STRUCTURAL AND STATIC ANALYSIS TECHNIQUES FOR ENHANCING COMPILER SUPPORT OF PREDICATED EXECUTION

BY

KEVIN MICHAEL CROZIER

B.S., Rensselaer Polytechnic Institute, 1997

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Electrical Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 1999

Urbana, Illinois

ACKNOWLEDGMENTS

I would first like to thank my advisor, Professor Wen-mei Hwu, for his support of my research activities. I appreciate the opportunity to develop my research skills within the IMPACT group.

This investigation would not have been possible without the support of both past and present members of the IMPACT group. I would like to especially thank three current members, David August, John Sias, and Dan Connors. I would like to thank David for answering my questions and being my mentor when I was new to the group. Furthermore, David's guidance and suggestions helped greatly in the preparation of this thesis. I wish to express my gratitude to John for ideas and assistance throughout my academic career along with a plentiful supply of ice cream sandwiches. Finally, I wish to thank Dan Connors for providing innumerable invaluable suggestions while I was working on this thesis. Dan took time away from his own work to offer many helpful comments, without which this thesis would not have reached its final form. I would also like to thank past IMPACT member Scott Mahlke for designing the current IMPACT hyperblock framework. It was his work that this thesis is built upon, and without his framework and insight, this thesis would not have been possible.

I also would like to thank my many friends for making my graduate school life survivable. I especially want to thank Patrick Eaton for being a great friend, colleague, and officemate while here in Illinois. His tools certainly made IMPACT a friendlier system to use and his humor made Illinois a friendlier place to live. I owe a great amount of thanks to Greg Chiocco, Tim Smith, Seth Bumpurs, Mike Brogioli, and Mike Adams for supporting me throughout my career at RPI and here at Illinois.

Finally, and most importantly, I extend my deepest gratitude and appreciation to my parents. I would like to thank my mother for always encouraging me to do that little extra and my dad for bringing home that first computer because without that, and their love, support, and guidance, I would not be where I am today.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

The performance of modern superscalar and very long instruction word (VLIW) processors depends on their ability to execute multiple instructions per cycle. These processors contain multiple data paths and multiple functional units to concurrently execute independent instructions from the instruction stream. In order to realize their performance potential, these processors demand that increasing levels of instruction-level parallelism (ILP) be exposed by the compiler. Unfortunately, recent studies have shown that conventional optimization and scheduling methods cannot expose enough parallelism for full utilization of these processors [1].

One of the major challenges to increasing the available ILP is overcoming the limitation imposed by the execution of branch instructions. This occurs for several reasons. First, the amount of ILP in basic blocks is very limited because of the small number of instructions in the block. In nonnumeric programs, on average 20% to 30% of instructions are branches, and this translates to between three and five instructions per basic block. Branch prediction is one solution that has been studied quite extensively. Proper branch prediction can increase the number of instructions available to the scheduler, allowing it to expose greater amounts of ILP. However, if a branch is mispredicted, a substantial performance penalty can be incurred [2].

In addition, branches can actually place an upper limit on the amount of ILP that can be exposed in the instruction stream. Under the assumption that 25% of instructions are branches in an instruction stream, an eight issue superscalar would need to execute two branches per cycle to sustain full utilization. However, handling multiple branch instructions leads to very complex pipeline and branch prediction hardware. Therefore, an eight issue processor's effective performance would be limited to four if it can only execute one branch per cycle.

Predication is one technique used to overcome the limitation caused by branches. Predication 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 an instruction [3],[4]. 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 [5],[6]. Predication can eliminate frequently mispredicted branches and reduce the penalties incurred by branch prediction misses. Eliminating branches also reduces the need for wide-issue processors to execute multiple branches per cycle. Finally, predication allows the compiler to expose multiple paths of execution to the processor.

The *hyperblock* is a structure used by the compiler to take advantage of predicated execution. A hyperblock groups together a set of related basic blocks that cover multiple control paths for optimization and scheduling. In the past, hyperblocks have been formed by enumerating all possible control flow paths through a region of code. This method sometimes leads to the formation of nonoptimal hyperblocks. Increased schedule height and excessive tail duplication are the primary problems observed in these nonoptimal hyperblocks. Another issue that has not been investigated is forming hyperblocks in regions with little or no profile weight. This has occurred because the path enumeration method is not effective on regions of code that have very low or no execution weight.

This thesis investigates a new hyperblock formation algorithm that attempts to form more ideal hyperblocks by looking more at program structure and less at execution paths. This algorithm also attempts to form hyperblocks for code regions of very low weight by using static branch prediction techniques. Finally, this thesis also investigates the code size issues related to forming and optimizing predicated code.

1.1 Organization of the Thesis

This thesis is organized into six chapters. Chapter 2 presents necessary background material on predicated architectures, static branch prediction and the IMPACT compiler. Chapter 3 presents the hyperblock structure. This chapter discusses formation and predicate-specific ILP optimizations. Chapter 4 describes the current path enumeration algorithm used in the IMPACT compiler, along with the newly designed block enumeration algorithm. Chapter 5 presents a set of experiments that evaluate two different methods of block selection for the hyperblock. Also presented is a set of experiments that illustrate the best performance that can be obtained using predication with today's compiler technology. Finally, in Chapter 6 conclusions and directions for future research are discussed.

CHAPTER 2

BACKGROUND

This chapter includes background information on predicated execution, static branch prediction, and the inner workings of the IMPACT compiler.

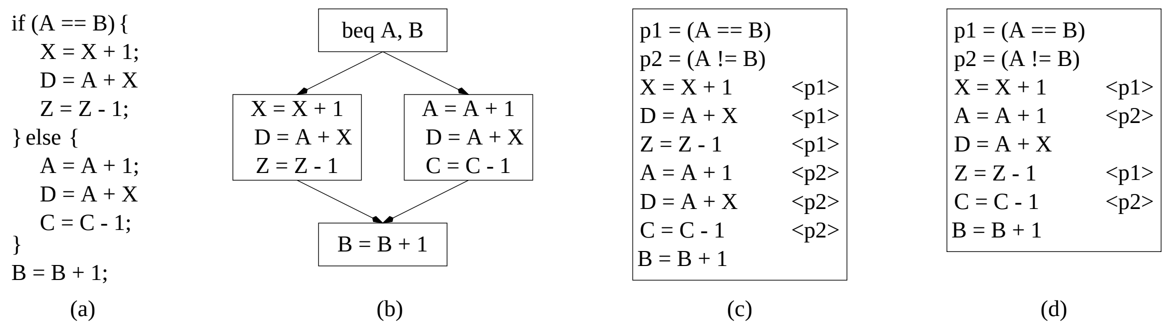
2.1 Predication

This section addresses the architectural support required to implement predicated execution. First, an overview of the predicated execution concept and its uses is presented. Next, a brief survey of predicated execution support in past and present processors is presented. Finally, the architectural extensions required to provide efficient support for predicated execution are described.

2.1.1 Overview

Predicated execution or guarded execution refers to conditional execution of instructions based on the value of a Boolean source operand, referred to as the instruction's predicate. If the predicate has the true value (a logic 1), the instruction is executed normally. If the predicate has the false value (a logic 0), the instruction is nullified and not allowed to modify the processor state. An architecture that provides predicated execution support allows the compiler to eliminate many of the conditional branches in an application.

Figure 2.1 contains a simple if-then-else construct to illustrate the concept of predicated execution. This hammock increments and decrements a number of variables based on the



```
if (A == B) {
    X = X + 1;
    D = A + X
    Z = Z - 1;
} else {
    A = A + 1;
    D = A + X
    C = C - 1;
}
B = B + 1;
```
(a)

```
beq A, B

X = X + 1        A = A + 1
D = A + X        D = A + X
Z = Z - 1        C = C - 1

B = B + 1
```
(b)

```
p1 = (A == B)
p2 = (A != B)
X = X + 1      <p1>
D = A + X      <p1>
Z = Z - 1      <p1>
A = A + 1      <p2>
D = A + X      <p2>
C = C - 1      <p2>
B = B + 1
```
(c)

```
p1 = (A == B)
p2 = (A != B)
X = X + 1      <p1>
A = A + 1      <p2>
D = A + X
Z = Z - 1      <p1>
C = C - 1      <p2>
B = B + 1
```
(d)

Figure 2.1 An example of predication: (a) simple if-then-else C code construct, (b) unpredicated code, (c) predicated code, (d) optimized code.

outcome of the comparison in the if-statement. In order to exploit predicated execution, the compiler will apply a transformation known as if-conversion. 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. In this manner, control dependences are effectively converted to data dependences.

The nonpredicated code generated for the segment is shown in Figure 2.1(b). Figure 2.1(c) shows the control flow graph after if-conversion. The *beq* instruction in Figure 2.1(b) has been replaced by the *predicate define* instructions in Figure 2.1(c). The details of the predicate define instruction will be discussed later in this section. In this example, predicate *p1* will be assigned a value of 1 if $A == B$ and 0 otherwise. Likewise, predicate *p2* will be assigned a value of 1 if $A \neq B$ and 0 otherwise. The instructions corresponding to the *if* portion of the hammock are predicated on *p1* and the instructions associated with the *else* portion of the hammock are predicated on *p2*. Figure 2.1(d) shows the predicated code after both predicate promotion and instruction merging optimizations have been applied. These optimizations are discussed in Chapter 3.

2.1.2 Survey of predicated execution in commercial systems

Predication is first seen in the Cydra 5, a VLIW multiprocessor system utilizing a directed-dataflow architecture [4],[7]. Each Cydra 5 instruction word contains seven operations, each of which is 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. After the operand fetch stage in the processor pipeline, the predicate specified by each operation is examined. If the value of the register is a logic 1, the instruction is allowed to proceed to the execution stage. Conversely, if the register value is a logic 0, the instruction is converted to a *no-op* and is essentially squashed.

The Cydra 5 also integrated predicated execution with modulo scheduling to control the prologue, epilogue, and iteration initiation of modulo scheduled loops [8],[9]. Code expansion normally required for modulo scheduling is virtually eliminated by using predicated execution in conjunction with rotating register files. Predication also allows loops with conditional branches to be efficiently modulo scheduled.

The Advanced RISC Machines (ARM) family of low power and low cost processors are targeted for embedded and multimedia applications [10]. The ARM instruction set architecture supports conditional execution of all instructions. The execution condition of each instruction is determined by looking at the 4-bit condition field in the instruction and the condition codes in the processor status register. The condition codes are set by compare instructions and the condition code specifies what comparison result (equal to, less than, greater than, etc.) the instruction should execute under. If the condition field and the compare instruction result match, the instruction executes. Otherwise the instruction is nullified. This support allows the ARM compiler to remove conditional branches from the instruction stream.

Recently, many general purpose processors are offering limited support for predicated execution. A conditional move instruction is provided in the DEC Alpha, SPARC V9, and Intel P6 processor instruction sets [11],[12],[13]. The conditional move instruction is a move instruction augmented with an additional source operand that specifies the condition under which the instruction is executed. As with a predicated move, the contents of the source register are copied to the destination register if the condition is true. The HP PA-RISC instruction set provides all branch, arithmetic, and logic instructions the capability to conditionally nullify the subsequent instruction [14]. In fact, the IMPACT compiler actually takes advantage of this feature when emulating predicated code.

The Multiflow Trace 300 series machines provided the select instruction [15]. Unlike the conditional move instruction, the select instruction always modifies the destination register. If the condition is true, the contents of the first source operand are copied to the destination. Conversely, if the condition is false, the contents of the second source operand are copied to the destination register.

Vector machines support conditional execution using mask vectors to control the execution of vectorized loops [16]. The vector mask allows a logical value to be specified so the machine can determine which instructions to execute at run time. The use of mask vectors allow vectorizing compilers to vectorize inner loops with if-then-else statements.

The recently announced Intel-Hewlett Packard co-designed IA-64 instruction set architecture will be the first general purpose architecture to support full predication [17]. The architecture includes a full set of predicate defining instructions, a separate predicate register file, and the addition of a predicate operand to each instruction. The IA-64 architecture also combines predication and rotating register files to support efficient modulo scheduling.

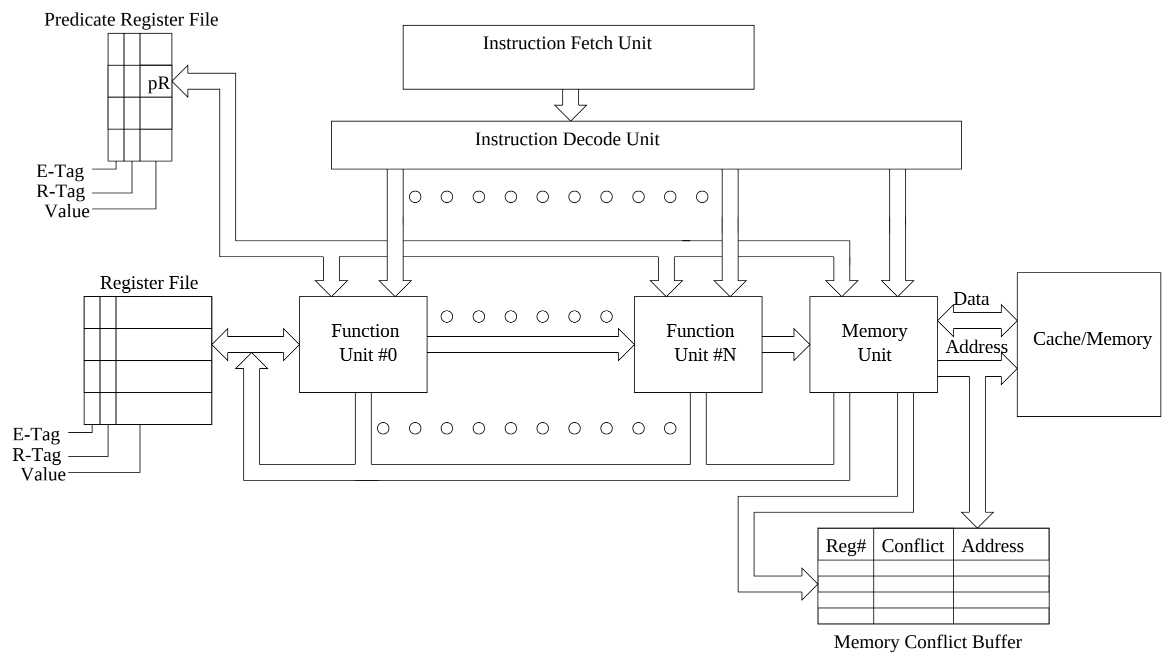


Figure 2.2 An EPIC style architecture with full predication support.

2.1.3 Architectural support for predicate execution

This thesis will focus on the full predication model that is based on the Cydra 5 and HP Labs PlayDoh architectures [4],[18]. The HP Labs PlayDoh architecture is a parameterized EPIC architecture intended to support public research on ILP architectures and compilation. An architecture supporting full predication must have four components: a predicate register file for holding the 1-bit predicate values, an additional source operand for each instruction to specify the predicate, a set of predicate defining instructions, and a conditional execution stage that can nullify instructions. A theoretical explicitly parallel instruction computing (EPIC) style architecture with full predication is shown in Figure 2.2. This architecture supports in-order issue of instructions to the fully pipelined functional units. Figure 2.3 shows the five-stage pipeline for this architecture.

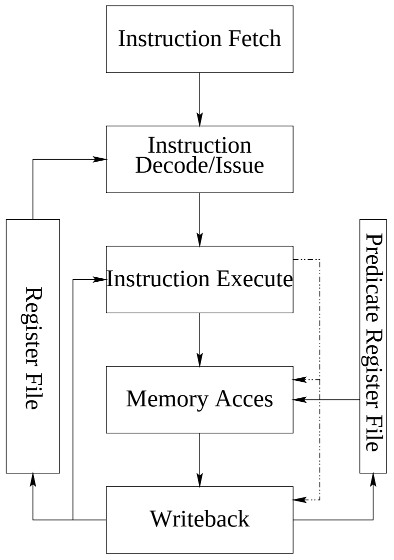


Figure 2.3 Pipeline diagram for the architecture.

