SUPPORTING PREDICATED EXECUTION: TECHNIQUES AND TRADEOFFS

BY

JAMES EARL MCCORMICK, JR.

B.S., University of Illinois, 1994

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, 1996

Urbana, Illinois

ABSTRACT

Predicated execution is the ability to associate a predicate, or guard, with an instruction and to use that predicate to determine whether the instruction executes or not. Much of the research to date on predicated execution assumes a high level of architectural support for predicated execution and focuses developing techniques to exploit these capabilities. Researchers have shown that predicate execution is a very useful tool in exposing and exploiting instruction level parallelism. Unfortunately, there are several factors that make it very difficult to develop a processor with a high level of architectural support for predicated execution. A big factor is that the importance of backward compatibility means that support for predication would usually have to come as an extension on an existing instruction set. Another factor is that some features which are assumed in a high level of support for predicated execution are difficult and costly to implement. Given these factors, techniques have to be developed that support predicated execution at various levels of performance and cost, and the tradeoffs among these techniques and levels of predication support need to be studied. This thesis discusses several existing techniques and presents several new techniques that support various levels of predicated execution and discusses the tradeoffs among these techniques and levels of support.

ACKNOWLEDGMENTS

There are many people who have contributed significantly to this work and without whom it would not have been possible. I can't mention everyone, but I'll try to mention those who contributed most.

My advisor, Professor Wen-mei Hwu, has helped to create a wonderful learning environment and allowed me to be part of it. He provided me with the resources and opportunities to learn and pushed me to use them to the best of my ability. I have benefited much from his advice and insight.

Scott Mahlke (now at HP Laboratories) pointed me down this path of research and has guided and contributed to my efforts every step of the way. He patiently answered many questions and helped me solve many problems. His *hyperblock* research and tools have been the basis for much of this research.

Rick Hank (now at HP Laboratories) has helped me solve many problems and answered many of my questions. He provided me with helpful advice along the way. He wrote the HP PA-RISC code generator and emulation support that was used in this research.

Many other people in the IMPACT research group have contributed to this work in various ways including providing a wonderful learning environment and developing various parts of the IMPACT compiler and associated tools.

My wife Iris has been a great encouragement to me. She carried more than her share of the load when I was overloaded even though she too is in graduate school. She listened to me even when I talked about my research. Her love and friendship have helped me survive.

My parents put me through undergraduate school with considerable sacrifice, and have contributed to my life in too many ways to mention. Without them, I would not be who I am or where I am.

All of these opportunities and people did not come into my life by chance. My strength does not come from myself. These things came from my God and Savior, the Maker of Heaven and Earth. Therefore, I intend to use what I have gained for Him.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Figure Page

CHAPTER 1

INTRODUCTION

1.1 Predicated Execution Background

In the unending quest to squeeze more performance out of microprocessors, researchers and designers are working to expose and exploit ever-increasing amounts of instruction level parallelism (ILP). Exposing large amounts of ILP is primarily the job of the compiler. Exploiting the exposed ILP is primarily the job of the processor architecture. However, these two jobs are not completely separable. The architecture and the compiler must work together in order to exploit large amounts of ILP. One area of research where such joint compiler and architecture work is particularly evident and effective is predicated execution, also referred to as guarded execution.

1.1.1 Motivation for predicated execution

Exposing and exploiting significant amounts of ILP are difficult tasks. A number of architectural and compiler techniques that expose and exploit ILP have been suggested. These techniques have been somewhat successful. However, to fully utilize 4 to 8 or more functional units per cycle, one must overcome a number of significant obstacles.

From an architectural perspective, branches remain a significant problem. Much general purpose code is control intensive code, in which branches make up as much as 40% of the instruction mix. While branch prediction [2], [3], [4] and run-time speculation have been effective at exposing and exploiting some ILP across branches, the cost and difficulty of doing good branch prediction, among other things, limit most architectures to handling only one branch per cycle. In code with 40% branches, such a constraint would limit IPC to 2.5. In addition, in many architectures, a large performance penalty is paid for each mispredicted branch. Clearly, in order to increase IPC up to 4 or 8 branches must be dealt with.

