	57M (1.74)	98M	106M (1.08)
072.sc	1403K	1617K (1.15)	3197K	3177K (0.99)
cccp	47K	44K (0.94)	29K	38K (1.31)
cmp	32	23 (0.72)	28K	26K (0.92)
eqn	2143K	2015K (0.94)	787K	850K (1.08)
grep	73	75 (1.03)	3074	3041 (0.99)
lex	102K	61K (0.60)	318K	359K (1.13)
qsort	284	395 (1.39)	421K	488K (1.16)
tbl	55K	57K (1.04)	46K	55K (1.18)
wc	33	27 (0.82)	2662	1705 (0.64)
yacc	418K	542K (1.29)	1113K	1162K (1.04)
Average	-	- (0.98)	-	- (1.06)

moted to reduce the overall dependence height of the hyperblock. Loads are prime targets for predicate promotion because they often are near the top of long dependence chains. The problem occurs when these loads cause cache misses. Before promotion, only those loads whose predicates evaluate to true are executed and thus can cause cache misses. But, after promotion, loads across many paths are unconditionally executed, and thus can introduce cache misses. Essentially, this procedure is achieving load speculation from many paths of control. The larger fraction of speculative loads increases the data working set of the benchmarks, and a growth in data cache misses is observed.

The effects are most notable in two benchmarks: *026.compress* and *cccp*. For these benchmarks, data cache misses are increased by 34% and 31%, respectively. The data cache miss increase for *cccp* does not have a pronounced effect on the overall performance (Figure 8.11) because the larger, 64K cache is able to absorb the additional misses. However, for *026.compress*, the additional misses lead directly to a performance loss because of the already existing cache problems in this benchmark. The increase in data cache misses for *eqn* is also noticeable in the overall performance. The 8% increase in data cache misses directly translates into a large performance loss.

The effects of predicated execution and hyperblock compilation techniques on the instruction cache are also examined in Table 8.6. The behavior varies widely across the benchmarks. For half of the benchmarks, the number of instruction cache misses is reduced under the Hyperblock General configuration, whereas, for the other half, the number of misses is increased with hyperblocks. These differences reflect the two competing effects on the instruction cache caused by hyperblock compilation techniques. On the one hand, the locality of the code tends to increase with hyperblocks because all of the important paths are overlapped in a single predicated block. Therefore, there is less branching to different portions of the program and execution is sequential a higher fraction of the time. Overall, the increased locality tends to reduce instruction cache misses. A drastic reduction surprisingly happens for *026.compress*, which has its instruction cache misses almost completely eliminated with hyperblocks. The working set is effectively localized to a few blocks with hyperblock compilation techniques. Unfortunately, these effects do not translate into positive performance results because of the data cache problems.

The opposite effect is because of the larger code blocks that occur with hyperblocks. Since hyperblocks tend to overlap the execution of multiple paths, the overall size of individual hyperblocks is significantly larger than superblocks. Optimizations such as loop unrolling and branch target expansion also tend to multiply the size effects by replicating hyperblocks one or more times. The net result is an increased instruction working set and hence a larger number of instruction cache misses. The two benchmarks affected most by the instruction cache are *023.eqntott* and *056.ear*. For *056.ear*, the additional instruction cache misses are absorbed by the 64K cache; thus, no noticeable performance loss is observed in Figure 8.11. However, the additional instruction cache misses for *023.eqntott* are not hidden and the result is a noticeable performance drop.

8.2.6 Predicate architecture issues

An architecture supporting predicated execution contains several unique components. These include the ability to nullify instructions, additional instructions to manipulate predicates, and the predicate register file. In this section, several issues regarding the effectiveness and utilization of the predicate architecture are examined.

The first issue examined is the frequency of predicated instructions which are nullified during program execution. Table 8.7 presents these data. The first two columns in the table contain previously presented data, the total dynamic instructions (Table 8.1) and the total predicated dynamic instructions (Table 8.5). Note that the "Predicated" column data do not include instructions whose predicates are eliminated via transformations such as predicate promotion. The ratio of predicated instructions to total instructions is shown in parentheses in the second column. The "Predicate Nullified" column shows the number of dynamic instructions nullified at run time by a false predicate. The ratio of nullified instructions to predicated instructions is shown in parentheses in the third column. These data provide an important measure of processor resource utilization in predicated code.

Table 8.7 Dynamic nullification frequency for predicated instructions.

Benchmark	Total	Predicated	Predicate Nullified
008.espresso	644M	186M (0.29)	124M (0.67)
022.li	33M	4287K (0.13)	1936K (0.45)
023.eqntott	892M	191M (0.21)	68M (0.36)
026.compress	108M	19M (0.18)	9733K (0.51)
052.alvinn	3604M	29M (0.01)	28M (0.97)
056.ear	11273M	906M (0.08)	479M (0.53)
072.sc	122M	21M (0.17)	13M (0.62)
cccp	3923K	706K (0.18)	383K (0.54)
cmp	921K	118K (0.13)	114K (0.97)
eqn	45M	1905K (0.04)	1357K (0.71)
grep	1647K	35K (0.02)	25K (0.71)
lex	46M	17M (0.38)	11M (0.65)
qsort	52M	15M (0.30)	7942K (0.53)
tbl	2934K	358K (0.12)	171K (0.48)
wc	1526K	1051K (0.69)	752K (0.72)
yacc	51M	11M (0.22)	7756K (0.71)
Average	-	- (0.20)	- (0.63)

The table shows that a relatively high frequency of predicated instructions is nullified. The overall average fraction of nullified instructions across the benchmarks is 63%, with as high as 7% for *052.alvinn* and *cmp*. In general, the fraction of nullified instructions is expected to be relatively high due to the large number of control flow paths and unbiased branches in many of the benchmarks. However, several of these large values are a bit misleading. For both of the aforementioned benchmarks, only a small fraction (1% and 13%) of the dynamic instructions are actually predicated. Thus, these extremely high nullification ratios are rather small when compared to the total instructions. A more serious issue seemingly occurs for the benchmark *wc*, which has 69% predicated instructions and 72% of those are nullified. For this benchmark, a significant fraction of useless instructions are executed. However, examination of the code for this benchmark shows that the majority of predicated instructions is used to fill idle processor

Table 8.8 Dynamic usage distribution of predicate comparison instruction types.