The Nx1 predicate register file is added to the base architecture to hold the predicate values. This register file is included for several reasons. First, it is inefficient to use a 32-bit general purpose register to hold a 1-bit value. Register porting is a significant issue for wide issue processors. By using a register file specifically for predicates, this issue is avoided as no additional ports for the general purpose register file are needed. It should be noted that the predicate register file behaves no differently than a conventional register file. For example, the register contents must be saved during a context switch or subroutine call.

Predicate values are manipulated with a new set of instructions added to the base architecture. The most important of these instructions are the predicate defining instructions. Predicate define instructions are inserted by the compiler to generate values for control of conditional execution. PlayDoh predicate define instructions generate two Boolean values using a comparison of two source operands and a source predicate. It has the following form:

$$pD_0\_type_0,\ pD_1\_type_1 = (src_0\ \mathbf{cond}\ src_1)\ \langle pSRC \rangle.$$

Table 2.1 Predicate definition truth table.

		PlayDoh types					
pSRC	Comp	UT	UF	OT	OF	AT	AF
0	0	0	0	-	-	-	-
0	1	0	0	-	-	-	-
1	0	0	1	-	1	0	-
1	1	1	0	1	-	-	0

The instruction is interpreted as follows: pD_0 and pD_1 are the destination predicate registers; $type_0$ and $type_1$ are the predicate types of each destination; src_0 **cond** src_1 is the comparison, where **cond** can be *equal (==)*, *not equal (! =)*, *greater than (>)*, etc.; *pSRC* is the source predicate register. The value assigned to each destination is dependent on the predicate type. PlayDoh defines three predicate types, *unconditional* (UT or UF), *wired-or* (OT or OF), and *wired-and* (AT or AF). Each type can be in either normal mode or complement mode, as distinguished by the T or F appended to the type specifier (U, O, or A). Complement mode differs from normal mode only in that the condition evaluation is treated in the opposite logical sense.

For each destination predicate register, a predicate define instruction can either deposit a 1, deposit a 0, or leave the contents unchanged. The predicate type specifies a function of the source predicate and the result of the comparison that is applied to derive the resultant predicate. Table 2.1 shows the deposit rules for each of the PlayDoh predicate types in both normal and complement modes. Each entry corresponds to the result assigned to the destination predicate. Note that a "-" means that the destination is left unchanged.

As shown in the table, the unconditional types are always assigned a value. For the UT-type, the value corresponds to the logical conjunction of the source predicate and the comparison result. Conversely, the OR-type and the AND-type each only assign a value in one circumstance. The OT-type conditionally writes a 1 if both its source predicate and comparison result are true. The OR-type can be used to efficiently compute the disjunction of multiple compare conditions by accumulating terms into an initially cleared predicate register. Since the operations computing terms conditionally write the same value, they can execute in any order or even in parallel. Similarly, the AND-type can be used to compute the conjunction of multiple compare conditions by accumulating terms into an initially set predicate register.

Effective use of the OR-type and AND-type predicates previously described required that the predicates be precleared or preset, respectively. Predicate clearing and predicate setting instructions are included expressly for this purpose. Instructions for saving and restoring the contents of the predicate register file must also be provided. These instructions allow for saving and restoring predicates across subroutine calls and context switches. Providing these instructions allow the compiler to handle predicate registers in the same manner as it handles conventional register types.

At some point during the execution of the predicated instructions, the instructions with false predicates must be nullified. This nullification can occur in a variety of places along the processor pipeline. The earliest an instruction can be nullified is during the decode/issue stage. The instruction's predicate would be fetched and if the value is false, the instruction is not issued. At the other extreme, an instruction can be nullified at the write-back stage of the pipeline. If the instruction's predicate is false, the instruction is not allowed to commit its results to the register file and store instructions are prevented from entering the store buffer. Figure 2.3 shows a processor pipeline that supports this type of nullification. All of nullification schemes have their advantages and disadvantages, and complete discussion of them can be found in [19].

2.2 Static Branch Prediction

Many compiler optimizations require information about the direction of conditional branches and relative execution frequencies of basic blocks. Examples of these optimizations include superblock formation, superblock scheduling [20], hyperblock formation [21], register allocation [22], and function inlining [23]. Execution frequencies may be obtained from dynamic profile information or may be estimated using static techniques. Static estimates are derived by examining the statements and structure of the program. Examining the loop edges, conditional branch expressions, and the static call graph, gives information that can be used to predict the dynamic behavior of the program.

Static estimation is more convenient than dynamic profiling for a number of reasons. It does not require multiple compilation steps, selection of sample inputs, or feedback to the compiler after the profile run. Selecting input sets that generate representative profiles is a difficult task. Some input sets may only exercise a very small section of the program, completely ignoring other important sections. Since static analysis is independent of the input set, it allows 100% coverage of all branches. The primary disadvantage of static estimation is that the information obtained is usually less accurate than the information obtained from dynamic profiling. However, static estimation is faster than profiling since there is no need to profile the program and recompile it with the run-time information. In addition, profiling may not be possible in all environments such as real-time and embedded systems where gathering profile information is not usually feasible.

Many researchers have explored static branch prediction and static estimation of running time. Fisher and Freudenberger discovered that branches tend to go in one direction most of the time [24]. Ball and Larus have developed several heuristics to help make accurate branch predictions statically [25]. Wall in [26] also looked at static profiling. He constructed estimated profiles and compared them to real profiles. Unfortunately, he found that the estimated profiles were inferior to the real profiles. Ramamoorthy used Markov models with control flow information to estimate the running time of programs [27]. Wagner et al. combined the Markov models with accurate static branch prediction methods to obtain accurate inter- and intra-procedural frequencies and extended the techniques used by Wall [28].

One important element of static estimation is predicting branch direction probabilities. Ball and Larus discovered several branch prediction heuristics that use information contained in the branches and the control flow graph created by the branches. Below is a summary of the Ball and Larus branch prediction techniques [25]. The first heuristic applies to loop branches:

Loop Branch Heuristic. Predict all loop-back branches as taken. This heuristic is valid because loops tend to iterate multiple times and only exit once.

The next heuristics analyze the branch condition and successor blocks:

Pointer Heuristic. Predict that a comparison of a pointer to NULL or to another pointer will fail. Since most pointers are nonnull and are rarely equal to each other, branches of this nature are most likely going to be false.

Opcode Heuristic. Predict that a comparison of an integer to less than or less than or equal to zero will fail. Also, predict that a comparison of an integer equal to a constant will fail. This occurs because negative values tend to indicate error values and errors rarely occur. It will also predict that a comparison to check if two floating point numbers are equal will fail.

Guard Heuristic. Given a register that is used before being defined in a successor block, and that register is an operand in the comparison, and given that the successor block does not postdominate the register use, predict that control flow will reach the successor block. This heuristic attempts to find a branch on a value that guards a later use of that value. Most often the function of guards is to catch the exceptional use of the value which is the less frequently executed case.

Loop Exit Heuristic. Predict that a branch in a loop in which no successor is the loop header will not exit the loop. Since the loop only exits once, this branch will only be true on the last iteration of the loop.

The next heuristics analyze only the successor blocks:

Loop Header Heuristic. Predict a successor that is a loop header or a loop preheader and does not postdominate will be taken. This heuristic states that if a branch chooses between executing a loop or avoiding the loop, it predicts that loop will be taken.

Call Heuristic. Predict a successor that contains a call and does not postdominate will not be taken. This heuristic appears to be valid because most conditional calls are to handle exceptional situations.

Store Heuristic. Predict a successor that contains a store instruction and does not post-dominate will not be taken. Ball and Larus discovered this heuristic more by accident than anything else. It seems to work well for the floating point benchmarks.

Return Heuristic. Predict a successor that contains a return will not be taken. Recursion is the most likely justification for this heuristic. Like the exit branch in a loop, the return in a recursive procedure is the exception and does not occur frequently.

Table 2.2 Branch probability predicted by Ball and Larus heuristics.

Heuristics	Probability of taking branch
Loop Branch	88%
Pointer	60%
Call	78%
Opcode	84%
Loop Exit	80%
Return	72%
Store	55%
Loop Header	75%
Guard	62%

In many cases several of Ball and Larus's heuristics apply to an individual branch. Given a branch, their algorithm tries each heuristic in a predetermined order until it reaches the first one that applies. They performed a wide variety experiments to determine the best order in which to place the heuristics. In the case that no heuristic applies to the branch, a random prediction is used [25]. Table 2.2 lists Ball and Larus's hit rates. For example, when the Call Heuristic applies to a branch, they claim that the branch will fail 78% of the time. Wu and Larus took this one step further by developing an algorithm based on the Dempster-Shafer theory of evidence, to combine the probability estimates of all applicable heuristics into a stronger probability estimate. This turned out to be a vast improvement over the predetermined method that Ball and Larus used. A similar algorithm could be used to combine the branch probabilities from multiple runs of a dynamic profile [29].

Hank et al. showed in [30] that, by utilizing the Ball and Larus heuristics along with hazard analysis, superblocks could be formed without dynamic profile information. The statically formed superblocks outperformed basic block code and were competitive with the superblocks formed with profile information. Finally, Deitrich combined a variety of static analysis techniques in his thesis [31]. His goals were to enable ILP optimizations for unprofiled code and to ensure that ILP optimizations on profiled code do not degrade performance when the program's behavior for a real input differs from the behavior seen during profiling. It is apparent from these studies that current static estimation techniques cannot achieve the accuracy obtained with dynamic profiling. However, if code is being compiled in an environment where profiling is not available but optimization is desired, static estimation is a good alternative, and in light of this information the next logical step is the application of these techniques to hyperblock formation and predicated execution. With the ability of predication to remove hard-to-predict branches, the accuracy of these techniques may improve to the point that reliance on dynamic profiling could be significantly decreased.

2.3 The IMPACT Compiler

The IMPACT compiler framework provides a suitable platform to explore and utilize many ILP techniques including modulo scheduling, speculation and predication. A block diagram of the IMPACT compiler is presented in Figure 2.4. The compiler front end translates the C code into a high-level intermediate representation that is referred to as *Pcode*. In Pcode memory dependence analysis [32],[33], loop-level transformations [34] and memory system optimizations [35],[36] are performed. Additionally, profile-guided code layout and function inline expansion are performed at this level [37],[38],[39]. Once optimization at the Pcode level is complete, the resultant code is then translated to IMPACT's low-level intermediate representation, Lcode.

Lcode is the lowest level intermediate representation in the IMPACT compiler. Lcode resembles the register transfer language found in most load/store processor instruction sets. Lcode

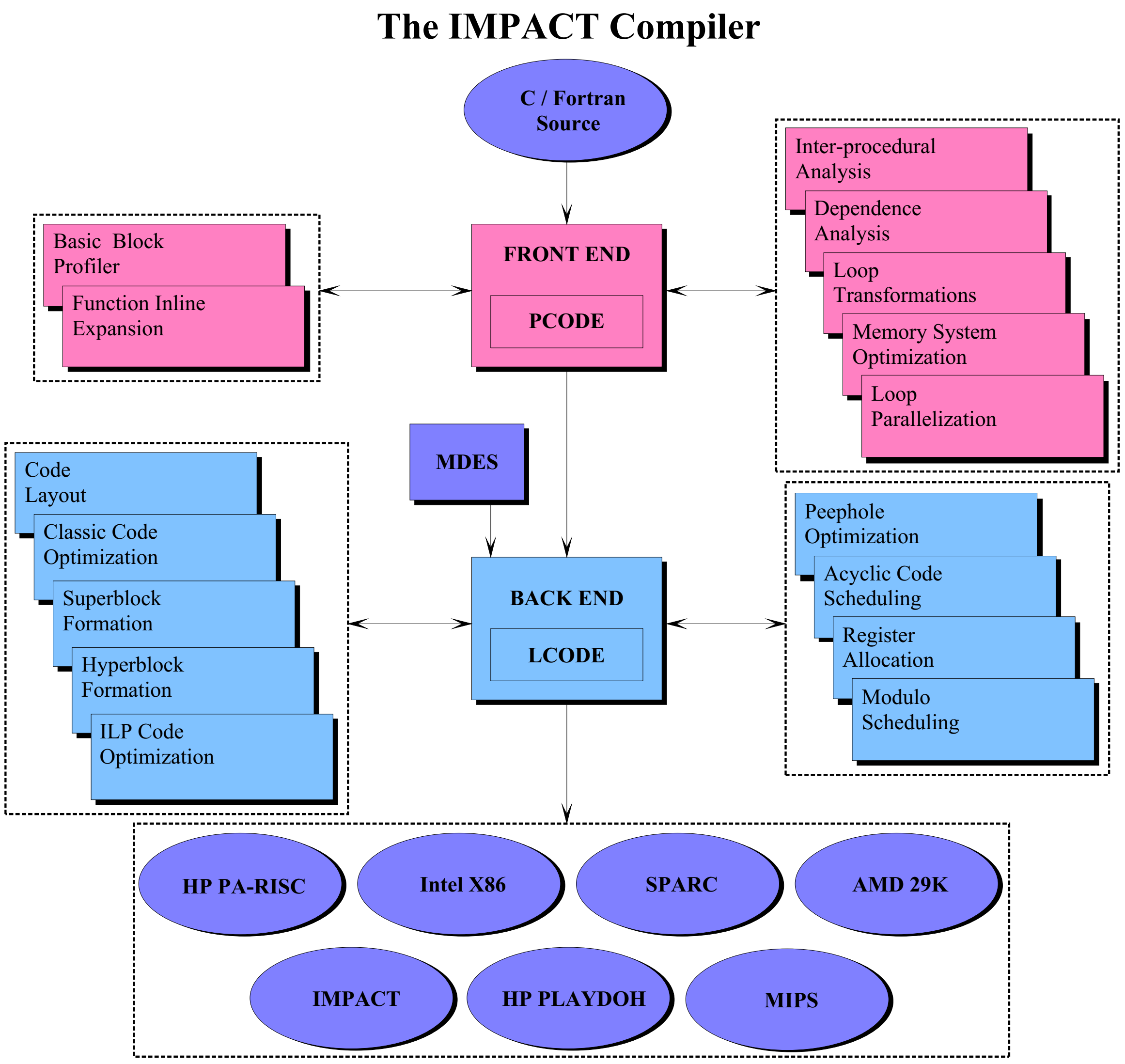


Figure 2.4 The IMPACT compiler.

is a generic intermediate representation not specific to any particular processor. For code generation to a particular architecture, Lcode is translated into *Mcode*. Mcode instructions map exactly to the target machine's assembly language instructions. To convert Lcode to Mcode the code generator breaks each Lcode instruction up into one more Mcode instructions that directly map to the target architecture. Mcode instructions are needed because of limited address modes, limited opcode availability, ability to specify literal operands, and the field width of literal operands within the target architecture.

Machine independent classic optimizations are applied at the Lcode level [40]. These optimizations 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. Identification of safe instructions for speculation and side-effect-free function calls is performed during interprocedural safety analysis [41] at this level.

Block formation techniques are also applied at the Lcode level. Superblock formation [20] is one type of block formation completed at this level. After superblock formation, superblock ILP optimizations are applied to take full advantage of the superblocks. For architectures supporting predication, hyperblocks [21] are formed along with superblocks. Advanced hyperblock formation is the focus of this thesis and is discussed in Chapters 3 and 4. All superblock ILP optimizations are performed on the hyperblocks along with an additional set of hyperblock specific optimizations designed to further exploit predicated execution.

IMPACT performs code generation at the Lcode level after block formation and optimization is complete. The two main components of code generation are the instruction scheduler and the register allocator. IMPACT can schedule code using either acyclic global scheduling [41],[42] or software pipelining using modulo scheduling [43]. Acyclic global scheduling involves two passes of the scheduler. Prepass scheduling is performed before register allocation, and postpass scheduling is performed after register allocation to generate the most efficient schedule. For modulo scheduling, loops targeted for software pipelining are identified. These loops are then scheduled using the modulo scheduler. The remaining code is scheduled with the acyclic global scheduler. Both techniques can take advantage of control and data speculation if the target architecture allows.

Register allocation is performed using the graph coloring method [44]. Profile information, if available, is used to assist the register allocator in making intelligent decisions. After register allocation a set of machine specific peephole optimizations is performed. These optimizations are designed to remove inefficiencies introduced during the Lcode to Mcode conversion and register allocation.