From a compiler perspective, branches are an obstacle to exposing ILP in several ways. First of all, while quite a bit is known about optimizing under the constraints of data flow, not as much is known about optimizing under the constraints of control flow. Second, scheduling simultaneously across multiple important control flow paths is very difficult. Scheduling simultaneously across multiple paths is important when paths of similar execution frequency overlap. Paths like these are caused by conditional branches which are not heavily biased one way or the other. Interestingly, these branches are also the hardest to predict and therefore, cause a significant amount of performance loss due to branch misprediction penalties. Compiler techniques such as trace scheduling [5], superblock scheduling [6], loop unrolling, loop peeling, and software pipelining [7] all focus on scheduling single cyclic or acyclic paths. Branches, then, are a big obstacle to exposing and exploiting ILP and must be dealt with. Predicated execution alleviates some of the difficulties which branches impose on exposing and exploiting ILP.

1.1.2 Definition of predicated execution

Predicated execution is conditional execution of instructions based upon a Boolean value called a *predicate*. We say that an instruction is *predicated*, or *guarded*, when it has a corresponding predicate. A predicated instruction with a *true* predicate executes normally. A predicated instruction with a *false* predicate is nullified, and thus prevented from modifying the processor state. An instruction that defines a predicate is called a *predicate definition instruction*. Predicated execution requires both architectural and compiler support.

1.1.3 Benefits

Predicated execution is a powerful technique with many benefits. One major benefit is that predicated execution combined with if-conversion allows the removal of branches from the dynamic instruction stream. Removing these branches reduces the number of branches the processor encounters each cycle. The importance of this is that many processors are very limited in their branch issue rate. In addition, removing branches from the dynamic instruction stream tends to reduce the number of branch mispredictions and the associated misprediction penalty. The second major benefit is that with predicated execution multiple paths of control can be overlapped into a single instruction stream. This overlap allows optimization and scheduling along multiple paths simultaneously. The third major benefit is that the conversion of control flow to data flow through if-conversion enables optimizations which were not feasible in the control flow domain. These additional optimizations are due to the fact that optimizing data dependences is more understood than optimizing control dependences.

1.2 Support for Predicated Execution

1.2.1 Processors/architectures with predicated execution support

A number of processors and architectures have included various kinds of predicated execution support. Some vector machines such as CRAY-1 used a vector mask register to control the issue of instructions. A number of current processors and architectures (Pentium Pro, DEC Alpha, SPARC V9) support conditional move instructions. The Multiflow Trace 300 series machines supported a *select* instruction which selects one of two source operands to copy to the destination operand. The HP PA-RISC architecture [8] supports instruction nullification by which some instructions can squash the execution of the next instruction.

The Cydra 5 had extensive predication support [9], [10]. The predication support in the Cydra 5 was mainly used to enhance the ability of modulo scheduling to expose ILP across loop iterations in numerical code. Each instruction in this architecture had a predicate operand. The predicates were defined by compare operations and special loop control instructions. Instruction predicates were stored in a separate predicate register file called the *iteration control register*, or ICR. The predicates were referenced relative to the iteration frame pointer, allowing a distinct set of physical registers to be used by each iteration of a loop. The predicates served two general purposes. One set of predicates kept track of those iterations of the loop which were executing. The value of these predicates was controlled by the the special loop control instructions. The idea is that, with help from the compiler, code from different iterations of the loop could execute simultaneously. The other predicates were used to determine which path through the loop would be executed. These predicates were defined by compare instructions. The Cydra 5 had the most extensive support for predication of any implemented architecture to date.