Benchmark	Single-U	Single-O	Dual-UU	Dual-OO	Dual-UO
008.espresso	0.471	0.231	0.020	0.157	0.122
022.li	0.165	0.680	0.139	0.000	0.016
023.eqntott	0.494	0.500	0.006	0.000	0.000
026.compress	0.041	0.733	0.226	0.000	0.000
052.alvinn	0.000	1.000	0.000	0.000	0.000
056.ear	0.200	0.503	0.298	0.000	0.000
072.sc	0.440	0.544	0.005	0.000	0.010
cccp	0.060	0.729	0.202	0.000	0.009
cmp	0.205	0.795	0.000	0.000	0.000
eqn	0.128	0.848	0.015	0.000	0.009
grep	0.003	0.997	0.000	0.000	0.000
lex	0.056	0.533	0.407	0.000	0.004
qsort	0.000	0.450	0.550	0.000	0.000
tbl	0.196	0.679	0.096	0.000	0.029
wc	0.286	0.143	0.000	0.000	0.571
yacc	0.209	0.650	0.025	0.000	0.116
Average	0.185	0.626	0.124	0.010	0.055

resources, while enabling branches to be eliminated. The net effect is positive performance gains despite the low resource utilization.

The usage of predicate comparison instructions is the second predicate architecture issue that is examined. Table 8.8 presents the dynamic usage distribution of the various types of predicate comparison instructions. As was described in Section 6.2.2, the predicate comparison instructions may have up to two destination operands, and each may be one of six types: unconditional, OR-type, AND-type, and their complements. The current hyperblock compilation techniques do not make use of the AND-type predicates, so there are none of these comparison types. The first two columns in the table correspond to single target unconditional compares and single target OR-type compares. For these predicate comparison instructions, no other comparison with the the same source operands could be found to combine into a dual target comparison. The last three columns correspond to dual target comparisons with the specified destination pair types. Note that the "Dual-UO" includes both unconditional/OR-type and OR-type/unconditional comparisons.

From the table, the most apparent result is the surprisingly high utilization of single target compares, about 80% on average. This indicates that there are a significant number of single-sided "if" statements which are if-converted. If-conversion of this nature always generates single target compares since there is no matching complement condition to pair up. Also, the branch combining optimization is a major source of single target OR-type compares. This optimization converts a series of branches into OR-type compares. Only a single target is required because only the taken branch condition is captured in the predicate.

Within the dual target compares, the unconditional/unconditional comparisons occur most frequently. This result should be expected though. If-conversion of simple if-then-else statements utilizes these comparison types almost exclusively. Therefore, their frequency is likely to be high. The least utilized dual target compares are the OR-type/OR-type. In general, because the complement of an OR-type predicate does not have much value, one can expect this comparison instruction to be infrequently utilized. The only benchmark where the OR-type/OR-type compares occur, a measurable fraction of the instructions is *008.espresso*. The hyperblocks that are formed with loop peeling in this benchmark provide several opportunities to make use of this dual target comparison.

The final component of the evaluated predicate architecture is the number of predicate registers utilized by hyperblock compilation techniques. Table 8.9 shows the dynamic usage distribution of predicate registers for the benchmarks. The entries in the table contain the weighted fraction of hyperblocks which require the specified range of predicate registers. The

Table 8.9 Dynamic usage distribution of predicate registers.

Benchmark	1-8	9-16	17-24	25-32	33-40	41-48
008.espresso	0.363	0.158	0.299	0.000	0.180	0.000
022.li	0.933	0.067	0.000	0.000	0.000	0.000
023.eqntott	0.119	0.014	0.859	0.000	0.008	0.000
026.compress	0.686	0.000	0.314	0.000	0.000	0.000
052.alvinn	1.000	0.000	0.000	0.000	0.000	0.000
056.ear	0.369	0.360	0.180	0.000	0.090	0.000
072.sc	0.537	0.272	0.191	0.000	0.000	0.000
cccp	0.782	0.000	0.032	0.044	0.141	0.000
cmp	0.000	1.000	0.000	0.000	0.000	0.000
eqn	0.879	0.120	0.000	0.000	0.000	0.000
grep	1.000	0.000	0.000	0.000	0.000	0.000
lex	0.369	0.020	0.028	0.000	0.583	0.000
qsort	1.000	0.000	0.000	0.000	0.000	0.000
tbl	0.938	0.034	0.025	0.003	0.000	0.000
wc	0.000	0.000	0.000	0.000	0.000	1.000
yacc	0.808	0.105	0.005	0.007	0.074	0.000
Average	0.611	0.134	0.121	0.003	0.067	0.063

predicate requirements are calculated by performing local register allocation for the predicate registers on each hyperblock. The register allocator has an unlimited supply of predicate registers, but attempts to utilize the minimal number of required registers such that no spilling is necessary. Thus, the number of required predicate registers is the maximum number of overlapping predicate lifetimes in a hyperblock.

From the table, the majority of hyperblocks requires only a small number of predicates. This is most obvious for benchmarks such as *052.alvinn*, *grep*, and *qsort*, where no hyperblocks in the entire benchmark require more than eight predicates. These benchmarks are characterized by simple control flow structures in the important portions of the code, which, in turn, require few predicates to represent. At the other extreme, there is a significant overall fraction (13%) of hyperblocks that require 33 or more predicates. The two benchmarks with the highest predicate requirements are *wc* and *lex*. For these benchmarks, if-conversion is utilized to remove complex control flow and thus requires more predicates. Subsequent optimizations such as loop unrolling further increase the predicate register requirements by creating larger hyperblocks with more overlapped control constructs. It is expected that more benchmarks will begin to show predicate usage distributions close to *wc* and *lex* with the application of more aggressive hyperblock optimizations and the use of new predicate optimizations such as control height reduction [95],[96].

8.2.7 Current level of performance

The results from Chapter 5 showed the performance improvement potential of superblocks, speculative execution in superblocks, and superblock ILP optimizations. Compared to traditional basic block compilation techniques, large performance improvements were observed. At the same time, several factors indicated that speculative execution in superblocks has performance limitations. With superblocks, there was a large fraction of under-utilized resources in wide issue processors. The limited ILP in superblocks was attributed to superblocks typically not being large enough and having frequently taken side exits. Also, the superblock performance relies heavily on support to execute multiple branches per cycle.

These factors motivated the investigation of predicated execution and hyperblock compilation techniques. The results presented in this chapter show that a large amount of success was achieved at improving performance by overcoming the superblock limitations. In this section, the experiments utilized to illustrate the superblock limitations are repeated to more clearly indicate the level of success to which limitations are addressed. These results also allow some perspective to be placed on the current performance level.