All Lcode modules take advantage of a detailed machine description database, *Mdes*, for the target architecture [45]. The database contains information such as number and type of functional units, size and width of register files, instruction latencies, instruction input/output constraints, addressing modes, and pipeline constraints. The Mdes is queried to assist with optimization, scheduling, register allocation and code generation.

The IMPACT compiler generates code for a variety of real and experimental architectures. The IMPACT and HP Labs PlayDoh architectures provide robust experimental frameworks for compiler and architecture research. The IMPACT architecture is a parameterized superscalar processor with an extended version of the HP PA-RISC instruction set. This architecture allows varying levels of support for speculative and predicated execution [46]. Because of its flexibility, all experiments in this thesis utilize the IMPACT architecture.

CHAPTER 3

HYPERBLOCK FORMATION

This chapter describes the hyperblock, a structure that allows the compiler to exploit predicated execution. Described are the compiler issues related to hyperblock formation. These issues include block selection, tail duplication, loop peeling, if-conversion and hyperblock optimization.

3.1 The Hyperblock

A *hyperblock* is a set of predicated basic blocks in which control can only enter from the top of the region, but may exit from one or more locations [21]. A single basic block in the hyperblock is designated as the entry block, and control flow may only enter the hyperblock at this point. The motivation behind forming hyperblocks is that many basic blocks from different control flow paths can be grouped into a single manageable block for compiler optimization and scheduling. However, not all basic blocks through which control may flow are included in the hyperblock. Blocks may be excluded from the hyperblock to allow for more effective optimization and scheduling.

The superblock is the predecessor of the hyperblock. Like the hyperblock, a superblock is a set of basic blocks in which control enters at the top of the region and may exit from one more locations. However, a superblock contains no predicated instructions, and consequently a superblock only encompasses the instructions from one path of control. Conversely, hyperblocks contain instructions from multiple paths of control. Hyperblocks provide a more robust framework for the compiler to optimize regions in the presence of non-heavily biased branches.

3.2 Hyperblock Formation

Forming hyperblocks is a four step-process. First, the blocks that will compose the hyperblock must be selected. Next, tail duplication is performed to remove side entrances from the group of selected blocks. Then, loop peeling is performed to remove any internal cycles from the selected blocks. Finally, the region of selected blocks is if-converted to form the final hyperblock. The following sections describe each of these steps in detail.

3.2.1 Block selection

The first step in forming a hyperblock is selecting which basic blocks from within a region to include in the hyperblock. Blocks are selected from regions that include loops, nested loops, and acyclic conditionals, called hammocks. The compiler attempts to identify the largest regions possible under two constraints. First, each basic block may only reside in one region. Second, the region may not contain any internal cycles. This constraint can be relaxed with the support of loop peeling which will be discussed later in this chapter. It is important to note that not all blocks in the region will be selected for inclusion in the hyperblock for two reasons. First, the region may become too large for effective compiler optimization and scheduling. Second, combining all the paths of execution could result in a loss of performance due to over-subscription of limited fetch or execution resources. Detailed descriptions of two block selection

```
tail_duplication($set\_of\_bb$)
  Let $set\_of\_bb$ be the blocks that are selected for hyperblock formation
    where $bb_1$ is the entry block of the hyperblock.
  $done$ = false
  while (not $done$)
    $done$ = true
    for each basic block in $set\_of\_bb$, $bb_i$
      if ($bb_i \neq bb_1$)
        mark each $bb_i$ that has predecessor not in $set\_of\_bb$

    for each basic block in $set\_of\_bb$, $bb_i$
      if ($bb_i$ is marked)
        if $bb_i$ is not duplicated yet
          duplicate $bb_i$
        else
          use previously duplicated $bb_i$
        change all incoming flow arcs of basic blocks not in $set\_of\_bb$ from
          pointing to $bb_i$ to duplicate $bb_i$.
        $done$ = false
```

Figure 3.1 Algorithm for hyperblock tail duplication.

algorithms are given in Chapter 4. It should be noted that the method used to select the blocks in no way changes how any of the remaining steps are performed.

3.2.2 Tail duplication

Next, the hyperblock must be transformed into a single entry region. To do this, 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 entrances.* Side entrances are removed from the hyperblock using a technique called tail duplication. An algorithm to perform tail duplication is shown in Figure 3.1. The tail duplication algorithm marks all of the blocks with side entrances. Then, all selected blocks which may be reached from a marked block are also marked. These marked blocks are duplicated, and the control arcs corresponding to the side entrances are

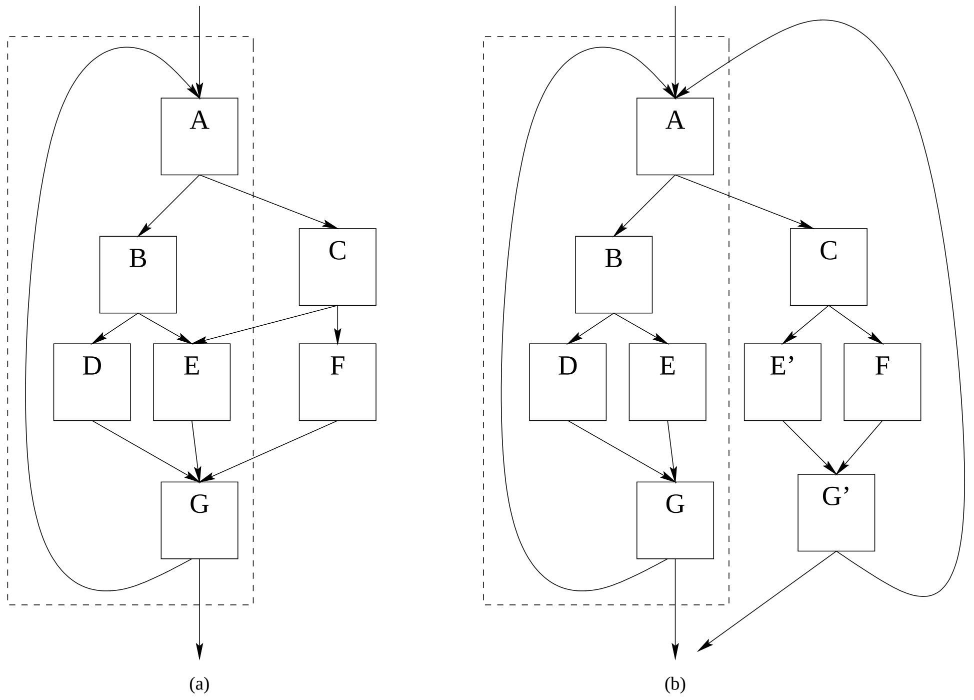


Figure 3.2 An example of tail duplication: (a) after block selection, (b) after tail duplication.

redirected to the duplicated blocks. To minimize the amount of code expansion created by this technique, blocks are duplicated at most once.

An example of tail duplication is shown in Figure 3.2. As shown in Figure 3.2(a) blocks E and F both contain a side entrance, because blocks C and F were not selected for inclusion in the hyperblock which is indicated by the dashed box. Therefore, both blocks E and G will be duplicated and the control flow from blocks C and F will be redirected to duplicated blocks E' and G'. Also, note that the control flow from block E' is directed to G'. If this were not done, an additional side entrance would have been created. The result of this tail duplication is shown in Figure 3.2(b). At the completion of tail duplication, all outside entrances to the region will come only through the entry block.

```
loop_peeling($set\_of\_bb$)
    Let $set\_of\_bb$ be the blocks that are selected for hyperblock formation.
    and $bb_1$ is the entry of the hyperblock.
    for each inner loop in $set\_of\_bb$, $loop_i$
        $k$ = expected number of times $loop_i$ is executed each time through the loop
        if ($k >$ MAX_NUM_PEELS) $k$ = MAX_NUM_PEELS
        duplicate $loop_i$ $k$ times
        change the loop back arc from last duplicated $loop_i$ to the original header block of $loop_i$
        add the duplicated $loop_i$ to $set\_of\_bb$
```

Figure 3.3 Algorithm for hyperblock loop peeling.

3.2.3 Loop peeling

Another condition that also must be satisfied before the blocks are if-converted is that there exist no nested inner loops inside the selected blocks. Normally, this is not an issue, since the compiler generally wants to optimize and schedule loops as a single entity. Because of transformations such as loop unrolling and register renaming, large amounts of ILP can be extracted from hyperblock loops. However, loops that execute infrequently are one important exception. These loops do not contain enough iterations to achieve the desired level of ILP using the unrolling techniques.

Loop peeling is an optimization that peels off iterations of an inner loop that are nested within the selected blocks [47]. An algorithm for performing loop peeling is given in Figure 3.3. These iterations can then be overlapped with the instructions from the surrounding blocks to increase ILP. The loop is usually peeled enough times so that the majority of the invocations just execute the peeled code. The peeled iterations will appear as acyclic code in the hyperblock. The predicates in the hyperblock are set to allow execution of just enough iterations. Any peeled iterations not required for execution will be nullified by their predicates. A copy of the original loop body, also called the recovery loop, is maintained to handle invocations of the peeled loop

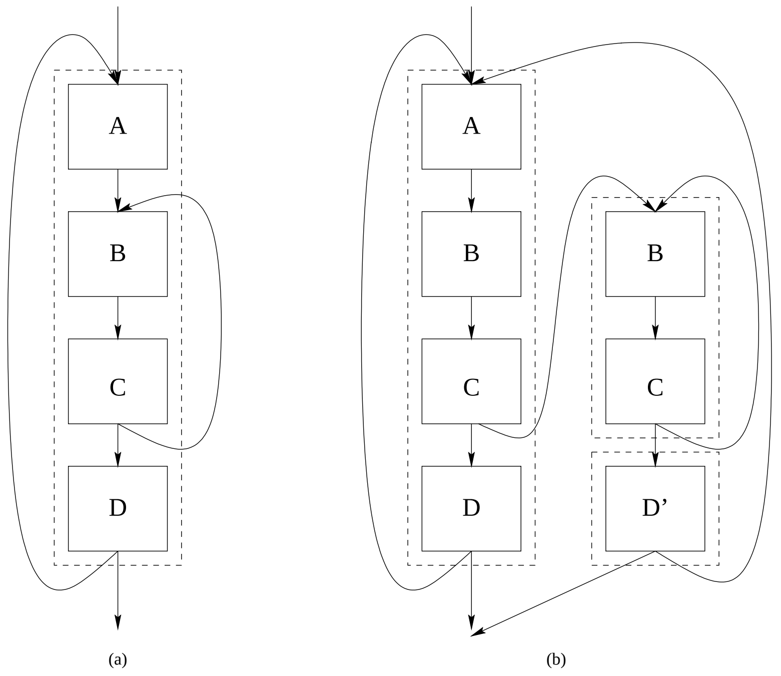


Figure 3.4 An example of loop peeling: (a) original flow graph, (b) after peeling one iteration of the inner loop and tail duplication.

that exceed the number of peeled iterations. A branch is placed after the last peeled iteration to test if more iterations are required. If these additional iterations are required, control flow will be transferred out of the hyperblock to the recovery loop to ensure program correctness.

In many cases, the code surrounding the inner loop is another larger outer loop. By peeling the inner loop and thus converting it into acyclic code, a hyperblock can be formed around the outer loop. The hyperblock then consists of the outer loop and the peeled iterations of the inner loop. The inner loop iterations can be transformed to enable high levels of ILP to be extracted from the outer loop.

An example of loop peeling is shown in Figure 3.4. In this example, all the blocks have been selected for inclusion in the hyperblock including the inner loop consisting of blocks B and C. This is shown in Figure 3.4(a). Before these selected blocks can be transformed into a

```
if_conversion($set\_of\_bb$)
    Compute_dominator($set\_of\_bb$)
    Compute_post_dominator($set\_of\_bb$)
    Compute_control_dependences($set\_of\_bb$)
    for each control depedence set, $cd_i$
        Assign control dependence $cd_i$ to predicate register $R_i$
        Add predicate define operations in basic blocks containing branches to be removed
    for each basic block, $bb_i$
        use the predicate register assigned in R
        Remove all forward branches
        predicate this basic block with the assigned predicate register
```

Figure 3.5 Algorithm for if-conversion.

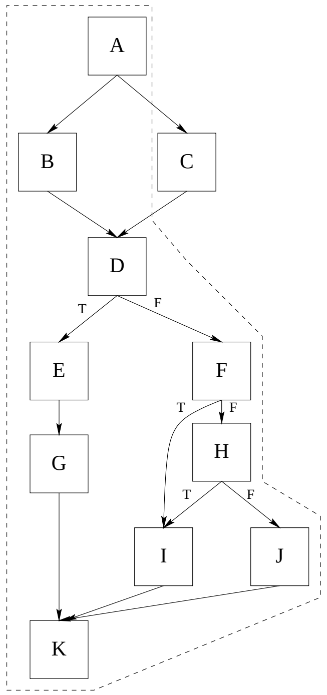
hyperblock, the inner loop backedge must be removed. For this simple example, the inner loop is expected on average to iterate once. Therefore, one iteration of the inner loop is peeled. The original loop code is placed outside the hyperblock to form the recovery loop, and a branch is placed at the end of the peeled iteration to this recovery loop. Note that after peeling, tail duplication is required for block D to prevent side entrances into the hyperblock. The final peeled and tail duplicated code is shown in Figure 3.4(b).

3.2.4 If-conversion

Finally, if-conversion is used to remove the conditional branches from the hyperblock, thereby converting control dependences to data dependences. Not all conditional branches will be eliminated; those branches that transfer control outside the hyperblock will remain after if-conversion. There are several steps in the if-conversion process. First, the dominator and postdominator information must be computed for each basic block that will form the hyperblock. Next, the control dependence information is calculated among the selected basic blocks using the previously computed dominator and postdominator information. The control dependence information is maintained as a set of edges in the control flow graph which determine the execution condition of a particular basic block. The algorithm for performing if-conversion is shown in Figure 3.5.

Each unique control dependence set is assigned a predicate register. All blocks that share a common control dependence set will be executed under control of the same predicate. Predicate defining instructions are inserted into all basic blocks which are the source the of the control dependence edges associated with a particular predicate. The predicate define condition is determined by the branch condition specified by a particular control dependence edge. After the predicate define instructions are inserted, the instructions in the selected blocks are conditioned with the predicate corresponding to the block's control dependence set. Finally, the code is placed linearly in the hyperblock using a topological sort of the selected block's control flow graph. Note that it is possible for a region of blocks to have only one control dependence on the entry block. This implies that the region encompasses only one path of control, and for this case predication is not required and the region will be transformed into a superblock rather than a hyperblock.

Figure 3.6 shows how localized control dependence is calculated for a control flow graph. The dashed line indicates the blocks selected for inclusion in the hyperblock. Block C has been excluded from the hyperblock. Consequently, blocks D though K will be tail duplicated and the control flow from block C will be redirected to the tail duplicated blocks. Blocks A, B, D, and K have no local control dependences. Therefore, these blocks will always be executed if the hyperblock is executed. The instructions in these blocks do not require predicates and are said to execute on the true predicate. The remaining blocks are control dependent on the edges specified in the figure. Control dependences are denoted by indicating the branch from



Control Dependence	Predicate Assignment
A : none	A : True
B : none	B : True
D : none	D : True
E : brD	E : p1 (U)
F : $\overline{\text{brD}}$	F : p2 (U)
G : brD	G : p1 (U)
H : $\overline{\text{brF}}$	H : p3 (U)
I : brF, brH	I : p4 (OR)
j : $\overline{\text{brH}}$	J : p5 (U)
K : none	K : True

Figure 3.6 Localized control dependence calculation and predicate assignment.

which they originate. The edges on the graph are marked T for true and F for false conditions. For example, the control dependence for block F is $\overline{brD}$, indicating that the branch condition in block D must be false for block F to be executed. In this example, the hyperblock contains five unique control dependence sets, thus five predicates are required. The mapping of control dependences to predicates and the assignment of predicates to basic blocks is also shown in Figure 3.6.