1.2.2 Motivation for research

Much of the research to date on predicated execution assumes a very high level of architectural support for predicated execution and focuses developing techniques to exploit these capabilities. This research has shown that predicate execution is a very useful tool in exposing and exploiting instruction level parallelism. Unfortunately, there are several factors which make it very difficult to develop a processor with a very high level of architectural support. A big factor is that the importance of backward compatibility means that support for predication would usually have to come as an extension on an existing instruction set. Another factor is that some features which are assumed in a very high level of support for predicated execution are difficult and costly to implement. Given these factors, techniques have to be developed which support predicated execution at various levels of performance and cost, and the tradeoffs among these techniques and levels of predication support need to be studied. This thesis discusses some of the possible architectural and compiler support options for predicate execution and briefly mentions some of the previous research in this area. Some new compiler techniques that support various levels of predicated execution are presented. Experimental results are presented and used to explore tradeoffs in the area of architectural and compiler support for predicated execution.

Chapter 2 discusses various design parameters of architectural support for predicated execution and presents the features used for the experiments in this thesis. Chapter 3 contains a general discussion of compiler support for predicated execution and then presents the compiler techniques used in the experiments. Chapter 4 presents experiments and the results. This information is used to discuss tradeoffs in the compiler and architectural support. The knowledge gained in Chapter 4 is used to choose a new configuration of architectural and compiler support for predicated execution. This new configuration is presented in Chapter 5 along with corresponding experimental results. Finally, Chapter 6 summarizes the ideas discussed in this thesis and suggests areas for further research.

CHAPTER 2

ARCHITECTURAL SUPPORT

This chapter discusses architectural support for predicated execution. Section 2.1 discusses some of the possible parameters involved in choosing architectural support for predicated execution. Sections 2.2 and 2.3 present two different levels of architectural support used in the experiments in this thesis.

2.1 Background

Many possible varieties of architectural support exist for predicated execution. The choice of the architectural support to provide is based on balancing the performance potential, cost, and impact of the support on the instruction set architecture (ISA). A number of general style choices exist. One choice is deciding on what predicate definition instructions to supply in the ISA. This choice can greatly affect the efficiency of the predicated execution as well as the cost of the architectural support.

A second choice is whether to support full or partial predication [1]. *Full predication* support refers to architectural support which allows any instruction (or most instructions) to be predicated. *Partial predication* support refers to architectural support which allows only a limited subset of instructions to be predicated. Full predication support provides more flexibility and more potential benefit, while partial predication support is less costly to implement and has less of an impact on the ISA.

A third choice is how to associate the predicates with the predicated instructions. One approach is to make the predicate an operand on the predicated instruction [11]. Another approach is to have some kind of mask register which holds the predicates and associates the predicates with the instructions based on instruction arrangement [12]. The first approach will be referred to as *operand-specified predicates* (OSP), and the second approach will be referred to as *mask-specified predicates* (MSP). OSP is the more flexible of the two approaches. This approach requires a register to hold the predicate and a source operand on the instruction to specify the predicate register. The advantage of OSP is that the predicates can be manipulated in much the same manner as other operands. The MSP approach is less flexible because of the association between the mask and the instruction arrangement. This association limits the range over which a predicate may be live and constrains the ability to dynamically rearrange instructions. However, the MSP approach has less of an impact in the ISA.

A fourth choice is how to suppress instructions with false predicates. In the *convert-to-NOP* model instructions with false predicates are converted to NOPs at issue time. In the *data selection* model, the predicate selects either the original destination value or the new destination value generated by the predicated instruction. The main disadvantage of the convert-to-NOP model is that the predicated instruction cannot be issued until its predicate is ready. The main disadvantage of the data selection model is that artificial flow dependences are created for instructions whose predicates do not overlap. These artificial flow dependences can have a serializing effect on instruction execution that reduces the benefit of predicated execution.

Several experimental models with various levels of predication support are used in this thesis. All of the models in this thesis are based on a generic ILP processor (either VLIW or superscalar) with in-order issue and register interlocking. A generic load/store ISA is assumed.

In addition, the baseline architecture is assumed to have support for speculative execution in the form of non-excepting, or silent, versions for each instruction. The next two sections will present the instruction set and micro-architecture extensions used to support predicated execution by the Full Predication and Conditional Move Predication models.