Table 8.10 Hyperblock characteristics.

Benchmark	HB size	HB Completion Ratio 1.00	≥ 0.90	≥ 0.70	≥ 0.50	≥ 0.30
008.espresso	80.4	0.77	0.79	0.82	0.85	0.88
022.li	20.8	0.68	0.67	0.79	0.82	0.89
023.eqntott	38.1	0.85	0.90	0.91	0.92	0.93
026.compress	42.9	0.54	0.57	0.67	0.83	0.93
052.alvinn	88.2	0.94	0.95	0.97	0.98	0.99
056.ear	86.0	0.84	0.88	0.96	0.98	0.99
072.sc	29.3	0.70	0.72	0.78	0.87	0.93
cccp	26.7	0.53	0.54	0.60	0.66	0.91
cmp	75.5	0.99	0.99	0.99	0.99	0.99
eqn	34.0	0.54	0.55	0.62	0.69	0.84
grep	49.7	0.54	0.61	0.70	0.76	0.91
lex	51.3	0.66	0.69	0.73	0.84	0.94
qsort	40.6	0.68	0.68	0.69	0.85	0.86
tbl	20.1	0.75	0.78	0.83	0.87	0.91
wc	131.1	0.99	0.99	0.99	0.99	0.99
yacc	33.9	0.61	0.64	0.69	0.74	0.86
Average	53.0	0.73	0.75	0.80	0.85	0.92

The characteristics of the hyperblocks generated by the compiler are presented in Table 8.10. The first column of data contains the average number of instructions in each hyperblock weighted by the execution frequency of the hyperblock. These data show the average size of the hyperblocks presented to the scheduler for each benchmark. Note that the data in the table include all blocks in the programs, not just the hyperblocks. Comparing the hyperblock results with the previous superblock results (Table 5.11) shows the hyperblock compilation techniques are highly effective at increasing the number of instructions available to the scheduler. On average, the block size increased from 34.5 to 53.0 instructions. Most importantly, several of the benchmarks with the smallest superblocks were increased dramatically. For example, the average size of the superblocks in *023.eqntott* was 19.3 instructions. With hyperblocks, the size was approximately doubled to 38.1 instructions.

The benchmarks which showed largest increases in block sizes also achieved large performance improvements. Examples of such benchmarks include *008.espresso*, *023.eqntott*, and *wc*. The increased number of instructions the scheduler has to consider clearly increases the efficiency of the resultant schedule. The inverse of this behavior is also true. For benchmarks which did not have a significant increase in block size, such as *022.li* and *tbl*, little performance improvement was observed with predicated execution. Clearly, the hyperblock techniques were less effective at creating large hyperblocks in these benchmarks, which results in only small performance gains over superblock code.

The remaining data in Table 8.10 show the weighted average completion ratio of hyperblocks. The completion ratio, as previously defined, is the percentage of time a specified fraction of the hyperblock is executed. For example, the "$\geq$0.90" column for *008.espresso* indicates 79% of the time, 90% or more of the instructions in the hyperblocks are executed. The previous superblock data showed that a large fraction of the time superblocks are exited prematurely though a side exit. The goal with hyperblocks was to increase the completion ratio by overlapping the execution of multiple paths in a single hyperblock. Table 8.10 again shows that the hyperblock techniques were indeed successful at this goal. The average fraction of executions which execute more than 90% of the blocks is raised from 61% to 75%. One of the most notable increases occurs for *008.espresso*, in which the fraction of executions which have a 90% completion ratio is raised from 39% to 79%. Similarly for *072.sc*, the 90% completion ratio is raised from 40% to 72%. The formation of hyperblocks by overlapping the execution of all the important paths enables the compiler to effectively form blocks with small early exit probabilities.

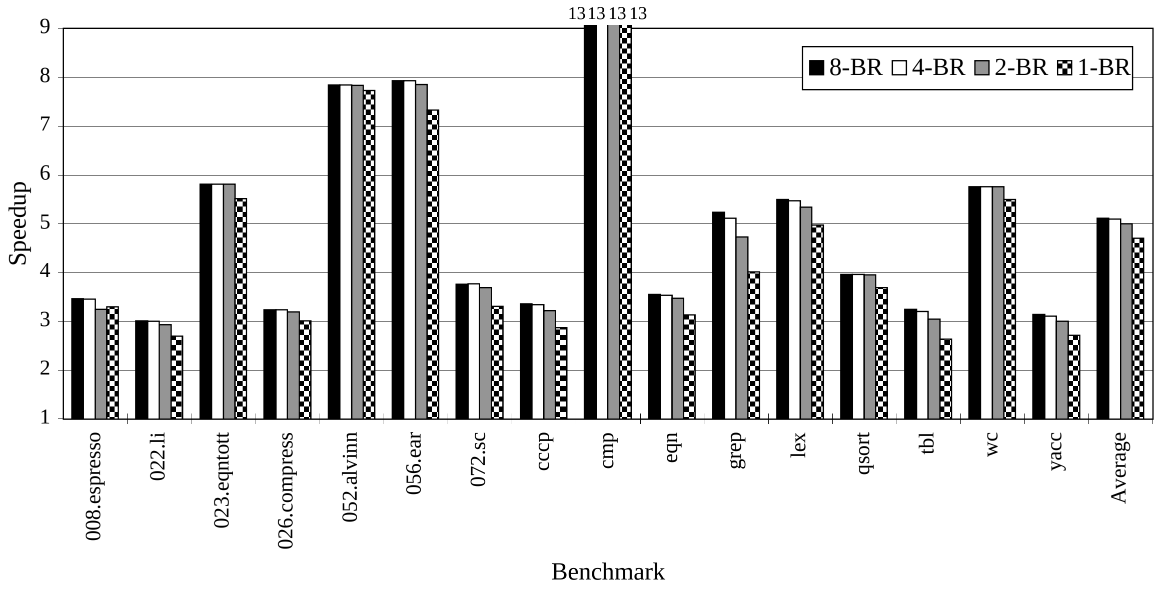


Figure 8.12 Effect of reducing the maximum number of branches executed per cycle for an issue-8 processor with predicated execution support.