Unconditional predicates (indicated by a U in the figure) are used for predicates that have a single edge in their control dependence set. Predicates p1, p2, p3, and p5 are unconditional type predicates. OR-type predicates (indicated by a OR in the figure) are used for predicates that have multiple edges in their control dependence sets. These predicates will be set to 1 if any of the edges are traversed. In this example, p4 is an OR-type predicate because of the two edges leading to block I. OR-type predicates must be cleared using an explicit predicate clear instruction before they can be set. On the other hand, unconditional predicates do not require these explicit clears.

3.3 Hyperblock Specific Optimizations

The formation of hyperblocks creates many opportunities for new compiler optimizations to increase the performance of predicated code. These optimizations not only improve the performance of the predicated code; they also perform transformations only made possible with predicated execution. This section will provide a brief description of three hyperblock optimization techniques: predicate promotion, branch combining, and instruction merging. For more detailed information on these techniques see [19].

3.3.1 Predicate promotion

Speculation is an important technique used to achieve high amounts of ILP in superscalar and VLIW processors. Speculation refers to executing instructions before their conditions for execution are completely known. Speculation comes in two forms. In the first form, instructions can be moved above the exit branches in the hyperblock. This form occurs in superblocks formed for unpredicated code as well. The second form occurs only in the predicate domain and is know as predicate promotion. By performing either type of speculation, the scheduler is given substantially more freedom, allowing it to achieve a more compact schedule.

Predicate promotion advances the predicate of an instruction to an ancestor predicate [21]. This is beneficial since the ancestor predicate is computed using fewer conditions than the original predicate. As a result, the promoted instruction is executed under fewer conditions

```
instruction_promotion()
  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. there is a unique $op(y), y < x$, such that
           $dest(y) = pred(x)$.
      4. $dest(x)$ is not live at $op(y)$.
      5. $dest(j) \neq dest(x)$ in $\{op(j), j = y + 1 \ldots x - 1\}$.
    then do:
      set $pred(x) = pred(y)$.
```

Figure 3.7 Algorithm for instruction promotion.

than the original program specified, making it a speculative instruction. An algorithm for instruction promotion is given in Figure 3.7. Typically, in hyperblocks, the critical dependence height comes from the instructions awaiting the computation of their predicates. Predicate promotion's major advantage is reducing the dependence height of the hyperblock. Predicate promotion allows the dependences between the predicate define instructions and the predicated instructions to be broken and can thereby reduce the dependence height of the hyperblock. The dependences can be completely broken if an instruction's predicate is advanced to the true predicate. The overall result of predicate promotion is that more compact schedules can be achieved through reduced dependence height and additional code motion freedom.

3.3.2 Branch combining

One problem with hyperblocks is the inclusion of infrequently taken exit branches. In a machine with limited branch resources, these branches can quickly become a performance bottleneck. These branches transfer control to blocks not selected for inclusion into the hyperblock. Typically, these blocks handle special cases, boundary conditions, and exceptions. If a hyperblock contains a large number of these branches, all the performance gained by the hyperblock formation can be lost.

To combat this problem, the compiler can perform a technique called branch combining. Branch combining replaces a group of exit branches by a corresponding group of predicate defines. All of these predicate defines are of the OR-type and write into the same destination predicate. Since multiple OR-type predicate defines can be issued per cycle, the processor is no longer limited by the single branch resource. The resultant predicate will be set to true if any of the exit branches were to be taken. Following the predicate defines will be a jump instruction controlled by the resultant predicate that transfers control to a block outside the hyperblock that will decode the exit condition. In this block, the branches are re-executed in original program order to determine which branch was originally taken. It is possible for multiple exit branches to be true, and in this case the first such branch will be taken since that branch would have been taken in the original code sequence.

3.3.3 Instruction merging

Unlike the previously described techniques, instruction merging is an optimization geared primarily for improving the efficiency of predicate code rather than increasing 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. This optimization actually results in the reduction of the total number of instructions in the hyperblock. Instruction merging can be effectively used to reduce the number of instructions for resource limited instruction classes such as branches and stores. Merging these instructions allows for more compact schedules to be achieved by reducing resource pressure.

```
instruction_merging()
  for each instruction, $op(x)$, in the hyperblock
    if all the following conditions are true:
      1. $op(x)$ can be promoted.
      2. $op(y)$ can be promoted.
      3. $op(x)$ is an identical instruction to $op(y)$.
      4. $pred(x)$ is the complement form of $pred(y)$.
      5. the same definitions of $src(x)$ reach $op(x)$ and $op(y)$
      6. $op(x)$ is placed before $op(y)$.
    then do:
      promote $op(x)$.
      delete $op(y)$.
```

Figure 3.8 Algorithm for instruction merging.

For instruction merging to occur, instructions with the same source operands, destination operands, and equivalent opcodes are identified. Next, the compiler determines if the same values for each of the operands reach both instructions. If so, these instructions are candidates for merging. Then, the compiler ascertains whether one of the instructions can be promoted to a new predicate which allows the other instruction to be eliminated. This new predicate represents the logical OR of the candidate instruction predicates. This algorithm is given in Figure 3.8. Currently, only two scenarios are checked for merging. In the first case, the candidate predicates are mutually exclusive with a common ancestor predicate. In this case, the ancestor predicate is used as the new predicate. The second case occurs when one of the candidate predicates is an ancestor of the other. For this case, the ancestor predicate itself serves as the new predicate.

CHAPTER 4

BLOCK SELECTION

This chapter describes the most important issue related to hyperblock compilation and the main contribution of this thesis, block selection. First, the path enumeration method of block selection is described as it is the current method used within the IMPACT compiler. Next, block enumeration is presented as another method of block selection that overcomes some of the problems with path enumeration.

4.1 Path Enumeration Based Block Selection

One method by which blocks are selected for inclusion in a hyperblock is to enumerate all execution paths through the region. This is called the *path enumeration* method of block selection. An execution path is a path of control flow from the entry block to the exit block in the region. A priority is calculated for each path in the region to determine its relative importance. Paths are included from highest priority (the main path) to lowest priority based on estimated available execution resources and characteristics of the path. The final set of blocks that end up forming the hyperblock is the union of all the blocks along the included paths of execution. The algorithm for performing path enumeration is given in Figure 4.1.

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

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

path_enumeration_based_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 4.1 Algorithm for path based block selection.

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 heights 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:

$$dep\_ratio_i = 1.0 - (dep\_height_i / \max_{1 \le j \le N}(dep\_height_j)) \tag{4.1}$$

$$op\_ratio_i = 1.0 - (num\_ops_i / \max_{1 \le j \le N}(num\_ops_j)) \tag{4.2}$$

$$priority_i = (probability_i \times hazard_i) \times (dep\_ratio_i + op\_ratio_i + K) \tag{4.3}$$

Equation (4.1) 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 (4.2). 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 (4.3). The priority is the product of two terms. The first term is the probability of path traversal 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 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.

Priorities for all the paths are determined and the paths are sorted and stored in priority order. Then, they are considered for inclusion in the hyperblock from highest to lowest priority. Paths are included into the hyperblock provided they meet the following three conditions. First, additional resources required by the path do not cause the hyperblock to exceed the estimated number of available resources. The total number of resources available to the hyperblock is estimated by multiplying the dependence height of the main path by the issue width of the processor. Then, the number of instructions in the main path is subtracted from this total, leaving the number of resources available for use by other execution paths. Second, the dependence height of the path may not exceed the dependence height of the main path. Finally, the priority of the path must be within some fraction of the priority of the last included path. This prevents low priority paths from being included into a region consisting of mostly high priority paths.

The aggressiveness of the path enumeration algorithm can be adjusted by changing two key parameters. The first parameter determines the minimum profile weight for a block to be included in the hyperblock. The second parameter specifies the minimum path execution ratio. This ratio is calculated by dividing the profile weight of the path being considered for inclusion by the profile weight of the main path. Reducing both of these parameters causes

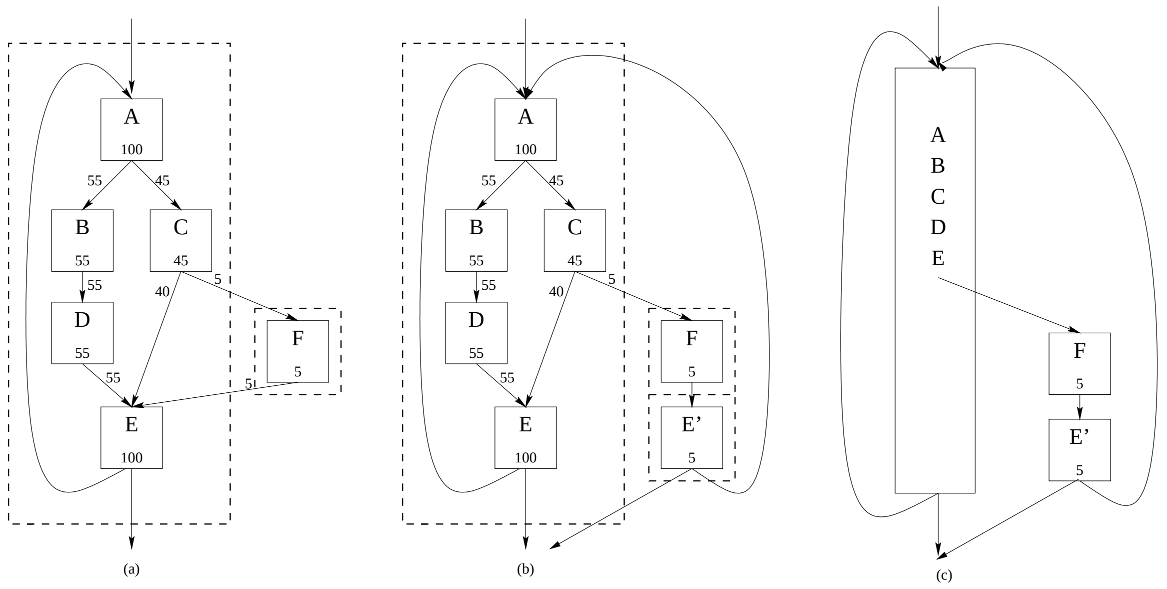


Figure 4.2 An example of path enumeration hyperblock formation: (a) after block selection, (b) after tail duplication, (c) after if-conversion

the algorithm to include lower weight, and therefore lower priority paths into hyperblocks. The most aggressive path enumeration is obtained by setting these parameters both to zero. This forces the algorithm to form hyperblocks in low profile weight regions. The effects of this more aggressive path enumeration are explored in Chapter 5.

To illustrate the path enumeration selection heuristic, an example is presented in Figure 4.2. This example shows the weighted control flow graph for a program loop segment. The numbers associated with each node and arc represent the dynamic execution frequency of the block and the branch direction, respectively. The main path in this example consists of blocks A, B, D, and E. There are two other possible paths through this region. The first path consists of blocks A, C, and E. The second path consists of blocks A, C, F, and E. The first path will be selected for inclusion in the hyperblock because it is executed relatively frequently when compared to the main path. The second path is rejected from inclusion in the hyperblock because it is executed infrequently. Therefore, the selected blocks that will comprise the hyperblock consist of blocks A, B, C, D, and E. This is shown by the dashed box around the blocks in Figure 4.2(a). Figures 4.2(b) and (c) show the hyperblock after tail duplication and if-conversion, respectively.

4.2 Block Enumeration Based Block Selection

A second method of selecting blocks for inclusion into a hyperblock is called *block enumeration*. This method is the main contribution of this thesis, and comparisons of this method to the previously described path enumeration method will be made in Chapter 5. Two potential problems can arise when using the path enumeration method for selecting basic blocks. First, if a large, complex region is passed to the block selector, a very large number of paths may exist in the region. Enumerating this large quantity of paths may lead to unacceptable compilation time. The second potential problem occurs when low priority paths are excluded from the hyperblock. Often these paths may only include one or two small zero or low profile weight basic blocks, and excluding these blocks can actually lead to several problems. Usually these blocks re-enter the selected region, and this leads tail duplication. In some cases, the amount of tail duplication and consequent code expansion can be quite large. Also, excluding these small zero weight blocks requires that a branch be inserted into the hyperblock, which could possibly increase the schedule or dependence height for the hyperblock.

Figure 4.3 shows an example that illustrates both of the previously discussed issues. This example is the *adpcm.decode* function in the *adpcm* benchmark. Figure 4.3(a) shows the original basic block layout of the function. Figures 4.3(b) and (c)shows the function after path enumeration hyperblock formation and after block enumeration hyperblock formation, respectively. The control flow is clearly more complex for the path enumerated hyperblock. It

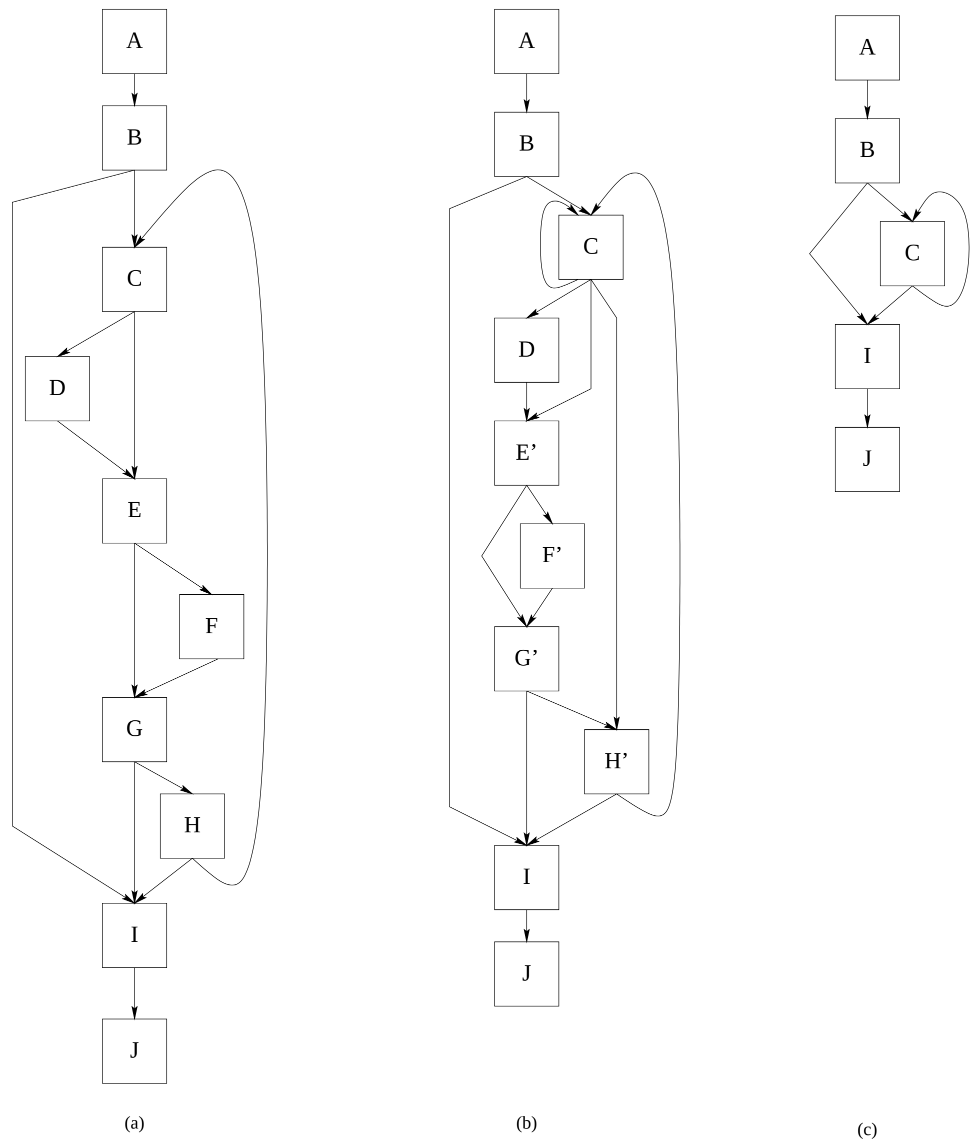


Figure 4.3 adpcm.decode function with (a) original basic block code, (b) path enumerated hyperblock formation, and (c) block enumeration hyperblock formation.

also requires more cycles to execute 3,099,400 versus 2,656,840 and consists of more instructions 81 versus 68. This inferior path enumerated hyperblock is created because a zero weight path is excluded from the hyperblock. Block D is never executed when the benchmark is profiled. Consequently, when a path runs through this block, it is given very low priority and is excluded from the hyperblock formed in block C. On the surface this might seem like a good idea as there is no point in wasting scarce functional units or predicate registers on instructions that, according to the profile information, will never execute. However, block D only contains two operations, and they will not utilize significant resources if included in the hyperblock. Excluding this block also causes significant tail duplication because it re-enters the resultant hyperblock. In Figure 4.3(b) blocks E', F', G' and H' all represent the necessary tail duplication. However, this does not explain the disparity between the number of cycles to execute the function. In the path enumerated hyperblock, blocks D, E', F', and G' are never executed and do not contribute to the number of cycles required for execution. What does cause the increased cycle count is increased schedule height for the branches to blocks D, E' and H'. The modeled processor only has one branch functional unit, and Figure 4.3(b) clearly shows that at least three cycles will be required just for branches to execute in block C. Since this loop is executed 147,520 times, one extra level of schedule height quickly adds up. By including block D in the hyperblock as shown in Figure 4.3(c), the excessive tail duplication is avoided, and predicating the instructions avoids the bottleneck for the single branch unit. This example clearly illustrates that in some cases the path enumeration method does not always form ideal hyperblocks. This coupled with the sometimes excessive compile time required to enumerate the paths indicates that a more advanced block selector needs to be developed.