2.2 Extensions for the Full Predication Model

The experimental model with the most extensive support for predication in this thesis will be referred to as the *Full Predication* (FP) model. The FP instruction set includes significant extensions to the base ISA in an effort to support predicated execution with maximum flexibility and efficiency. The extensions to the instruction set for predication support are those of the HP PlayDoh Architecture [11]. The first extension is that all instructions are predicated. In addition to each instruction having a predicate operand, the FP set includes a number of instructions for defining predicates. It contains a predicate definition instruction for each comparison opcode in the base instruction set. The format of a predicate definition instruction is shown below.

$$\text{pred\_}\!<\!cmp\!>\ \text{Pout1}_{<type>},\ \text{Pout2}_{<type>},\ \text{src1},\ \text{src2}\ (\text{P}_{in})$$

The comparison, <*cmp*>, can be *ge*, *lt*, *eq*, etc. The predicate definition instruction has up to two predicate destinations, *Pout1* and *Pout2*. Predicate definition instructions can also be predicated as represented by *Pin* in the instruction format. The predicate <*type*> represents the rules that are used to assign the predicate value based on the comparison of *src1* and *src2* and the value of the incoming predicate, *Pin*. There are six predicate types in the FP instruction set. These types are *unconditional (U)*, *OR*, and *AND* along with their complements. Table 2.1 shows the rules for assigning predicate values associated with these six types. An unconditional

Table 2.1 Predicate definition truth table.

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

type predicate is always assigned a value regardless of the value of the incoming predicate, *Pin*. An unconditional type predicate is assigned a 1 iff the incoming predicate, *Pin*, is true and the result of the comparison is true. Otherwise it is assigned a 0. An OR-type predicate is assigned a 1 iff the incoming predicate is true and the result of the comparison is true. Otherwise it remains unchanged. Since OR-type predicate definitions can only set a predicate to 1, OR-type predicates must be initialized to 0 before they are defined and used. An important characteristic of the semantics for the OR-type predicate definitions is that multiple OR-type predicate definitions with the same destination register can be executed simultaneously since the order of definition does not matter.

The FP instruction set also includes a *pred_set* instruction and a *pred_clear* instruction. The *pred_set* instruction simultaneously sets all predicates to 1 and the *pred_clear* simultaneously clears all predicates to 0.

A number of micro-architecture features are assumed for the FP architecture. First, all of the predicates are stored in a predicate register file of 1-bit registers. Second, the convert-to-NOP method is used for suppression of execution for an instruction with a false predicate. Instructions with false predicates are converted to NOPs in the decode/issue stage of the pipeline before they are issued. The next stage is the execute stage. This stage completely executes predicate definition instructions. Register bypassing logic sends the new predicate definitions

```
if (a&&b)
   j = j + 1;
else
   if (c)
      k = k + 1;
   else
      k = k − 1;
i = i + 1;
```

(a)

```
   beq   a,0,L1
   beq   b,0,L1
   add   j,j,1
   jump  L3
L1:
   bne   c,0,L2
   add   k,k,1
   jump  L3
L2:
   sub   k,k,1
L3:
   add   i,i,1
```

(b)

```
pred_clear
pred_eq  p1$_{OR}$,p2$_{\overline{U}}$,a,0
pred_eq  p1$_{OR}$,p3$_{\overline{U}}$,b,0 (p2)
add      j,j,1 (p3)
pred_ne  p4$_{U}$,p5$_{\overline{U}}$,c,0 (p1)
add      k,k,1 (p4)
sub      k,k,1 (p5)
add      i,i,1
```

(c)

Figure 2.1 Example of predication, (a) source code, (b) assembly code, (c) assembly code after if-conversion.

to the decode/issue stage in time for the suppression of instructions with false predicates. Therefore, the dependence distance between the definition of a predicate and its use is just one cycle.

Figure 2.1 contains a simple code segment in three representations: source code (a), base instruction set (b), and FP instruction set (c).