A second limiting factor of superblocks is the large performance loss that occurs when the number of branch resources is reduced. Figure 8.12 presents the speedup of an issue-8 processor with predicated execution over the base processor as the maximum number of branches per cycle is reduced from eight to one. Comparing the hyperblock results with the previous superblock results (Figure 5.12) shows that the performance has become substantially more stable with predicated execution. With superblocks, much of the large performance gains that were achieved for an issue-8 processor are lost when the processor can only execute 1 or 2 branches each cycle. The branch resources quickly become the performance bottleneck as their availability is decreased. However in the hyperblock code, the high performance level is closely maintained regardless of the number of branches. The ability of the compiler to eliminate branches with predicated execution clearly removes the branch resource bottleneck.

Table 8.11 Dynamic instruction mix with hyperblock compilation techniques.

Benchmark	Load	Store	IALU	FALU	Branch
008.espresso	0.187	0.027	0.722	0.000	0.065
022.li	0.299	0.140	0.329	0.000	0.231
023.eqntott	0.234	0.005	0.702	0.000	0.058
026.compress	0.162	0.067	0.646	0.000	0.124
052.alvinn	0.341	0.090	0.193	0.349	0.026
056.ear	0.197	0.173	0.142	0.438	0.050
072.sc	0.211	0.023	0.614	0.012	0.140
cccp	0.161	0.045	0.592	0.000	0.202
cmp	0.225	0.000	0.746	0.000	0.029
eqn	0.172	0.104	0.537	0.000	0.188
grep	0.183	0.073	0.628	0.000	0.115
lex	0.187	0.251	0.495	0.000	0.067
qsort	0.230	0.225	0.409	0.000	0.135
tbl	0.276	0.029	0.483	0.000	0.211
wc	0.061	0.000	0.811	0.000	0.129
yacc	0.202	0.041	0.615	0.000	0.142
Average	0.208	0.081	0.541	0.050	0.120

The effect of hyperblock compilation techniques on the dynamic instruction mix is presented in Table 8.11. Instructions are broken into five categories, memory load, memory store, integer ALU, floating-point ALU, and branch. The run-time value of the predicate for instructions is not considered when computing the data in this table. Comparing the the hyperblock instruction mix and the superblock instruction mix (Table 5.12) clearly shows the reduction in the fraction of branch instructions. The average fraction of branches is reduced from 35.1% with superblocks to 12.0% with hyperblocks. The consequence of this large reduction in branches is an almost equally large increase in integer ALU instructions, 28.1% to 54.1%. The introduction of new instructions to manipulate predicates as well as the additional arithmetic instructions obtained from overlapping the execution of multiple control paths are the major sources for the increase of integer ALU instructions.

A somewhat surprising result is the uniform reduction in the fraction of load instructions with hyperblock compilation techniques. The average fraction of loads is dropped from 23.6% to 20.8%, and in all but one benchmark a reduction is observed. A portion of this behavior is because of the increase in the total instructions for hyperblock code (see Table 8.1). However, the primary cause of this behavior is the reduced amount of load speculation that is employed in the hyperblock code. With superblock code, a large fraction of the load instructions are speculated and generally speculated over many branches. Thus the execution frequency of loads is substantially increased. With hyperblock code, load speculation primarily occurs with predicate promotion rather than code motion across branches. The instruction mix data presented in the table does not take the run-time predicate into account. Thus, predicate promotion does not increase the fraction of load instructions. The net result is that the large increases in load frequency caused by speculation are observed to a lesser extent in the hyperblock code. Hence, an overall reduction in the fraction of dynamic instructions that are loads is obtained.

The utilization of the available processor resources that is achieved using hyperblock compilation techniques is evaluated in Figure 8.13. The average fraction of executed and unused IPC is plotted for an issue-8 processor capable of executing at most one branch per cycle. The corresponding experiment for superblocks is presented in Figure 5.11. Comparing the superblock and hyperblock results shows that the executed IPC has increased noticeably for most of the benchmarks. For example, the executed IPC for *008.espresso* has increased from 3.2 to 5.8. Overall the average executed IPC across the benchmarks increases from 4.5 to 5.2. It should be noted that the superblock experiment purposefully utilizes no limitation on the number of branches allowed each cycle to present the most optimistic value, whereas the hyperblock experiment limits the number of branches to one per cycle to present a more realistic value.

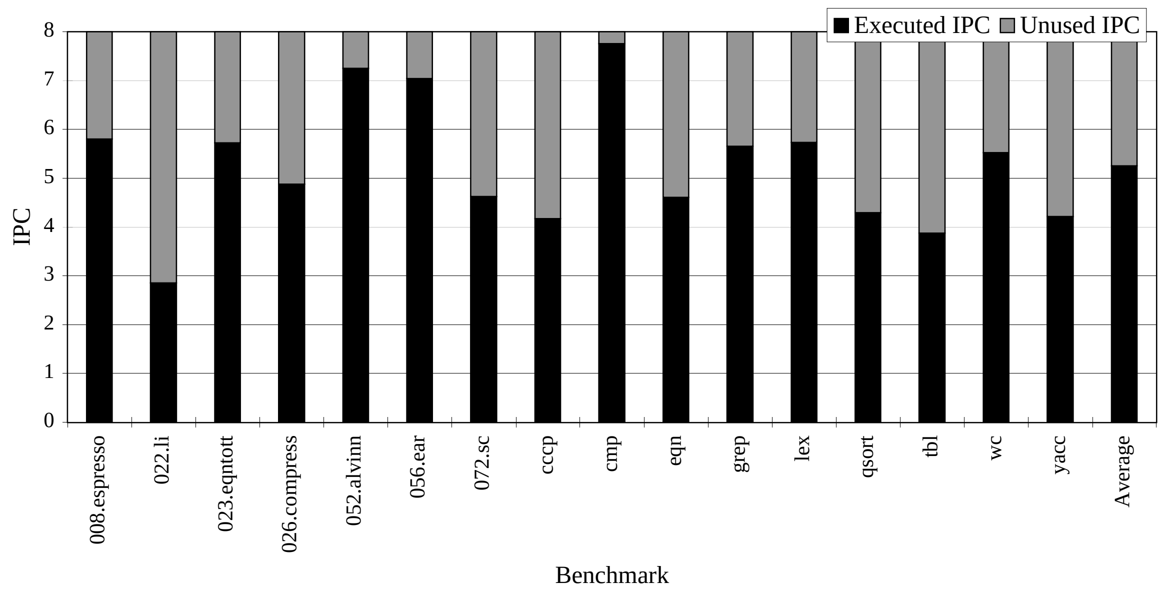


Figure 8.13 Average executed and unused instructions per cycle for an issue-8, one-branch processor using hyperblock compilation techniques.