The block enumeration method of block selection is composed of a variety of techniques. The overall algorithm is given in Figure 4.4. First, the main execution path through the region is found. This is accomplished by starting at the entry block to the region and following the control flows of highest weight to the exit block of the region. This is very similar to the algorithm used to form traces during superblock formation. The algorithm for finding the main path is given in Figure 4.5. Hazards such as register jumps and unsafe stores are avoided in

```
/* Predefined variables for block selection */
ISSUE_WIDTH = 1 to MAX_WIDTH  /* As specified in the machine description file */
RES_MULTIPLIER = 2
MAX_DEP_GROWTH = 3
MAX_TAIL_DUP_GROWTH = 1.3
MAX__STATIC_TAIL_DUP_GROWTH = 1.05
block_enumeration_block_selection(region)
    if ($region\_weight > min\_region\_weight$)
        find the main path in the region
    else
        find the static path in the region
    /* For the rare case that a main path can not be found */
    if (no main path found)
        return NULL
    $avail\_resources = ISSUE\_WIDTH \times dep\_height\_main\_path \times RES\_MULTIPLIER$
    while ($blocks\_to\_be\_examined$)
        form a block path with unselected blocks
        make a region consisting of the new path and previously selected blocks
        /* Check if there enough resources available to include the path */
        if ($num\_ops\_new\_region > avail\_resources$)
            continue
        /* Prevent paths with large relative dependence heights from being included */
        if ($dep\_height\_new\_region > (dep\_height\_selected\_region \times MAX\_DEP\_GROWTH)$)
            continue
        /* Do not include paths that increase tail duplication */
        if ($((code\_size\_of\_{n}ew\_region\_with\_tail\_dup \div code\_size\_of\_new\_region)$
            $< MAX\_TAIL\_DUP\_GROWTH)$
            continue
        /* Include the path in the hyperblock */
        $selected\_region = new\_region$
    $sel\_blocks$ = all blocks contained within $selected\_region$
    return $sel\_blocks$
```

Figure 4.4 Algorithm for block enumeration based block selection.

the process. If a hazard free main path can not be found through the region, a hyperblock will not be formed. This is a very rare case, and has rarely if ever been observed in the benchmark studies. After the selection of the main path, statistics consisting of dependence height and resources usage are calculated for this main path. Then, blocks that were not selected are considered for inclusion one *block path* at a time. A block path represents a block or group of blocks that is entered from a block on the main path and exits back into another block on the main path. After the selection of a block path, it is combined with the main path and

```
select_main_path(region)
    $current\_cb = entry\_cb$
    while ($current\_cb$ != $exit\_cb$)
        $sel\_blocks = current\_cb \cup sel\_blocks$
        foreach $successor$ of $current_c b$
            if ($successor$ contains a hazard)
                continue
            if (($successor\_weight > selected\_successor\_weight$)$\vee$ (no successor selected))
                /* Include the block in the main path */
                $selected\_successor = successor$
        $current\_cb = selected\_successor$
    return $sel\_blocks$
```

Figure 4.5 Algorithm for selecting the main path.

again dependence height and resource usage are calculated. If no hazards are introduced, issue slots are available, dependence height is not increased, and tail duplication is not increased by more than a specified factor, then the blocks comprising the block path in conjuction with the main path blocks will be included in the hyperblock. The algorithm includes a parameter that specifies the amount of tail duplication allowed for each region. Code expansion is a common concern, so this parameter is set not to allow the region to expand more than 30%. This process continues until either all the blocks in the region are included in the hyperblock, or until all the block paths have been examined. Note that this algorithm does not enumerate every execution path through the hyperblock. Instead, it enumerates each block that adds new possible paths of execution through the hyperblock. The block enumeration algorithm attempts to form optimum hyperblocks by examining the structure and layout of the region, rather than examing all possible execution paths.

An example of the main path selection algorithm and the block enumeration based selector is presented in Figure 4.6. The first step in processing the region shown in Figure 4.6(a) is to select the main path. The entry block A is added as the first block in the main path. The

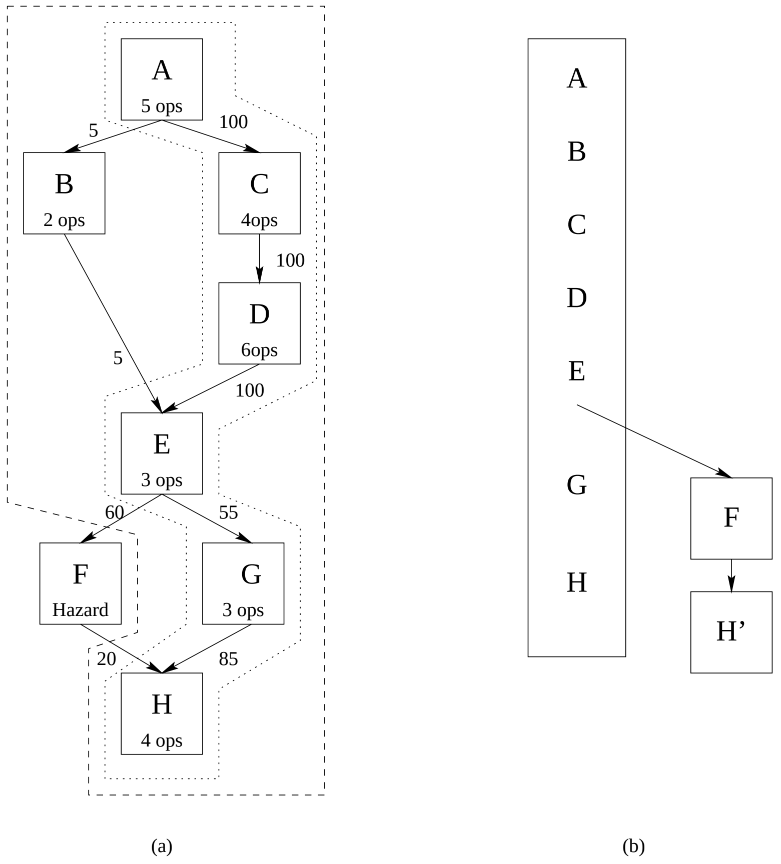


Figure 4.6 Block enumeration selection example: (a) after block selection, (b) after tail duplication and if-conversion.

successors of block A are considered for inclusion, and block C is chosen because of the higher flow weight leading to it. The flow weights are indicated by the numbers next to the control flow edges. Block D is selected because it is the only sucessor to block C, and block E likewise. Next, blocks F and G are considered for inclusion. Block F is rejected even though it has the higher flow weight because it contains a hazard; therefore block G is choosen for inclusion. Finally, block H is selected because it is the only successor to block G and it is the exit block. The blocks selected for the main path are surrounded by the lightly dotted line. Statistics will be calculated for the main path including the number of issue slots required, dependence height, and the amount of tail duplication. Next, the algorithm will try and expand the hyperblock by including blocks that were not selected in the main path. Starting with block A, each block in the main path will be examined for successors not included in the main path. The algorithm determines that block B was not included in the original main path and itself forms a complete block path. Using block B in conjuction with the main path, the compiler computes the region statistics again. These statistics are then compared to the previously computed statistics of the main path. If the added block does not cause the region to exceed the available slots, increase dependence height, and increase the amount of tail duplication, then it will be added to the hyperblock. In this case, the main path has plenty of slots available and adding block B will actually reduce the amount of tail duplication. Therefore, it is added to the region even though it has a low profile weight and probably would have been excluded from the region if the path enumeration method had been used. This process continues until all the main path blocks have been examined for successors. Block F will again be examined for inclusion and will again be rejected because of the hazard it contains. All blocks selected for inclusion in the hyperblock are surrounded by the heavier dashed line. Figure 4.6(b) shows the completed hyperblock after tail duplication and if-conversion.

This algorithm also attempts to form hyperblocks in regions with very low or no execution weight. The path enumeration algorithm could do this, but without execution weights, a new method for calculating the path priorities would have to be found. However, the block enumeration algorithm works with only a minor change to how the main path is found. Since execution weights are not available or are very low, the method of following the highest flow weight is not possible and could be unreliable. Therefore, the main path must be determined with a method that is independent of profile information. A method based on the static branch prediction techniques discussed in Chapter 2 is used. Starting with the entrance block of the region, branch destinations are inspected to determine if they meet the following criteria: they do not contain to subroutine call, they do not contain an unsafe pointer store, and they do not contain a register jump. If the branch destination meets these criteria, the destination block is added to the main path. If two successors of one block both meet this criteria, the larger of the two successors is added to the main path. Chances are the other block will be added later during the block enumeration phase. The larger of the blocks is chosen because it is likely to have the largest dependence height and will consume the most resources. Again, if a hazard free path can not be found through the region, the hyperblock formation attempt will be aborted. The algorithm for the static selection method is given in Figure 4.7. After finding the staticaly selected main path, block enumeration progesses just as before. Block paths are considered and included as long as they do not reduce the performance of the already included blocks. As before the amount of tail duplication allowed is adjustable based on the setting of a parameter. The goal in forming low weight hyperblocks is to reduce code size, while providing an optimzation opportunity with predication. Therefore, a minimal amount of tail duplication is allowed for the low weight regions. Experimental evaluation indicates that allowing the code of a region to increase in size no more than 5% allows an adequate number of low weight hyperblocks to be formed.

Figure 4.6 will again be used to illustrate the operation of the static selection algorithm. However, this time the control flow edge weights will be ignored since, for a region to be passed to the static selector, it would have very low or no profile weights. This example will be looked at with two different configurations for block F. First, assume block F is as shown in the figure with the indicated hazard. Block selection would progress in much the same way as the previous example. The algorithm attempts to select the main path statically. Starting with block A's

```
static_based_main_path_selection(region)
    $current\_cb = entry\_cb$
    while ($current\_cb$ != $exit\_cb$)
        $sel\_blocks = current\_cb \cup sel\_blocks$
        foreach $successor$ of $current_cb$
            if ($successor$ contains a jsr)
                continue
            if ($successor$ contains a register jump)
                continue
            if ($successor$ contains an unsafe pointer store)
                continue
            if (($successor\_weight > selected\_successor\_weight$)$\vee$ (no successor selected))
                /* Include the block in the main path */
                $selected\_successor = successor$
        $current\_cb = selected\_successor$
    return $sel\_blocks$
```

Figure 4.7 Algorithm for statically selecting the main path.

successors, each is checked for hazards, of which there are none. Therefore, the block with more instructions will be choosen: block C. This progresses until the exit block is reached, rejecting block F along the way for the hazard. Then as before, the selected path's various statistics are computed for later comparison to newly selected paths. The statically selected main path is comprised of blocks A, C, D, E, G and H. Each main path block is then checked for successors to determine if the hyperblock can be expanded. Block B will be included in the hyperblock since adquate resources are available. Block F will be rejected because of the hazard. The statically selected region matches the region that was selected using the profile weights and the same hyperblock would be formed. However, forming this region causes code expansion because block H must be tail duplicated. If the amount code expansion created by duplicating block H was more than 5% of the region size the region would be rejected and a hyperblock would not be formed. Conversely, if block H was very small the hyperblock would be allowed to form, since the amount of tail duplication would be minimal. Looking at this example again, assume that block F contains two hazard-free instructions. Main path formation would proceed the same as before. Then as block enumeration progresses, block F would be selected this time, thus forming a hyperblock of the entire region with no tail duplication required.

The motivation for forming these low weight hyperblocks is twofold. First, forming these regions can reduce code size. In the above example, at least two instructions and possibly more are eliminated. Performing this technique over a number of regions in each function has the potiental of significant code size savings over an entire program. Second, dynamic profiling does not execute all portions of the program, and in fact, can completely exclude significant areas. Performing this techinique allows the compiler an attempt to optimize these areas that might otherwise be ignored, and thus a possible performance increase will be realized when these areas of the program are entered in the real world.

CHAPTER 5

EXPERIMENTAL EVALUATION

The effectiveness of hyperblock compilation techniques is presented in this chapter. The methodology used to conduct the experiments is first described. The results are then presented and include the performance and code size issues of hyperblocks.

5.1 Methodology

The execution time for each benchmark is derived from the static code schedule weighted by dynamic execution frequencies obtained from profiling. Previous experience with this method of execution time estimation has demonstrated that it accurately estimates simulations of the modeled machine with perfect caches.

5.1.1 Processor model

The processor modeled in this study is an in-order issue superscalar with register interlocking. The issue rate is varied from eight to infinity, where the issue rate is the maximum number of instructions the processor can fetch and issue each cycle. The processor is assumed to have uniform functional units, except for branches. This places no restriction on the type of instructions that can be fetched and issued except for branches which are limited to one per cycle. The register file includes 64 predicate registers, 64 integer registers, and 64 floating-point registers. All of the experiments in this chapter utilize a perfect cache and memory system along with perfect branch prediction. The instruction set is an extended version of the instruction set for

Table 5.1 Instruction latencies.

Function	Latency	Function	Latency
Integer ALU	1	Floating-point ALU	2
Memory Load	2	Floating-point Multiply	2
Memory Store	1	Floating-point Divide (single)	8
Branch	1 / 1 slot	Floating-point Divide (double)	15

the HP PA-RISC processor, and the instruction latencies assumed are those of the HP PA-RISC 7100 as shown in Table 5.1. The extensions include silent versions of all excepting instructions and support for predicated execution. All experiments utilize the general speculation model to support speculative code motion.

5.1.2 Benchmarks

All evaluations in this thesis use the set of twenty-two benchmarks shown in Table 5.2. The benchmarks consist of three programs from the SPEC CINT95 suite, five programs from the SPEC CINT92 suite, eight common Unix utilities, and six from the MediaBench suite. The MediaBench suite is a relatively new set of benchmarks intended for embedded applications [48]. These benchmarks are evaluated because companies such as Motorola, Lucent, and Texas Instruments have expressed interest in including predication into their high performance digital signal processors [49], [50]. All of these benchmarks were chosen because of their control intensive nature and lack of easily exploitable ILP.

5.2 Results

The performance improvement presented in this section is calculated using a speedup calculation. Speedup is computed by dividing the total number of execution cycles for the base

Table 5.2 Benchmark set.