2.3 Extensions for Partial Predication

The experimental model with the least support for predication in this thesis will be referred to as the *Conditional Move Predication* (CMP) model. The CMP instruction set has only two additional instructions over the base instruction set. The first additional instruction is the *conditional move*, or *cmov*, instruction. The semantics for this instruction are shown below. The *cmov* instruction is very similar to a predicated move instruction. The main difference is that the predicate of a cmov instruction is a value held in the least significant bit of an integer register. The integer register which holds the predicate is denoted *cond*. If the least significant bit of *cond* is a 1 (predicate = true), then value in *src* is copied to *dest*. Otherwise, *dest* is unmodified.

cmov dest,src,cond

if (cond) dest = src

The second added instruction is the *complemented cmov*, or $\overline{cmov}$, instruction. This instruction is very similar to the *cmov* instruction. The semantics are shown below. The difference between the *cmov* instruction and the $\overline{cmov}$ is that for the latter case, the least significant bit of *cond* is considered to hold the complement of the predicate. In other words, a 0 in the least significant bit of *cond* causes the value in *src* to be copied to *dest*.

$\overline{\textbf{cmov}}$ **dest,src,cond**

if (!cond) dest = src

As with the predicated instructions in the FP architecture, the dependence distance between the definition of the *cmov* condition, *cond*, and the use of the condition by a *cmov* instruction is 1. Similarly, *cmov* instructions with false predicates are nullified in the issue/decode stage.

The minor ISA extensions required for CMP support provide a natural way for existing ISAs to incorporate partial predication. The two source operands and one destination operand fit well into current 3 operand ISAs. In fact, as mentioned in Section 1.2.1, a number of instruction sets already provide similar instructions.

CHAPTER 3

COMPILER SUPPORT

3.1 Background

Compiler support for predicated execution involves techniques to convert code from one level of predication to another. Techniques that convert unpredicated code to predicated code are generally based on a concept called if-conversion. The term *if-conversion* is generally used to mean the conversion of control dependence to data dependence through the use of Boolean instruction guards. If-conversion was first introduced by Allen et al. [13]. They presented some techniques to eliminate some control flow in loop bodies by converting it to explicit data flow which computed explicit Boolean expressions for when each instruction should execute. The execution of these instructions was guarded by an *if* statement based on the corresponding Boolean expression. Park and Schlansker [14] presented an if-conversion technique which converts low level unpredicated code to low level predicated code. Their technique eliminates all control flow within the region of if-conversion. They present efficient semantics for predicated execution and show that based upon those semantics, their if-conversion technique has the following characteristics:

1. It minimizes the number of predicates in use and thus the number of predicate defining operations.
2. It can be applied to any flow graph (including cyclic and irreducible).
3. The time complexity, $O(n^2)$, is dominated by computation of control dependences.

Mahlke et al. [15] introduced a comprehensive set of techniques for generating efficient fully predicated code from unpredicated code. These techniques will be discussed in detail in Section 3.2. Warter and Hwu [16] introduced a technique called *reverse if-conversion* for converting predicated code into unpredicated code. This technique allowed unpredicated code to be converted to predicated code and back. The reason for this process is that some optimizations are possible in predicated code that are not possible in unpredicated code.

Th next two sections present the compiler support of predicated execution used by the FP and CMP models. For both models, the code is compiled from the source code to the generic baseline instruction set. Then, Mahlke's hyperblock techniques are applied to transform the code into the FP instruction set. The FP code is fully optimized. Then, for the CMP model, new compilation techniques are used to convert the FP code into CMP code. This compilation path is a new method of compiling for architectures with limited predication support such as a conditional move instruction. It has the advantage of being able to apply a more sophisticated conceptual framework to exposing ILP and optimizing the code. Section 3.2 discusses the hyperblock techniques, and Section 3.3 presents the techniques for converting FP code into CMP code. The final section in this chapter contains two examples which illustrate some of the characteristics of this compiler support.