Despite the branch limitations, the increased resource utilization shows that the hyperblock compilation techniques are effective at increasing the ILP to achieve performance improvement.

Figure 8.13 also shows that a significant fraction of idle resources still remain for many of these benchmarks. The most notable example is *022.li* in which over 5 IPC is idle for an issue-8 processor. Many performance limitations associated with data, memory, and control dependences remain in the code which limits the available ILP. Predicated execution in conjunction with hyperblock compilation techniques has made positive progress, but clearly there is a large performance potential to be gained from further ILP research.

In summary, an extensive evaluation of the effectiveness of predicated execution supported by hyperblock compilation techniques has been presented in this chapter. Substantial performance improvements were achieved compared to superblock code with aggressive speculation support. The primary sources of the improved performance are the reduction in dynamic branches, the decrease in branch mispredictions, and the ability to efficiently overlap the execution of multiple paths of control. Several issues are also evaluated, including the effects of hyperblock compilation techniques on the instruction/data caches; the relative contributions of speculation and predication to overall performance; and the utilization of the architectural components to support predicated execution. For additional experimental results on the effectiveness of predicated execution, the interested reader is referred to [106], [107], [108], and [109].

CHAPTER 9

CONCLUSION

9.1 Summary

Branch instructions pose serious difficulties for processors that exploit ILP. These problems arise for several reasons. First, branches limit code motion freedom by imposing control dependences to enforce the proper ordering conditions between branches and other instructions. Second, branches cause substantial run-time overhead from misprediction penalties. Finally, branches limit processor throughput when the branch execution bandwidth cannot keep up with the branch frequency in the instruction stream. For superscalar and VLIW processors, conventional architectural and compilation methods do not provide enough support to allow effective exploitation of ILP in the presence of branches. In this dissertation, two techniques to overcome these difficulties are investigated, speculative execution and predicated execution.

The first technique, speculative execution, allows the compiler to remove dependences between instructions and prior branches. In this manner, the compiler may eliminate control dependences to allow more instructions to be executed concurrently. Speculative execution is utilized by the compiler via an efficient structure called the superblock. Superblocks are formed using a combination of trace selection and tail duplication. Superblock formation isolates important execution paths and systematically eliminates constraints due to unimportant paths. Subsequent optimizations applied to superblocks expand the size and increase the ILP along the important execution paths. Superblock scheduling is then applied to aggressively overlap the execution of instructions in the superblock through the use of speculative execution.

Experimental results show that speculative execution along with superblock compilation techniques are highly effective at improving the performance of superscalar and VLIW processors. The largest performance gains over traditional techniques are achieved with the combination of superblock ILP optimizations and general speculation. Superblock formation alone yields only modest performance improvements by combining groups of basic blocks together into a single structure. Instruction overlap is still significantly limited by data dependences and the inability to overlap the execution of loop iterations. Superblock ILP optimizations aggressively transform the superblock to increase ILP in loops and straight-line code. As a result, the scheduler has many more opportunities to reorder instructions and achieve a compact schedule. The importance of generalized speculation is also shown by the results. Without speculation support, the scheduler is unable to take advantage of the increased ILP because control dependences limit code motion. General speculation provides the scheduler with the necessary freedom to achieve a high degree of concurrency.

The experiments also show that speculative execution in superblocks alone has several limitations. First, there are a large number of resources which cannot be filled by the superblock techniques alone. Thus, there is a wide range of potential performance improvement beyond that achieved with superblock techniques. The limited ILP in superblocks can be attributed to superblocks typically not being large enough and having frequently taken side exits. These characteristics motivate the expansion of the compiler scope from a single path of execution to overlapping multiple execution paths. The second factor is the large performance loss incurred when the number of branches that a wide-issue processor can execute is reduced. The instruction stream contains a high fraction of branches, and when the number of branches which may be executed is limited, the branch resource bottleneck is exposed. This factor motivates the need for generalized techniques to eliminate branches from the instruction stream to overcome the resource bottlenecks.

The limitations of solely performing speculative execution in superblocks lead to the second technique investigated in this dissertation, predicated execution. Predicated execution allows the compiler to completely remove some branches from the instruction stream by utilizing conditional execution. In addition, predicated execution provides an efficient interface for the compiler to overlap the execution of multiple paths of control. Predicated execution is exploited in the compiler using a structure referred to as a hyperblock. Hyperblocks are a generalized form of superblocks which take advantage of both speculative and predicated execution. Hyperblocks are formed through a sequence of steps which include basic block selection, tail duplication, and if-conversion. The goal of hyperblock formation is to intelligently select basic blocks from many different control flow paths to be merged into a single manageable structure using if-conversion. Basic blocks are systematically selected for inclusion in hyperblocks to eliminate unbiased branches, maximize ILP optimization opportunities, control the overall dependence height, and avoid over-committing processor resources.

Predicated execution also provides the opportunity for new optimization opportunities to increase the efficiency of predicated code and to perform new transformations made possible with conditional execution. In this dissertation, three important predicate-specific optimizations are introduced: predicate promotion, branch combining, and loop peeling. Predicate promotion advances the predicate of an instruction to allow the instruction to be executed before the predicate is known. Hence, predicate promotion provides support for speculation in the predicate domain. Branch combining allows infrequently taken branches to be converted into predicate comparison instructions, while the branches themselves are moved outside the hyperblock. The net result is that the compiler can drastically reduce the frequency of branch instructions in the important sections of the code. Finally, predicated loop peeling allows infrequently iterated loops to be unraveled and overlapped with surrounding code. In many cases, outer loop parallelism can be effectively exploited after peeling the innermost loops. Overall, these transformations provide the compiler with powerful tools for increasing the ILP with predicated execution.

A detailed evaluation of predicated execution using hyperblock compilation techniques is presented in this dissertation. The results show that substantial performance improvements are achieved over superblock code with aggressive speculation support. There are three major sources of performance gain. First, the ability to remove a large fraction of the branches from the instruction stream alleviates the branch resource bottleneck. Second, removing unbiased branches decreases the number of branch mispredictions which in turn reduces the time the processor is stalled while branch mispredictions are repaired. Finally, the ability to overlap the execution of multiple control paths increases the ILP the compiler can expose in the presence of branches. Overall, the results show that the combination of speculative and predicated execution are highly effective for exploiting ILP in nonnumeric applications.

9.2 Future Research

The work presented in this dissertation motivates several promising opportunities for future research. These include the areas of hyperblock formation, predicate-specific optimization, and generalized acyclic scheduling.