Benchmark	Benchmark Description
124.m88ksim	Architecture simulator (SPEC CINT95)
129.compress	File compression utility (SPEC CINT95)
130.li	Lisp interpreter (SPEC CINT95)
008.espresso	Truth table minimization (SPEC CINT92)
022.li	Lisp interpreter (SPEC CINT92)
023.eqntott	Boolean equation minimization (SPEC CINT92)
026.compress	File compression utility (SPEC CINT92)
072.sc	Spreadsheet (SPEC CINT92)
cccp	C preprocessor (Unix)
cmp	Compare (Unix)
eqn	Format math expressions for troff (Unix)
grep	Regular expression matcher (Unix)
lex	Lexical analyzer generator (Unix)
qsort	Quick sort (Unix)
wc	Word count (Unix)
yacc	Parser generator (Unix)
adpcm	audio encoding and decoding (MediaBench)
expic	Image compression (MediaBench)
g721	Voice compression (MediaBench)
jpeg	Jpeg image compression and decompression (MediaBench)
mpeg	Mpeg video encoding and decoding (MediaBench)
rasta	Speech recognition (MediaBench)

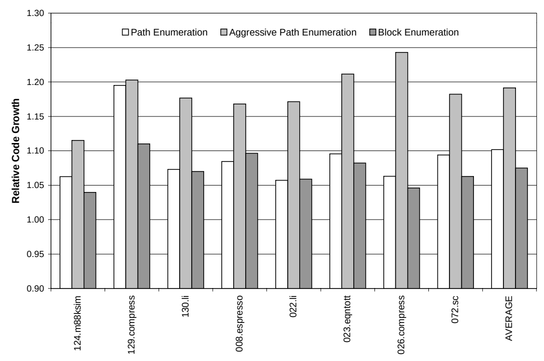
configuration by the total execution cycles of the test configuration. The base configuration for all experiments presented in this section is optimized basic block code compiled specifically for the three different processor configurations. A speedup value greater than one indicates a performance improvement. The code size results are presented as a relative change. This is calculated by dividing the code size for test configuration by the base configuration. Again, the base configuration is optimized basic block code. A value greater than one indicates that code expansion has occurred.

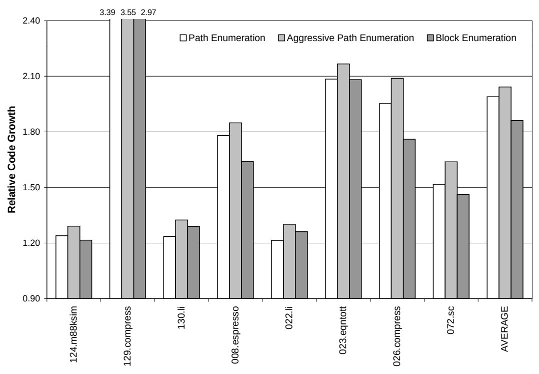
The block enumeration method will be compared against the path enumeration method and an aggressive version of the path enumeration method. The baseline IMPACT parameters are used with the path enumeration algorithm. For aggressive path enumeration, the minimum block weight and minimum path execution ratio parameters are set to zero. This forces the path enumeration algorithm to become more aggressive and form hyperblocks in low and zero weight regions. This was done so that low weight regions would be formed by the path enumeration algorithm, and comparisons could be made between it and the block enumeration algorithm. For all of these methods, the hyperblocks formed were specifically for an eight issue machine, and then the code was scheduled and run on each of the three different machine configurations.

5.2.1 Code size

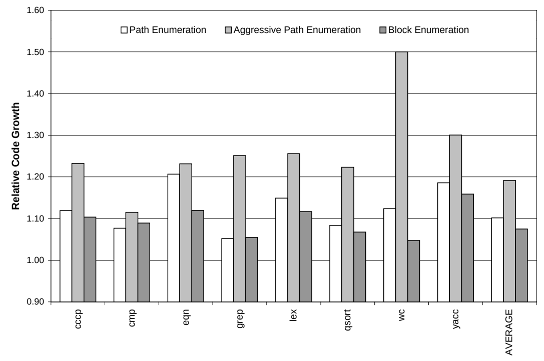
Code size is an important issue in evaluating the performance of any compiler optimization. The code size effects of block formation and superscalar optimization are shown in Figure 5.1. The graphs in the left column show the amount code growth for the benchmarks immediately after block formation. The graphs in the right column show the amount of code growth for the benchmarks after block formation and superscalar optimization. The code growth after block formation occurs because of tail duplication and loop peeling. The average code growth for path enumeration is 1.10, for aggressive path enumeration is 1.19, and for block enumeration is 1.07. The decrease in code growth for the block enumeration method comes from two factors. First, as was shown in Chapter 4 the block enumeration method has the potential to reduce tail duplication and therefore code size by including specific low weight blocks. Second, the block enumeration method also has the capability to reduce the code size of low and zero profile weight regions, regions that the path enumeration method completely ignores. Together, these two factors account for the smaller codes generated by the block enumeration method. The larger code size for aggressive path enumeration is the result of excessive tail duplication.

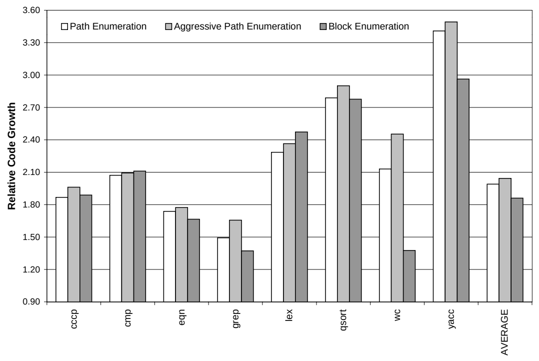
SPEC benchmark code growth.



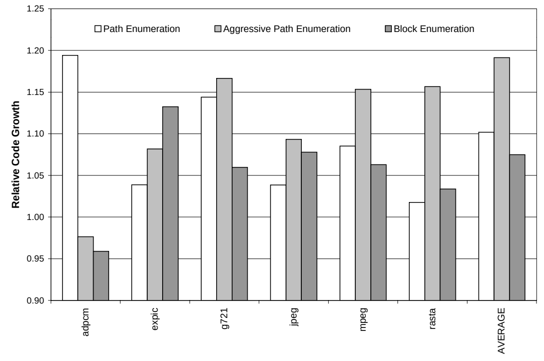


Unix benchmark code growth.





MediaBench benchmark code growth.



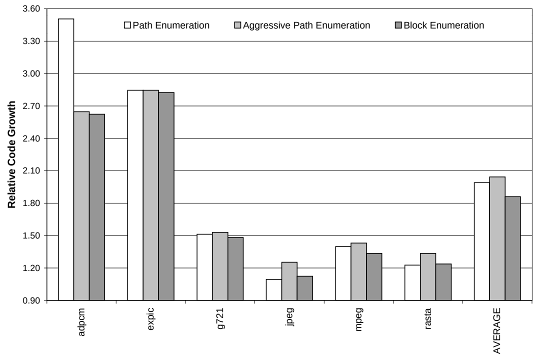


Figure 5.1 Comparison of code growth for each of the formation types.

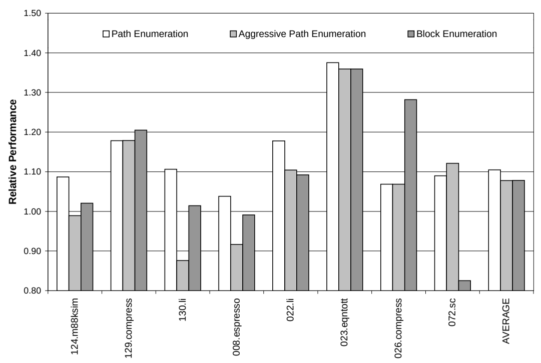
As will be shown later in this chapter, the aggressive algorithm forms a large number hyperblocks and this large number of hyperblocks causes an excessive amount of tail duplication.

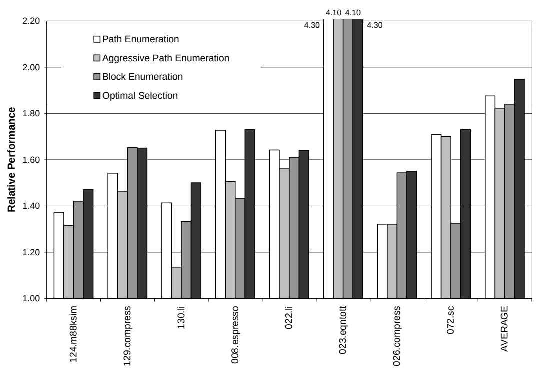
The graphs clearly show that block formation and superscalar optimization do increase code size, but this code growth also brings increased performance. After superscalar optimization the average code growth for each of the block selection methods is 1.99 for path enumeration, 2.04 for aggressive path enumeration, and 1.86 for block enumeration. The additional code growth observed after the superscalar optimizations occurs because optimizations such as loop unrolling and branch combining increase code size as well as ILP.

5.2.2 Performance

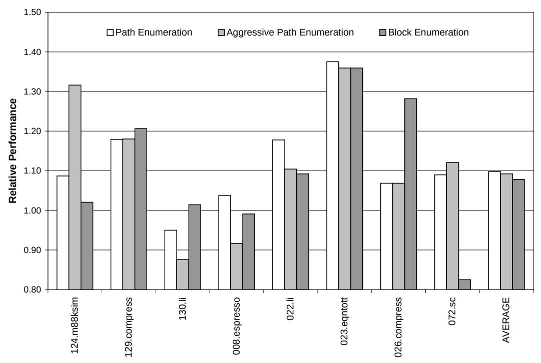
Figures 5.2, 5.3, and 5.4 show the performance of the benchmarks. As before, the column on the left is the performance immediately after block formation is completed, and the column on the right is the performance after block formation and superscalar optimization. On these graphs the dark bar marked optimal selection represents the best performance that can be obtained using the most intelligent combination of block formation techniques. On average the path enumeration has the best performance with an average speedup of 2.12 for the infinite issue machine. However, higher average speedup comes at the cost of greater code size as explained in the previous section. The block enumeration method has an average speedup of 2.03 and the aggressive path enumeration has a speedup of 2.04. At lower issue rates the aggressive path enumeration drops to 1.91 for the 16 issue machine and 1.82 for the 8 issue machine. The performance losses are likely due to the over-subscription of resources and increased schedule height caused by the large number of hyperblocks and the large amount of code growth.

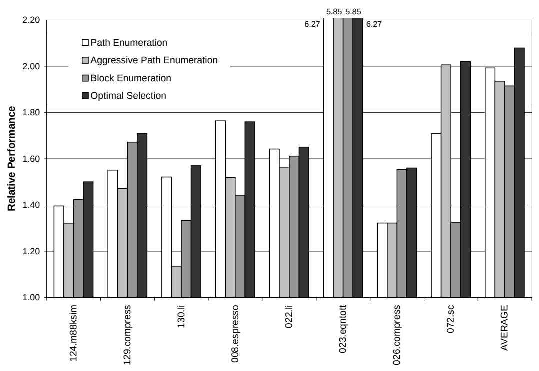
Eight issue performance for the SPEC benchmarks.



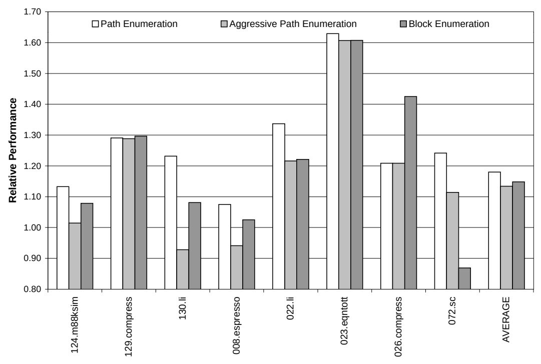


Sixteen issue performance for the SPEC benchmarks.





Infinite issue performance for the SPEC benchmarks.



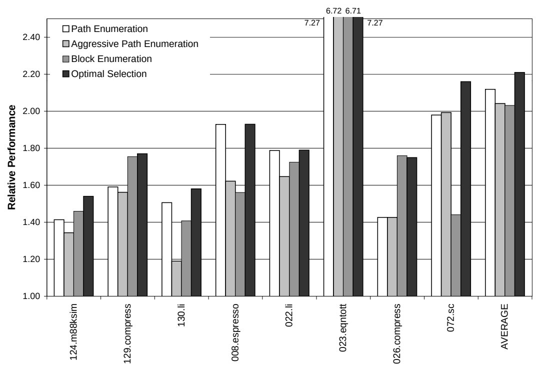
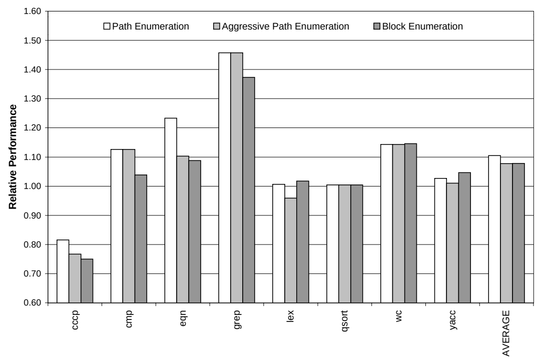
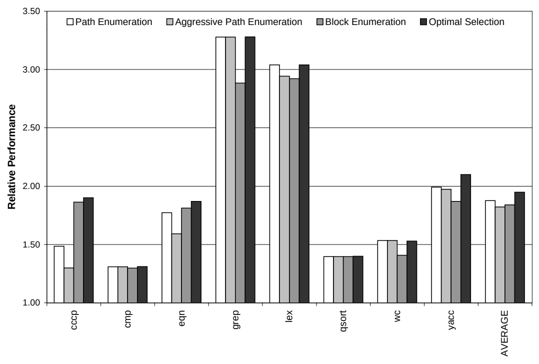


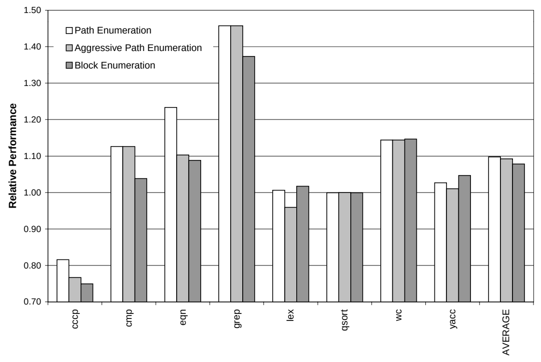
Figure 5.2 Performance measurements of the SPEC benchmarks.

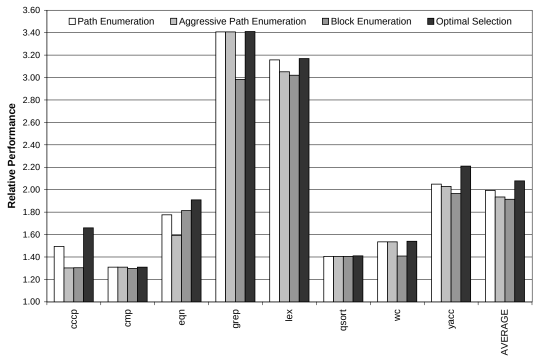
Eight issue performance for the Unix benchmarks.



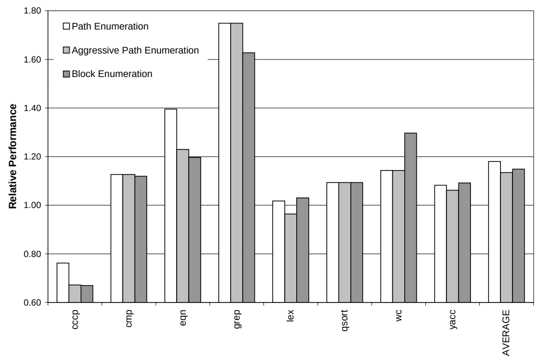


Sixteen issue performance for the Unix benchmarks.





Infinite issue performance for the Unix benchmarks.