3.2 Compiler Support for the Full Predication Model

The compiler support for the FP model is based on the hyperblock techniques in [15]. These hyperblock techniques are motivated by two problems with conventional compiler support for predicated execution. First, conventional if-conversion combines all paths of control regardless of their execution frequency and size. This approach results in a large number of useless instructions being executed. Second, speculative execution is difficult to combine with predicated execution. Hyperblock techniques solve these problems by very carefully choosing and forming regions on which to perform if-conversion. A *hyperblock* is defined to be a collection of connected basic blocks in which control may only enter from the top, but may exit from one or more locations. All control flow between the basic blocks in a hyperblock is eliminated through if-conversion. Heuristics are used to choose which paths to include in the hyperblock. These heuristics attempt to balance between two occasionally conflicting priorities. The first priority is to include only the frequently executed paths. The second priority is to avoid saturating the processor resources. The result is a structure called a hyperblock which will allow simultaneous optimization of the most important paths. This technique tends to eliminate branches which are not heavily biased in either direction. In addition, compiler-controlled speculation is made much easier by the fact that hyperblocks do not have any side entrances.

Hyperblock formation is done in the following three steps:

1. Use tail duplication and loop peeling to remove all side entrances from basic blocks which are to be formed into a hyperblock.
2. Apply node splitting to eliminated dependences created by control path merges.
3. Use if-conversion to replace the set of selected basic blocks with a hyperblock.

These hyperblock techniques are discussed more thoroughly in [15].

3.3 Compiler Support for Partial Predication

The compiler support for the CMP model consists of a set of conversions and optimizations that transforms a program that uses the FP instruction set to an equivalent program that uses the CMP instruction set. The two general categories of instructions from the FP set which must be converted are *predicated* instructions and *predicate definition* instructions. The predicate definition instructions are basically complex logical instructions. They are converted into several standard logic instructions. The general approach for converting predicated instructions is to perform the operation speculatively, putting the result in a temporary register, and then using a conditional move to copy the result into the original destination. The conversions for both of these types of instructions have three important features. First of all, since many instructions are speculated, the efficiency of these conversions requires some architectural support for speculative execution. Second, each instruction conversion increases the number of instructions. Third, each instruction conversion increases the dependence height associated with the original instruction from one to two. The transformation from fully predicated code to cmov predicated code occurs in three general steps: predicate promotion, basic conversions, and peephole optimizations.

3.3.1 Predicate promotion

Predicate promotion is the process of removing the data dependence between a predicate definition instruction and a predicated instruction. Removing such a dependence allows the predicated instruction to be moved ahead of the predicate definition. This process of predicate promotion and code reordering is equivalent to breaking the control dependence between a branch and a subsequent instruction by moving that instruction ahead of the branch. Both processes result in speculative execution of the previously dependent instruction. Speculative execution can be used to effectively reduce the dependence height along important paths. This advantage of speculative execution comes at a cost, one of which is the execution of some useless instructions. Another cost of speculative execution is that it occasionally requires the addition of move instructions to accomplish register renaming. Speculation can also affect the performance of the instruction and data caches.

Speculation is employed in the FP code through predicate promotion. The tradeoffs are slightly different for predicate promotion than for control speculation because predicated instructions that are nullified also waste execution resources. For the FP code, heuristics are used to balance the tradeoffs involved. Predicate promotion is even more beneficial for CMP code than for FP code. The reason is that predicate promotion often completely eliminates the predicate on an instruction. Because, in general, each conversion of a predicated instruction into an equivalent representation in the CMP instruction set increases the number of instructions, removal of predicates through predicate promotion can increase the efficiency of the resulting CMP code. Also, the cost of occasionally inserting register renaming moves is not a factor in CMP code because such a move would have been inserted in the conversion from FP code to CMP.