The first opportunity is in the area of hyperblock formation. The hyperblock formation techniques presented in this dissertation demonstrate the effectiveness of the approach. The formation algorithm works well for innermost loops. However, for general acyclic code, inefficient hyperblocks are often produced. The primary reason for this behavior is that the external boundaries of loops are firmly designated by the loop structure itself. With these in place, the formation algorithm can efficiently identify the set of blocks within the loop to convert into a hyperblock. On the other hand, for general acyclic code, such firm boundaries do not exist. As a result, the hyperblocks tend to grow too far along particular paths of control, and not far enough along others. New techniques to form efficient hyperblocks in acyclic code will likely increase the effectiveness of the hyperblock techniques across all the benchmarks.

Another subject within hyperblock formation that was not examined in this dissertation is the issue of forming hyperblocks without profile information. Currently, profile information is essential to identifying the important control paths in the code. Hyperblock formation utilizes the profile information along with several static measures, such as dependence height, instruction count, and hazard existence, to create hyperblocks. Instead of profile information, the hyperblock formation techniques could utilize static branch prediction techniques or other static measures in conjunction those static measures currently utilized. Since many users are often unable or unwilling to provide profile information to the compiler, the ability to form efficient hyperblocks without profile information certainly expands the potential use of hyperblock techniques by a large margin.

A second area of future research is predicate-specific optimization. In this dissertation, several such optimizations were introduced, including predicate promotion, branch combining, predicated loop peeling, and instruction merging. These optimizations target improving the quality of the predicated code and providing new transformation capabilities that are made possible with support for predicated execution. Overall, the predicate-specific optimizations utilized in this dissertation greatly improved the overall performance of the hyperblock code. These optimizations are probably the beginnings of an extremely important area of research. The ability to conditionally execute instructions with predicates rather than branches provides the compiler with a powerful tool. New optimizations which utilize predicated execution to reduce dependence height, increase code motion freedom, and reduce resource contention will be highly important for future ILP compilers. Also, the introduction of overlapped paths of control in a single block introduces new challenges to the compiler to improve the quality of such code. Compilers for architectures that have predicated execution will require efficient techniques to handle predicates. Further research in the area of predicate-specific optimizations is definitely warranted.

The final area of future research motivated by this dissertation is that of generalized acyclic scheduling. In this dissertation, superblock scheduling and hyperblock scheduling are utilized exclusively. Superblock scheduling provides an efficient paradigm to effectively schedule across basic block boundaries along a single path of control. Hyperblock scheduling is a generalization of superblock scheduling which supports predicated execution. However, the limitation of the hyperblock approach is that scheduling is restricted to a single net of predicated code. Code motion across hyperblocks is not supported. New scheduling techniques which attack program graphs containing a mixture of control flow and predicated execution are required. The ability to aggressively move code across hyperblocks will substantially increase the ILP exposed by the scheduler.

REFERENCES

[1] 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.

[2] 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.

[3] M. A. Schuette and J. P. Shen, "An instruction-level performance analysis of the Multiflow Trace 14/300," in *Proceedings of the 24th International Workshop on Microprogramming and Microarchitecture*, pp. 2–11, November 1991.

[4] J. E. Smith, "A study of branch prediction strategies," in *Proceedings of the 8th International Symposium on Computer Architecture*, pp. 135–148, May 1981.

[5] T. Ball and J. R. Larus, "Branch prediction for free," in *Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation*, pp. 300–313, June 1993.

[6] R. E. Hank, S. A. Mahlke, R. A. Bringmann, J. C. Gyllenhaal, and W. W. Hwu, "Superblock formation using static program analysis," in *Proceedings of the 26th Annual International Symposium on Microarchitecture*, December 1993.

[7] S. McFarling and J. Hennessy, "Reducing the cost of branches," in *Proceedings of the 13th International Symposium on Computer Architecture*, pp. 396–403, June 1986.

[8] W. W. Hwu, T. M. Conte, and P. P. Chang, "Comparing software and hardware schemes for reducing the cost of branches," in *Proceedings of the 16th International Symposium on Computer Architecture*, pp. 224–233, May 1989.

[9] J. A. Fisher and S. M. Freudenberger, "Predicting conditional branch directions from previous runs of a program," in *Proceedings of 5th International Conference on Architectual Support for Programming Languages and Operating Systems*, pp. 85–95, October 1992.

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

[11] J. Lee and A. J. Smith, "Branch prediction strategies and branch target buffer design," *IEEE Computer*, pp. 6–22, January 1984.

[12] T. Y. Yeh and Y. N. Patt, "Two-level adaptive training branch prediction," in *Proceedings of the 24th Annual International Symposium on Microarchitecture*, pp. 51–61, November 1991.

[13] T. Y. Yeh and Y. N. Patt, "Alternative implementations of two-level adaptive branch prediction," in *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pp. 124–134, May 1992.

[14] S. Pan, K. So, and J. T. Rahmeh, "Improving the accuracy of dynamic branch prediction using branch correlation," in *Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 76–84, October 1992.

[15] D. W. Wall, "Limits of instruction-level parallelism," in *Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 176–188, April 1991.

[16] M. Butler, T. Yeh, Y. Patt, M. Alsup, H. Scales, and M. Shebanow, "Single instruction stream parallelism is greater than two," in *Proceedings of the 18th International Symposium on Computer Architecture*, pp. 276–286, May 1991.

[17] P. Y. Hsu and E. S. Davidson, "Highly concurrent scalar processing," in *Proceedings of the 13th International Symposium on Computer Architecture*, pp. 386–395, June 1986.

[18] B. R. Rau, D. W. L. Yen, W. Yen, and R. A. Towle, "The Cydra 5 departmental supercomputer," *IEEE Computer*, vol. 22, pp. 12–35, January 1989.

[19] J. R. Allen, K. Kennedy, C. Porterfield, and J. Warren, "Conversion of control dependence to data dependence," in *Proceedings of the 10th ACM Symposium on Principles of Programming Languages*, pp. 177–189, January 1983.

[20] J. C. Park and M. S. Schlansker, "On predicated execution," Tech. Rep. HPL-91-58, Hewlett-Packard Laboratories, Palo Alto, CA, May 1991.

[21] S. A. Mahlke, D. C. Lin, W. Y. Chen, R. E. Hank, and R. A. Bringmann, "Effective compiler support for predicated execution using the hyperblock," in *Proceedings of the 25th International Symposium on Microarchitecture*, pp. 45–54, December 1992.