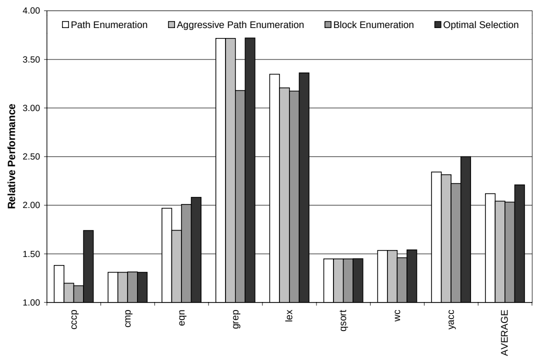
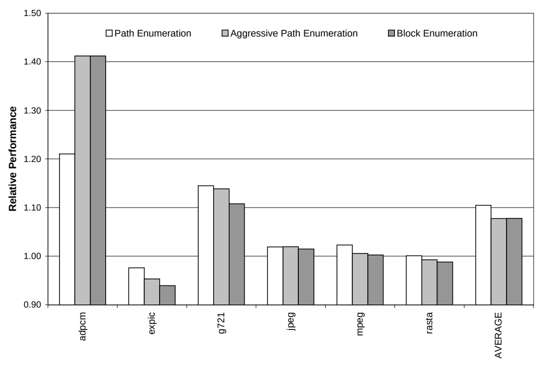
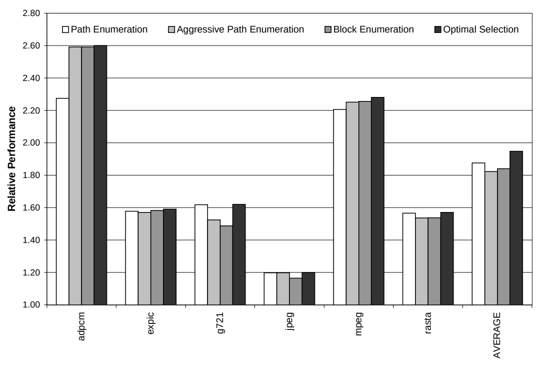


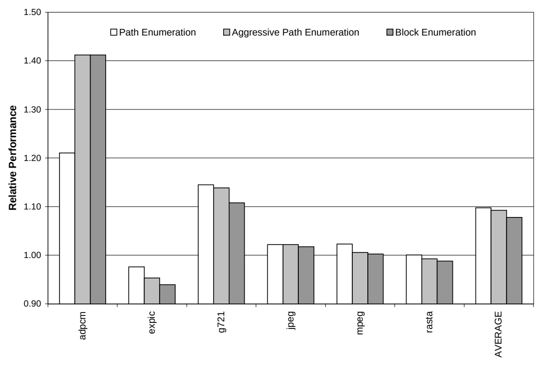
Figure 5.3 Performance measurements of the Unix benchmarks.

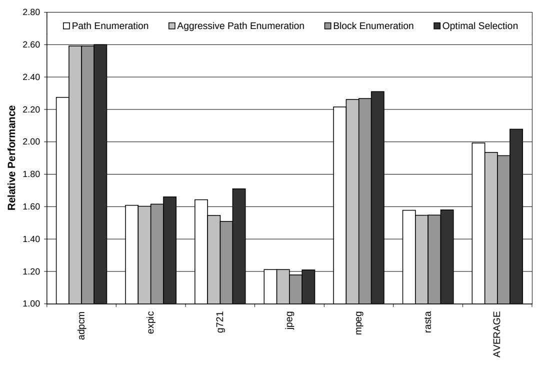
Eight issue performance for the MediaBench benchmarks.



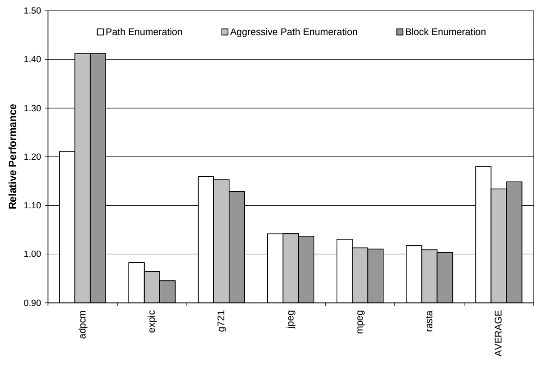


Sixteen issue performance for the MediaBench benchmarks.





Infinite issue performance for the MediaBench benchmarks.



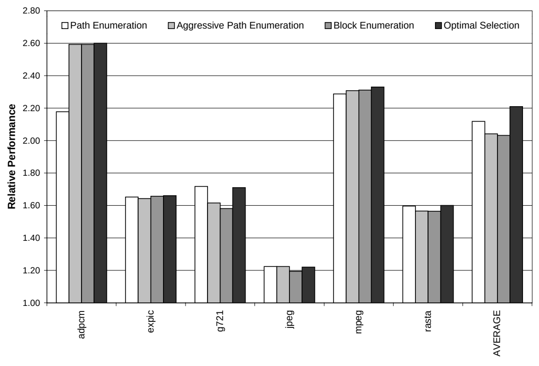
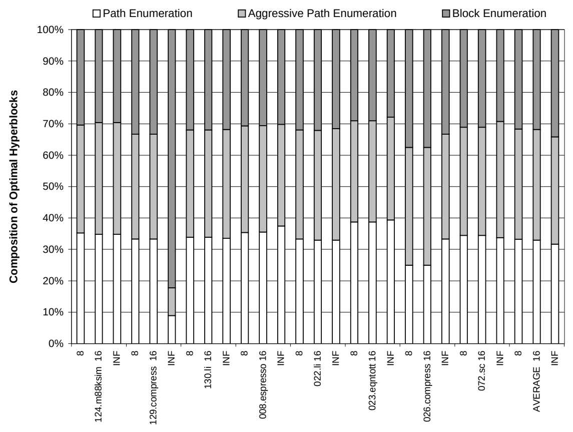


Figure 5.4 Performance measurements of the MediaBench benchmarks.

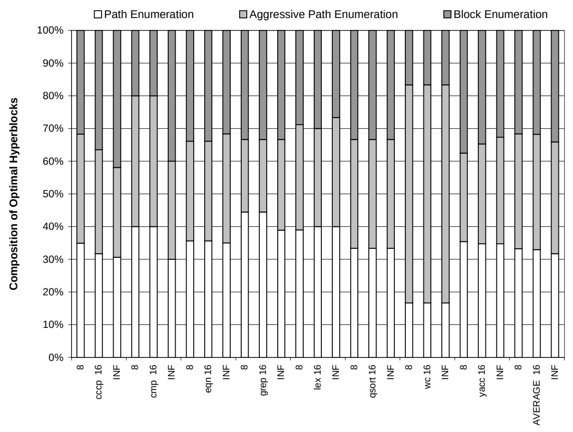
Figure 5.5 shows the composition of the optimal hyperblocks. The percentages presented in these graphs are weighted such that if each of the three methods formed an identical hyperblock it would count as 33% for each method. Similarly, if two algorithms formed the same hyperblock it would count as 50% for both methods. For non-low-weight hyperblocks, it would be expected that the path enumeration algorithm and the aggressive path enumeration algorithm would form identical hyperblocks. This is, in fact, observed for most benchmarks as on average both the path enumeration algorithm and aggressive path enumeration algorithm have even shares. In many cases, the block enumeration algorithm will also form identical hyperblocks to the path enumeration algorithm. This is also observed as on average the composition of the optimal blocks is approximately 33% for each algorithm. There are, of course, a few exceptional cases where one method or another forms more efficient hyperblocks. Some of these cases will be specifically addressed below.

A few benchmarks provide interesting cases worth examining. *072.sc* is one particular benchmark on which the block enumeration algorithm performs particularly poorly. It was determined that for two particular functions, accounting for 50% of the execution time, the algorithm was forming larger hyperblocks and these larger blocks had characteristics that prevented them from being unrolled during superscalar optimization. A decrease in performance by a factor of four is observed for these blocks indicating that loop unrolling is critical in extracting adequate amounts of ILP from these loops. Both *026.compress* and *adpcm* exhibit substantial gains for the block enumeration method. The reasons for the improvement of the *adpcm* benchmark are discussed in Chapter 4. The speedup obtained in *compress* results from decreased schedule height in the hyperblock that forms the main loop. In the path enumerated version, the main loop hyperblock contains eight branch operations. On the other hand, the

Optimal hyperblock composition for SPEC benchmarks.



Optimal hyperblock composition for Unix benchmarks.



Optimal hyperblock composition for MediaBench benchmarks.

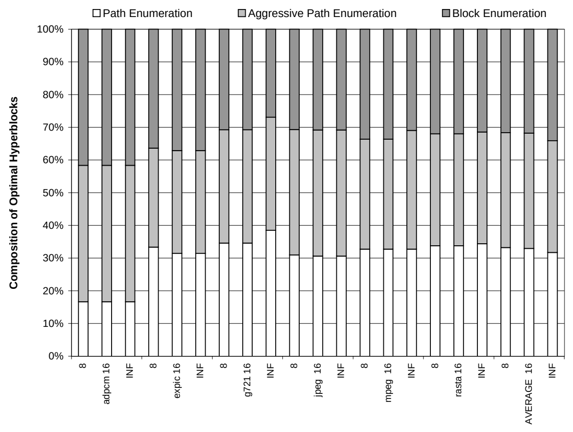


Figure 5.5 Comparison of the optimal hyperblock composition.

Table 5.3 Code size and speedup for selected functions.

Benchmark, Function	Path Enumeration		Block Enumeration		Comparison	
	Code Size	Exe. Cycles	Code Size	Exe. Cycles	Code Red.	Speedup
124.m88ksim, checklmt	123	121548	89	103703	0.72	1.17
130.li, xllist_c	149	490775	39	662573	0.25	0.74
008.espresso, read_cube	826	422116	624	273386	0.75	1.54
022.li, xsetq	39	16407	27	16407	0.69	1.00
023.eqntott, qsort	783	372294	616	372163	0.78	1.00
026.compress, compress	1481	33171076	1244	28394933	0.83	1.17
072.sc, num_search	298	14756	106	6307	0.35	2.34
072.sc, sync_refs	1782	6803917	1136	3418439	0.63	1.99
eqn, getstr	359	2190942	349	1832803	0.97	1.20
lex, yyparse	1003	54108	952	29200	0.94	1.85
yacc, skipcom	239	95097	190	82036	0.79	1.16
adpcm, adpcm_decoder	378	1649428	232	1446880	0.61	1.14
expic, run_length_decode	459	151299	298	151429	0.64	1.00

main hyperblock loop in the block enumerated version consists of three branch operations. In this case the block enumeration version actually forms a smaller hyperblock; however, it turns out to be a more efficient hyperblock.

All of the methods seem to perform at about the same level and near the optimal selection for the MediaBench benchmarks. The MediaBench benchmarks are smaller programs intended for embedded systems with very specific applications. The benchmarks tend to contain small tight inner loops, and all the methods handle this type of region equally well. Therefore, all the methods rely on the superscalar optimizations to extract the ILP from the hyperblock loops they form.

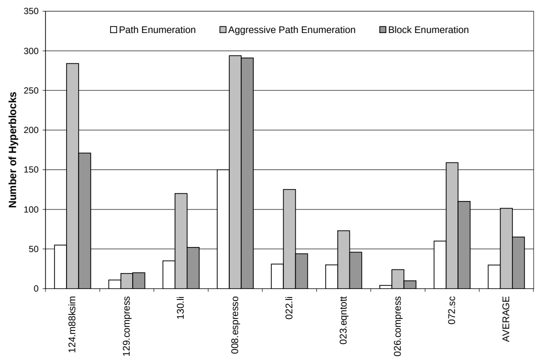
Table 5.3 shows code size and performance of specific functions from the tested benchmarks. These functions were selected because they show that it is possible to reduce code size by significant amounts and still maintain or even increase the performance of the same code. The most notable of these functions is num_search from *072.sc*. Its code size is cut by almost 66%

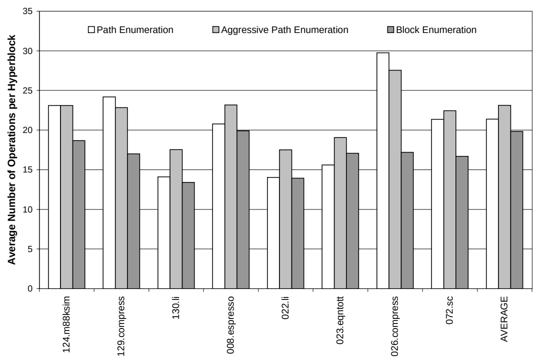
and the performance is more than doubled. The block enumeration method forms one larger main loop hyperblock while the path enumeration method forms two smaller hyperblocks. This single larger hyperblock gives the scheduler a larger region of instructions to select from and consequently more freedom to create a small compact schedule for the loop. While this table shows substantial performance gains for two functions from *072.sc* these gains cannot erase the performance lost from the previously described functions that could not be unrolled. Some of the functions such as getstr from *eqn* and yyparse from *lex* do not show much code size reduction, they do show some performance increase from the block enumeration method. The function xllist_c from *022.li* is another interesting case. The code size is reduced by a factor of four and only a minimal performance loss is observed. These examples illustrate that it is possible to reduce code size, while maintaining and sometimes even increasing performance.

Figure 5.6 shows statistics concerning the numbers and sizes of hyperblocks created during block formation. As expected, the aggressive path enumeration method forms the largest number of hyperblocks consisting of on average the largest number of operations. Further proof that the aggressive algorithm is over-predicating for the given architecture. The block enumeration method forms on average 65 hyperblocks per benchmark, consisting of 20 operations, while the path enumeration method forms on average 30 hyperblocks per benchmark, consisting of 21 operations. Contributing to the number of hyperblocks formed with the block enumeration method are the low and zero profile weight blocks. The performance effects of predicating these regions cannot be measured, but their contribution to the number of hyperblocks is noted.

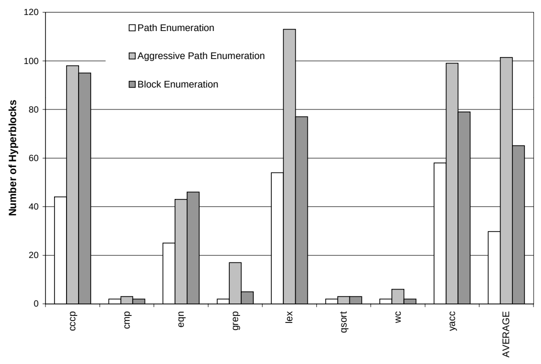
Figure 5.7 shows the percentage of zero and low weight hyperblocks formed by the aggressive path enumeration method the block enumeration method. The path enumeration method is

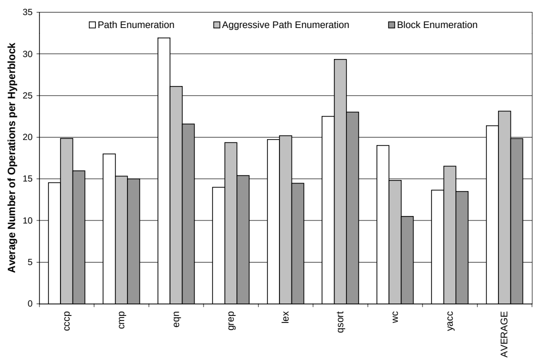
Hyperblock statistics for SPEC benchmarks.



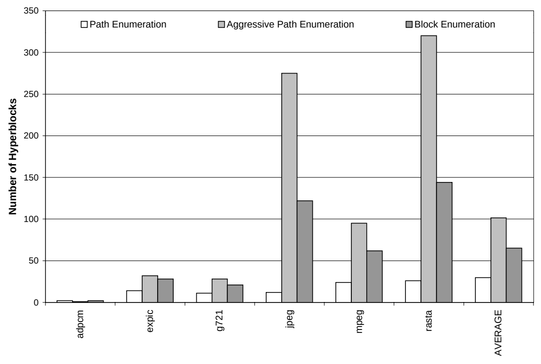


Hyperblock statistics for Unix benchmarks.





Hyperblock statistics for MediaBench benchmarks.