The benefits of predicate promotion for CMP code are illustrated in Figure 3.1. The top left quadrant shows FP code before predicate promotion. The top right quadrant shows the CMP code that would result from conversion of the individual predicated instructions. These conversions will be discussed more fully in the next section. Applying predicate promotion to the code in the top left quadrant produces the code in the bottom left quadrant. The code in the bottom right quadrant then shows the CMP code that would result from conversion of the

	fully predicated code	**partially predicated code**
before promotion	load temp1,addrx,offx (Pin) mul temp2,temp1,2 (Pin) add y,temp2,3 (Pin)	load temp3,addrx,offx cmov temp1,temp3,Pin mul temp4,temp1,2 cmov temp2,temp4,Pin add temp5,temp2,3 cmov y,temp5,Pin
after promotion	load temp1,addrx,offx mul temp2,temp1,2 add y,temp,2,3 (Pin)	load temp1,addr,offx mul temp2,temp1,2 add temp3,temp2,3 cmov y,temp3,Pin

operation: load x
y = 2x+3

Note: non-excepting instructions assumed.

Figure 3.1 Example of predicate promotion.

instructions in the bottom left quadrant. In this example, applying predicate promotion before the conversion from FP code to CMP code reduces the number of instructions in the CMP code from six to four. Because of its great benefit for CMP code, additional predicate promotion is performed before the conversion from FP code to CMP code.

3.3.2 Basic conversions

After predicate promotion, the compiler uses a set of transformations, referred to as *basic conversions*, to map individual instructions from the FP instruction set to the CMP instruction set. The basic conversions must map two general categories of instructions which exist in the FP instruction set but not in the CMP instruction set: *predicated* instructions and *predicate* definition instructions. The semantics of a predicated instruction requires that, in the presence of a true predicate, the program produce the same results as if the instruction were not predicated. In the presence of a false predicate, the instruction should have the same effect on the results as if it did not exist. An instruction may affect the results of a computation in two ways.

First, an instruction may produce a value which is subsequently used by other instructions to affect the final result of the program. Second an instruction may cause a program terminating exception such as divide-by-zero, illegal memory address, overflow, or underflow. The basic conversions must insure that the CMP code will follow these intended semantics. Careful consideration must also be given to the semantics of predicate definition in both the FP and CMP architectures when creating these basic conversions. A predicate register in the FP architecture is a 1-bit register. In the CMP architecture, the least significant bit of integer register is used to hold a 1-bit predicate. Although the semantics of the *cmov* instruction depend only on the least significant bit of the integer register holding the predicate, the basic conversions will maintain the convention of using an integer 0 for a false predicate and an integer 1 for a true predicate. The peephole optimization section will show that relaxing this convention can provide some optimization opportunities.

The basic conversions are considerably simplified if the underlying processor has full support for speculative execution. In particular, if the processor has non-excepting, or silent, versions of each instruction, the basic conversions can be much more efficient. The basic conversions can be more efficient in this case because they do not have to generate tests to avoid unwanted exceptions. Figure 3.2 shows some representative examples of the basic conversions in this case. In these examples, the registers named *Pin* and *Pout* are predicate registers in the FP code (on the left) and integer registers in the CMP code (on the right). At this point in the compilation process these registers are still virtual registers, and so they are simply renamed.

Example conversions for two predicate definition instructions are shown in Figure 3.2. The first predicate definition is an unconditional *(U)*, type predicate definition with an incoming predicate, *Pin*, based on a less-than comparison. If the incoming predicate, *Pin*, is 1 and the

Fully Predicated Code		Basic Conversions, Non-excepting Instructions	
predicate definition instructions			
pred_lt	Pout_{U}, src1, src2 (Pin)	lt and	rtemp, src1, src2 Pout, Pin, rtemp
pred_lt	Pout_{OR}, src1, src2 (Pin)	lt and or	rtemp, src1, src2 rtemp, Pin, rtemp Pout, Pout, temp
arithmetic & logic instructions			
add	dest, src1, src2 (Pin)	add cmov	rtemp, src1, src2 dest, rtemp, Pin
div	dest, src1, src2 (Pin)	div cmov	rtemp, src1, src2 dest, rtemp, Pin
memory instructions			
store	addr, off, src (Pin)	sub cmov store	rtemp, $safe, off rtemp, addr, Pin rtemp, off, src
load	dest, addr, off (Pin)	load cmov	rtemp, addr, off dest, rtemp, Pin
branch instructions			
jump	label (Pin)	bne	Pin, 0, label
blt	src1, src2, label (Pin)	ge blt	rtemp, src1, src2 rtemp, Pin, label
jsr	label (Pin)	beq jsr NEXT:	Pin, 0, NEXT label