[22] R. Gupta and M. L. Soffa, "Region scheduling: An approach for detecting and redistributing parallelism," *IEEE Transactions on Software Engineering*, vol. 16, pp. 421–431, April 1990.

[23] 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.

[24] D. M. Gallagher, "Memory disambiguation to facilitate instruction-level parallelism compilation," Ph.D. dissertation, Department of Electrical and Computer Engineering, University of Illinois, Urbana, IL, 1995.

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

[26] 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.

[27] 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.

[28] 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.

[29] 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.

[30] 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.

[31] 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.

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

[33] P. P. Chang, D. M. Lavery, S. A. Mahlke, W. Y. Chen, and W. W. Hwu, "The importance of prepass code scheduling for superscalar and superpipelined processors," *IEEE Transactions on Computers*, vol. 44, pp. 353–370, March 1995.

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

[35] 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.

[36] D. M. Gallagher, W. Y. Chen, S. A. Mahlke, J. C. Gyllenhaal, and W. W. Hwu, "Dynamic memory disambiguation using the memory conflict buffer," in *Proceedings of 6th International Conference on Architectual Support for Programming Languages and Operating Systems*, pp. 183–193, October 1994.

[37] 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.

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

[39] 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.

[40] 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.

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

[42] 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.

[43] 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.

[44] 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.

[45] W. W. Hwu and Y. N. Patt, "HPSm, a high performance restricted data flow architecture having minimal functionality," in *Proceedings of the 13th International Symposium on Computer Architecture*, pp. 297–306, June 1986.

[46] 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.

[47] W. M. Johnson, *Superscalar Microprocessor Design.* Englewood Cliffs, NJ: Prentice-Hall, Inc., 1991.

[48] P. Tirumalai, M. Lee, and M. Schlansker, "Parallelization of loops with exits on pipelined architectures," in *Proceedings of Supercomputing '90*, November 1990.

[49] J. Ellis, *Bulldog: A Compiler for VLIW Architectures.* Cambridge, MA: The MIT Press, 1985.

[50] P. P. Chang and W. W. Hwu, "Trace selection for compiling large C application programs to microcode," in *Proceedings of the 21st International Workshop on Microprogramming and Microarchitecture*, pp. 188–198, November 1988.

[51] 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.

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

[53] E. Morel and C. Renviose, "Global optimization by suppression of partial redundancies," *Communications of the ACM*, pp. 96–103, February 1979.

[54] L. Feigen, D. Klappholz, R. Cassazza, and X. Xue, "The revival transformation," in *Conference Record of POPL '94: 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pp. 421–434, January 1994.

[55] J. Knoop, O. Ruthing, and B. Steffen, "Partial dead code elimination," in *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation*, June 1994.

[56] D. J. Kuck, R. H. Kuhn, D. A. Padua, B. Leasure, and M. Wolfe, "Dependence graphs and compiler optimizations," in *Proceedings of the 8th ACM Symposium on Principles of Programming Languages*, pp. 207–218, January 1981.

[57] T. Nakatani and K. Ebcioglu, "Combining as a compilation technique for VLIW architectures," in *Proceedings of the 22nd International Workshop on Microprogramming and Microarchitecture*, pp. 43–55, September 1989.

[58] A. Aiken and A. Nicolau, "Optimal loop parallelization," in *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation*, pp. 308–317, June 1988.

[59] K. Anantha and F. Long, "Code compaction for parallel architectures," *Software Practice and Experience*, vol. 20, pp. 537–554, June 1990.

[60] P. G. Lowney, S. M. Freudenberger, T. J. Karzes, W. D. Lichtenstein, R. P. Nix, J. S. O'Donnell, and J. C. Ruttenberg, "The Multiflow Trace scheduling compiler," *The Journal of Supercomputing*, vol. 7, pp. 51–142, January 1993.

[61] J. C. Dehnert and R. A. Towle, "Compiling for the Cydra 5," *The Journal of Supercomputing*, vol. 7, pp. 181–227, January 1993.

[62] K. Ebcioglu, R. D. Groves, K. Kim, G. M. Silberman, and I. Ziv, "VLIW compilation techniques in a superscalar environment," in *Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation*, pp. 36–48, June 1994.

[63] J. L. Baer and D. P. Bovet, "Compilation of arithmetic expressions for parallel computations," in *Proceedings of IFIP Congress*, pp. 34–46, 1968.

[64] D. J. Kuck, *The Structure of Computers and Computations.* New York, NY: John Wiley and Sons, 1978.

[65] R. M. Stallman, *Using and porting GNU CC.* Free Software Foundation, Inc., 1989.

[66] P. P. Chang, S. A. Mahlke, W. Y. Chen, N. J. Warter, and W. W. Hwu, "IMPACT: An architectural framework for multiple-instruction-issue processors," in *Proceedings of the 18th International Symposium on Computer Architecture*, pp. 266–275, May 1991.

[67] P. P. Chang, N. J. Warter, S. A. Mahlke, W. Y. Chen, and W. W. Hwu, "Three architectural models for compiler-controlled speculative execution," *IEEE Transactions on Computers*, vol. 44, pp. 481–494, April 1995.

[68] B. R. Rau and C. D. Glaeser, "Some scheduling techniques and an easily schedulable horizontal architecture for high performance scientific computing," in *Proceedings of the 20th Annual Workshop on Microprogramming and Microarchitecture*, pp. 183–198, October 1981.

[69] K. Ebcioglu, "A compilation technique for software pipelining of loops with conditional jumps," in *Proceedings of the 20th Annual Workshop on Microprogramming and Microarchitecture*, pp. 69–79, December 1987.

[70] M. D. Smith, M. S. Lam, and M. A. Horowitz, "Boosting beyond static scheduling in a superscalar processor," in *Proceedings of the 17th International Symposium on Computer Architecture*, pp. 344–354, May 1990.

[71] 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.

[72] 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.

[73] 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.

[74] G. R. Beck, D. W. Yen, and T. L. Anderson, "The Cydra 5 minisupercomputer: Architecture and implementation," *The Journal of Supercomputing*, vol. 7, pp. 143–180, January 1993.

[75] Hewlett-Packard Company, Cupertino, CA, *PA-RISC 1.1 Architecture and Instruction Set Reference Manual*, 1990.

[76] D. L. Weaver and T. Germond, *The SPARC Architecture Manual.* SPARC International, Inc., Menlo Park, CA, 1994.