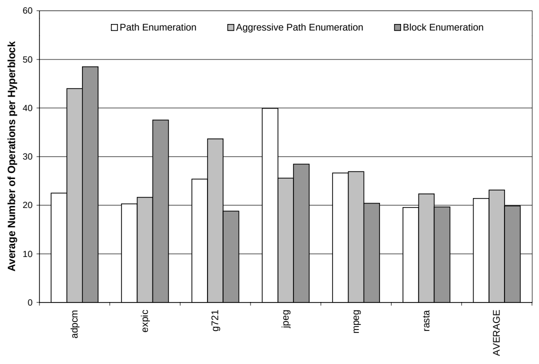
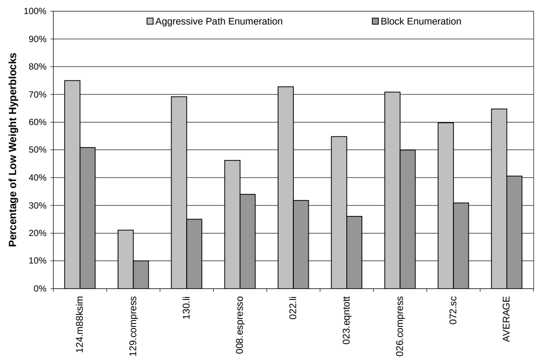
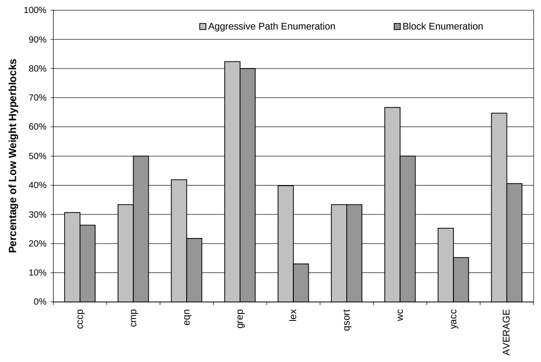


Figure 5.6 Comparison of hyperblock statistics.

Low weight hyperblock statistics for SPEC benchmarks.



Low weight hyperblock statistics for Unix benchmarks.



Low weight hyperblock statistics for MediaBench benchmarks.

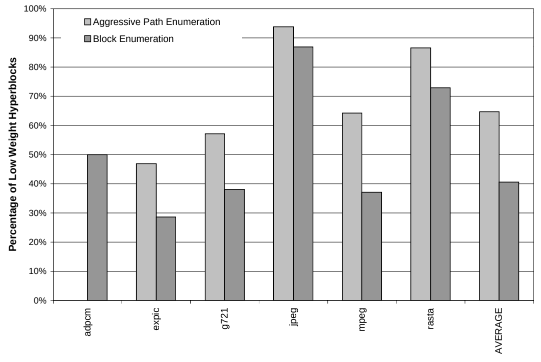


Figure 5.7 Comparison of the percentage low weight hyperblocks formed.

not included in these graphs because with the baseline parameters it does not form any low weight regions. The graphs clearly show that aggressive path enumeration forms a large number of low weight hyperblocks. Nearly 65% of the hyperblocks formed by aggressive path enumeration are in low weight regions, meaning that, of the 100 hyperblocks formed, on average per benchmark by aggressive path enumeration, 65 of them are in low weight regions. The large amount of code growth and mediocre performance of the aggressive path enumeration method indicate that the method is too aggressive and is forming too many hyperblocks. On the other hand, only about 40% of the hyperblocks formed by block enumeration are low weight, and the formation of these blocks does not appear to hinder performance and actually contributes to a reduction in code size when compared to the path enumeration method. This translates to on average about 26 low weight hyperblocks formed per benchmark by block enumeration.

Decreased compilation time is another advantage of block enumeration. Table 5.4 presents the compile time results for both of the block selection methods. The compilation and memory usage experiments were run on an unloaded HP 9000/780 workstation with 384 MB of memory and a 180 MHz PA-RISC processor. Because block enumeration forms regions over low profile weight regions, the parameters for path enumeration were set such that it would also make an attempt to operate on low weight regions for this experiment. It should be noted that the aggressiveness of these parameter settings was far less than the aggressive settings used in the previous experiments for the aggressive path enumeration method. This was done so that a fair comparison between both methods could be made. For most all of the benchmarks, the compile speed for block enumeration is faster than for path enumeration. In some cases, the compile time for block enumeration is almost half that of path enumeration. The regions in these benchmarks tend to be large with complex control flow, and this leads to a large number

Table 5.4 Compile time and memory usage for the benchmark set.

Benchmark	Path Enumeration		Block Enumeration		Ratios	
	Comp. Time	Mem. Usage	Comp. Time	Mem. Usage	Comp. Time	Mem. Usage
124.m88ksim	9:47.89	5,912 kB	6:25.37	5,277 kB	0.66	0.89
129.compress	0:24.96	1,271 kB	0:18.39	1,246 kB	0.74	0.98
130.li	3:59.82	1,521 kB	2:26.42	1,603 kB	0.61	1.05
008.espresso	9:35.67	4,232 kB	6:10.85	4,314 kB	0.64	1.02
022.li	3:54.34	1,410 kB	2:23.34	1,427 kB	0.61	1.01
023.eqntott	6:16.67	1,767 kB	1:00.47	1,373 kB	0.16	0.77
026.compress	0:43.78	2,627 kB	0:22.38	2,139 kB	0.51	0.81
072.sc	9:23.98	9,688 kB	4:42.28	9,787 kB	0.50	1.01
cccp	5:09.74	5,899 kB	3:11.69	5,400 kB	0.62	0.92
cmp	0:04.06	1,943 kB	0:03.88	1,287 kB	0.95	0.66
eqn	1:52.09	2,680 kB	1:03.32	2,049 kB	0.56	0.76
grep	0:19.68	1,709 kB	0:14.70	1,738 kB	0.75	1.02
lex	3:49.00	5,550 kB	2:26.81	4,736 kB	0.64	0.85
qsort	0:03.49	984 kB	0:03.64	980 kB	1.04	1.00
wc	0:03.96	1,087 kB	0:03.11	1,091 kB	0.78	1.00
yacc	2:04.99	3,532 kB	1:31.74	2,791 kB	0.73	0.79
adpcm	0:23.36	1,496 kB	0:02.39	997 kB	0.10	0.67
expic	0:36.97	6,080 kB	0:33.83	5,220 kB	0.92	0.85
g721	0:18.73	7,182 kB	0:18.47	7,165 kB	0.99	0.99
jpeg	5:26.28	12,207 kB	4:17.44	12,449 kB	0.78	1.02
mpeg	2:58.72	6,416 kB	1:34.07	6,375 kB	0.53	0.99
rasta	17:43.23	9,418 kB	13:39.84	9,615 kB	0.77	1.02

of paths to enumerate. Consequently, the path enumeration algorithm is slowed by having to enumerate the large number of paths. It is further slowed by having to store and sort the large number of paths. In a few cases, the compile times are nearly equal, indicating that the benchmark is dominated by smaller regions with only a few execution paths.

Smaller memory footprint is one final advantage observed from Table 5.4 for block enumeration. The memory usage columns in the table represent the memory required to compile the most complex function of the benchmark using each block selection algorithm. For many of the benchmarks, block enumeration requires less memory than path enumeration. This occurs because block enumeration does not incur the overhead of storing the potentially large numbers of paths created by the path enumeration method. In some cases, the memory usage of the two methods is approximately equal, indicating that the overhead of storing the paths and the blocks is the same. The regions in these benchmarks are smaller and less complex, so less paths exist for enumeration, thus requiring less storage space. These results further illustrate that enumerating the blocks in a region is more efficient than enumerating all the execution paths in a region.

In summary, an extensive evaluation of both the path enumeration and block enumeration algorithms has been presented in this chapter. The performance of the path enumeration algorithm was on average slightly better than the block enumeration algorithm. However, application of the block enumeration algorithm results in smaller code size, faster compilation time, and reduced memory footprint. The optimal results show that both algorithms, when utilized together, can be used to obtain substantial performance improvements over optimized basic block code.

CHAPTER 6

CONCLUSION AND FUTURE WORK

6.1 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 thesis, one technique to overcome these difficulties was investigated: predicated execution.

Specifically, this thesis described an advanced hyperblock block selection heuristic, called block enumeration. Block enumeration is one heuristic that can be used to select blocks for inclusion in a hyperblock. When selecting blocks, block enumeration relies more on the structure and layout of a region, and less on the execution paths that run through it, .

Experimental results show that block enumeration can result in substantial performance gains while at the same time reducing code size. These performance gains come from including low weight blocks that in the past were normally excluded. Including these low weight blocks can result in smaller schedule height and reduced tail duplication. Reduced code size is also observed because the block enumeration method is able to form hyperblocks in regions with very little or no profile weight. While neither path enumeration or block enumeration can form optimal hyperblocks all the time, working together these heuristics can form optimal hyperblocks for a wide a variety of programs.

This work, combined with the work of others in the field, shows that predication is an extremely valuable tool in extracting instruction-level parallelism from programs.

6.2 Future Work

The work presented in this thesis further motivates some promising opportunities for future research. These include the areas of block selection heuristics and static block selection.

The first opportunity is in the area of block selection. This thesis only examined path enumeration alone and block enumeration alone. The best hyperblock heuristic would use both of these heuristics together and would have the ability to determine which heuristic is forming the more optimal hyperblock. In addition, with the use of path profiling, the path enumeration algorithm could be made to be far more efficient [51],[52],[53]. With the information from path profiling, the path enumeration algorithm would no longer need to enumerate all the paths. It would simply assign priorities to the paths discovered during path profiling, and select the best paths for inclusion in the hyperblock.

The second opportunity is in the area of static block formation. This thesis provides only a brief examination of hyperblock formation in the absence of profile information. A block selection heuristic built with both static branch prediction and static profile estimation could be used to form hyperblocks in systems in which dynamic profiling is not possible.

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*, April 1989, pp. 272–282.

[2] 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*, May 1991, pp. 276–286.

[3] P. Y. Hsu and E. S. Davidson, "Highly concurrent scalar processing," in *Proceedings of the 13th International Symposium on Computer Architecture*, June 1986, pp. 386–395.

[4] B. R. Rau, D. W. L. Yen, W. Yen, and R. A. Towle, "The Cydra 5 departmental supercomputer," *IEEE Computer*, vol. 22, no. 1, pp. 12–35, January 1989.

[5] 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*, January 1983, pp. 177–189.

[6] J. C. Park and M. S. Schlansker, "On predicated execution," Hewlett Packard Laboratories, Palo Alto, CA, Tech. Rep. HPL-91-58, May 1991.

[7] G. R. Beck, D. W. Yen, and T. L. Anderson, "The Cydra 5 minisupercomputer: Architecture and implementation," *The Journal of Supercomputing*, vol. 7, no. 1, pp. 143–180, January 1993.

[8] J. C. Dehnert and R. A. Towle, "Compiling for the Cydra 5," *The Journal of Supercomputing*, vol. 7, no. 1, pp. 181–227, January 1993.

[9] J. C. Dehnert, P. Y. Hsu, and J. P. Bratt, "Overlapped loop support in the Cydra 5," in *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems*, April 1989, pp. 26–38.

[10] A. V. Someren and C. Atack, *The ARM RISC Chip, A Programmer's Guide.* Reading, MA: Addison-Wesley, 1994.

[11] D. L. Weaver and T. Germond, *The SPARC Architecture Manual.* Menlo Park, CA: SPARC International, Inc., 1994.

[12] Digital Equipment Corporation, *Alpha Architecture Handbook.* Maynard, MA: Digital Equipment Corporation, 1992.

[13] Intel Corporation, *The Pentium Microprocessor.* Santa Clara, CA, 1993.

[14] Hewlett-Packard Company, Cupertino, CA, *PA-RISC 1.1 Architecture and Instruction Set Reference Manual*, 1990.

[15] 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, no. 1, pp. 51–142, January 1993.

[16] H. Zima and B. Chapman, *Supercompilers for Parallel and Vector Computers.* Reading, MA: Addison-Wesley, 1991.

[17] L. Gwennap, "Intel, HP make EPIC disclosure," *Microprocessor Report*, vol. 11, no. 14, pp. 1–9, October 1997.

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

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

[20] 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, no. 1, pp. 229–248, January 1993.

[21] S. A. Mahlke, D. C. Lin, W. Y. Chen, R. E. Hank, R. A. Bringmann, and W. W. Hwu, "Effective compiler support for predicated execution using the hyperblock," in *Proceedings of the 25th International Symposium on Microarchitecture*, December 1992, pp. 45–54.

[22] D. Callahan and B. Koblenz, "Register allocation via hierarchical graph coloring," in *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*, June 1991, pp. 192–203.

[23] P. P. Chang, S. A. Mahlke, and W. W. Hwu, "Using profile information to assist classic code optimizations," *Software Practice and Experience*, vol. 21, no. 12, pp. 1301–1321, December 1991.

[24] J. A. Fisher and S. M. Freudenberger, "Predicting conditional branch directions from previous runs of a program," in *Proceedings of 5th International Conference on Architectual Support for Programming Languages and Operating Systems*, October 1992, pp. 85–95.

[25] T. Ball and J. R. Larus, "Branch prediction for free," in *Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation*, June 1993, pp. 300–313.

[26] D. W. Wall, "Predicting program behavior using real and estimated profiles," in *Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation*, June 1991, pp. 59–70.

[27] C. V. Ramamoorthy, "Discrete markov analysis of computer programs," in *Proceedings of ACM 20th Annual National Conference*, 1965, pp. 386–391.

[28] T. A. Wagner, V. Maverick, S. L. Graham, and M. A. Harrison, “Accurate static estimators for program optimization,” in *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation*, June 1994, pp. 85–96.

[29] Y. Wu and J. R. Larus, “Static branch prediction and program profile analysis,” in *Proceedings of the 27th Annual International Symposium on Microarchitecture*, December 1994, pp. 1–11.

[30] 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*, pp. 247–255.

[31] B. L. Deitrich, “Static program analysis to enhance profile independence in instruction-level parallelism compilation,” Ph.D. thesis, Department of Electrical and Computer Engineering, University of Illinois, Urbana, IL, 1998.

[32] 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.

[33] D. M. Gallagher, “Memory disambiguation to facilitate instruction-level parallelism compilation,” Ph.D. thesis, Department of Electrical and Computer Engineering, University of Illinois, Urbana, IL, 1995.

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

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

[36] Y. Yamada, “Data relocation and prefetching for programs with large data sets,” Ph.D. thesis, Department of Computer Science, University of Illinois, Urbana, IL, 1995.

[37] 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*, May 1989, pp. 242–251.

[38] P. P. Chang, “Compiler support for multiple instruction issue architectures,” Ph.D. thesis, Department of Electrical and Computer Engineering, University of Illinois, Urbana, IL, 1991.

[39] 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, no. 5, pp. 349–370, May 1992.

[40] 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.

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

[42] 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, no. 3, pp. 353–370, March 1995.

[43] D. M. Lavery, "Modulo scheduling for control-intensive general-purpose programs," Ph.D. thesis, Department of Electrical and Computer Engineering, University of Illinois, Urbana, IL, 1997.

[44] 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, 1995.

[45] J. C. Gyllenhaal, B. R. Rau, and W. W. Hwu, "HMDES version 2.0 specification," IMPACT, University of Illinois, Urbana, IL, Tech. Rep. IMPACT-96-03, 1996.

[46] D. I. August, D. A. Connors, S. A. Mahlke, J. W. Sias, K. M. Crozier, B. Cheng, P. R. Eaton, Q. B. Olaniran, and W. W. Hwu, "Integrated predication and speculative execution in the IMPACT EPIC architecture," in *Proceedings of the 25th International Symposium on Computer Architecture*, June 1998, pp. 227–237.

[47] D. I. August, "Hyperblock performance optimizations for ILP processors," M.S. thesis, Dept. of Electrical and Computer Engineering, University of Illinois, Urbana, IL, 1996.

[48] C. Lee, M. Potkonjak, and W. Mangione-Smith, "Mediabench: A tool for evaluating and synthesizing multimedia and communications systems," in *Proceedings of the 30th Annual International Symposium on Microarchitecture*, December 1997, pp. 330–335.

[49] T. R. Halfhill, "StarCore reveals it's first DSP," *Microprocessor Report*, vol. 13, no. 6, pp. 13–16, May 1999.

[50] J. Turleyand and H. Hakkarainen, "TI's new 'C6x DSP screams at 1,600 MIPS," *Microprocessor Report*, vol. 11, no. 2, pp. 14–17, February 1997.

[51] T. Ball and J. R. Larus, "Efficient path profiling," in *Proceedings of 29th Annual Int'l Symposium on Microarchitecture*, December 1996, pp. 46–57.

[52] T. Ball and J. R. Larus, "Programs follow paths," Microsoft Research, Microsoft Research, Redmond, WA, Tech. Rep. MSR-TR-99-01, January 1999.

[53] D. Melski and T. Reps, "Interprocedural path profiling," Department of Computer Sciences, University of Wisconsin Madison, Tech. Rep. CS-TR-98-1382, September 1998.