[77] 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.

[78] 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.

[79] G. Kane, *MIPS R2000 RISC Architecture.* Englewood Cliffs, NJ: Prentice-Hall, Inc., 1987.

[80] D. I. August, B. L. Deitrich, and S. A. Mahlke, "Sentinel scheduling with recovery blocks," Tech. Rep. CRHC-95-05, Center for Reliable and High-Performance Computing, University of Illinois, Urbana, IL, February 1995.

[81] J. R. Goodman and W. C. Hsu, "Code scheduling and register allocation in large basic blocks," in *Proceedings of the 1988 International Conference on Supercomputing*, pp. 442–452, July 1988.

[82] S. Freudenberger and J. Ruttenberg, "Phase ordering of register allocation and instruction scheduling," in *Code Generation - Concepts, Tolls, Techniques*, May 1991.

[83] 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.

[84] W. Y. Chen, P. P. Chang, T. M. Conte, and W. W. Hwu, "The effect of code expanding optimizations on instruction cache design," *IEEE Transactions on Computers*, vol. 42, pp. 1045–1057, September 1993.

[85] *Microprocessor Forum*, (San Francisco, CA), October 1994.

[86] *Hot Chips VII*, (Stanford University, Palo Alto, CA), August 1995.

[87] J. C. Dehnert, P. Y. Hsu, and J. P. Bratt, "Overlapped loop support in the Cydra 5," in *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 26–38, April 1989.

[88] A. V. Someren and C. Atack, *The ARM RISC Chip, A Programmer's Guide.* Reading, MA: Addison-Wesley Publishing Company, 1994.

[89] Digital Equipment Corporation, *Alpha Architecture Handbook.* Maynard, MA: Digital Equipment Corporation, 1992.

[90] Intel Corporation, Mt. Prospect, IL, *Pentium Pro Family Developer's Manual*, 1996.

[91] D. S. Blickstein, P. W. Craig, C. S. Davidson, R. N. Faiman, K. D. Glossop, R. B. Grove, S. O. Hobbs, and W. B. Noyce, "The GEM optimizing compiler system," *Digital Technical Journal*, vol. 4, no. 4, pp. 121–136, 1992.

[92] H. Zima and B. Chapman, *Supercompilers for Parallel and Vector Computers.* Reading, MA: Addison-Wesley Publishing Company, 1991.

[93] J. E. Smith and A. R. Pleszkun, "Implementation of precise interrupts in pipelined processors," in *Proceedings of the 12th Annual International Symposium on Computer Architecture*, pp. 36–44, June 1985.

[94] G. S. Sohi and S. Vajapeyam, "Instruction issue logic for high-performance interruptable pipelined processors.," in *Proceedings of the 14th Annual Symposium on Computer Architecture.*, pp. 27–34, June 1987.

[95] M. Schlansker, V. Kathail, and S. Anik, "Height reduction of control recurrences for ILP processors," in *Proceedings of the 27th International Symposium on Microarchitecture*, pp. 40–51, December 1994.

[96] M. Schlansker and V. Kathail, "Critical path reduction for scalar programs," in *Proceedings of the 28th International Symposium on Microarchitecture*, pp. 57–69, December 1995.

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

[98] E. Sprangle and Y. Patt, "Facilitating superscalar processing via a combined static/dynamic register renaming scheme," in *Proceedings of the 27th International Symposium on Microarchitecture*, pp. 143–147, December 1994.

[99] S. A. Ziegler, "Aggressive hardware support for predicated execution in out-of-order execution superscalar processors," M.S. thesis, Department of Electrical and Computer Engineering, University of Illinois, Urbana, IL, 1993.

[100] J. Ferrante, K. J. Ottenstein, and J. D. Warren, "The program dependence graph and its use in optimization," *ACM Transactions on Programming Languages and Systems*, vol. 9, pp. 319–349, July 1987.

[101] D. C. Lin, "Compiler support for predicated execution in superscalar processors," M.S. thesis, Department of Electrical and Computer Engineering, University of Illinois, Urbana, IL, 1992.

[102] N. J. Warter, S. A. Mahlke, W. W. Hwu, and B. R. Rau, "Reverse if-conversion," in *Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation*, pp. 290–299, June 1993.

[103] R. C. Johnson and M. S. Schlansker, "Analysis techniques for predicated code," Tech. Rep. to appear, Hewlett-Packard Laboratories, Palo Alto, CA, 1996.

[104] A. E. Eichenberger and E. S. Davidson, "Register allocation for predicated code," in *Proceedings of 28th Annual International Symposium on Microarchitecture*, pp. 180–191, December 1995.

[105] D. I. August, "Hyperblock performance optimizations for ILP processors," M.S. thesis, Department of Electrical and Computer Engineering, University of Illinois, Urbana, IL, 1996.

[106] D. N. Pnevmatikatos and G. S. Sohi, "Guarded execution and branch prediction in dynamic ILP processors," in *Proceedings of the 21st International Symposium on Computer Architecture*, pp. 120–129, April 1994.

[107] S. A. Mahlke, R. E. Hank, R. A. Bringmann, J. C. Gyllenhaal, D. M. Gallagher, and W. W. Hwu, "Characterizing the impact of predicated execution on branch prediction," in *Proceedings of the 27th International Symposium on Microarchitecture*, pp. 217–227, December 1994.

[108] G. Tyson, "The effects of predicated execution on branch prediction," in *Proceedings of the 27th International Symposium on Microarchitecture*, pp. 196–206, December 1994.

[109] S. A. Mahlke, R. E. Hank, J. E. McCormick, D. I. August, and W. W. Hwu, "A comparison of full and partial predicated execution support for ILP processors," in *Proceedings of the 22nd International Symposium on Computer Architecture*, pp. 138–149, June 1995.

VITA

Scott Alan Mahlke was born on May 12, 1967, in Carbondale, Illinois. He pursued his undergraduate studies at the University of Illinois in Urbana, Illinois, where he received the B.S. degree in Electrical Engineering in May of 1988. In the fall of 1988, he began his graduate studies at the University of Illinois. During his graduate tenure, he was a member of the Center for Reliable and High-Performance Computing and the IMPACT project directed by Professor Wen-mei Hwu. He completed the M.S. degree in Electrical Engineering in 1991. After completing the Ph.D. work in 1995, he joined Hewlett-Packard Laboratories in Palo Alto, California. He is currently a member of the compiler and architecture research group at HP Laboratories.