Figure 3.2 Basic conversions assuming non-excepting instructions available.

comparison is true, this instruction puts a 1 in *Pout*; otherwise, it puts a 0 in *Pout*. In the CMP code, the less-than comparison is performed first, placing the result (1 for true, 0 for false) into a temporary register, *rtemp*. The incoming predicate, *Pin*, is then *and*-ed with the result of the comparison, and the result is placed in *Pout*. The second predicate definition example is that of an *OR*-type predicate. The difference between an *OR*-type and a *U*-type predicate definition is that in the cases in which the *U*-type definition writes a 0 into *Pout*, the *OR*-type definition does not write anything into *Pout* leaving the original value of *Pout* unchanged. The equivalent CMP code for the *OR*-type predicate definition is similar to that for the *U*-type definition. A comparison and an *and* instruction are used to produce the same 0 or 1 result, and then this result is *or*-ed with the previous value of the predicate to produce the final value of the predicate. Thus, a predicate value of 1 cannot be changed to a 0 by this code. Predicate definition instructions can have two destinations. In such a case, the instruction is converted as if it were two separate predicate definition instructions. The basic conversion for a predicate definition instruction in the FP instruction set can produce from one to six instructions in the CMP instruction set.

The most efficient and straightforward basic conversions are for predicated instructions with register destinations as shown by the predicated *add* conversion in Figure 3.2. In the CMP code, the *add* is executed speculatively with the result going into a temporary register, *rtemp*. A *cmov* instruction is then used to move the result from the temporary register, *rtemp*, to the original destination, *dest*, based on the value of *Pin*, the original predicate.

The basic conversions for instructions with memory destinations are somewhat more complicated. Because the CMP instruction set does not have a conditional store instruction, the store must occur regardless of the value of the original predicate. Instead, the approach taken is to conditionally modify the address of the store instruction so that the memory destination is the original memory destination in the case of a true predicate and a different unused memory location in the case of a false predicate. The predicated *store* conversion in Figure 3.2 gives an example of such a conversion. In the CMP code, the offset, *off*, is subtracted from an unused, safe address, *safe_addr*, so that *safe_addr - off* is placed in *rtemp1*. The purpose of doing this subtraction is that now if the value in *rtemp1* is used as the base address of the *store*, the resulting memory address will be *safe_addr*. The next instruction is a *cmov* which overwrites the value in *rtemp1* with the original base address, *addr*, if the original predicate, *Pin*, is true. The resulting value in *rtemp1* is then used as the base address of an unconditionally executed store.

Example conversions for predicated branch instructions are also shown in Figure 3.2. A predicated unconditional branch is converted into a conditional branch that tests the value of the predicate, *Pin*, and branches when the predicate is not 0. A predicated conditional branch is implemented in two instructions. The first instruction is a comparison which is opposite to that of the comparison in the original conditional branch. This comparison places a 0 in *rtemp* if the original comparison would have been true. The branch should be taken if and only if the predicate is true and the original comparison would have been true. Equivalently, the branch should be take if and only if *Pin* is 1 and *rtemp* is 0. Since *Pin* and *rtemp* can only take on the values 1 and 0, the branch should be taken if and only if *rtemp* is less than *Pin*. Therefore, the last instruction in the CMP code is a conditional branch which is taken when *rtemp* is less than *Pin*. Predicated subroutine calls cannot be handled directly in the CMP set, so the basic conversion for predicated subroutine calls resorts to reverse-if conversion. An example is shown in Figure 3.3. The branches produced by this conversion are the only ones that occur in the

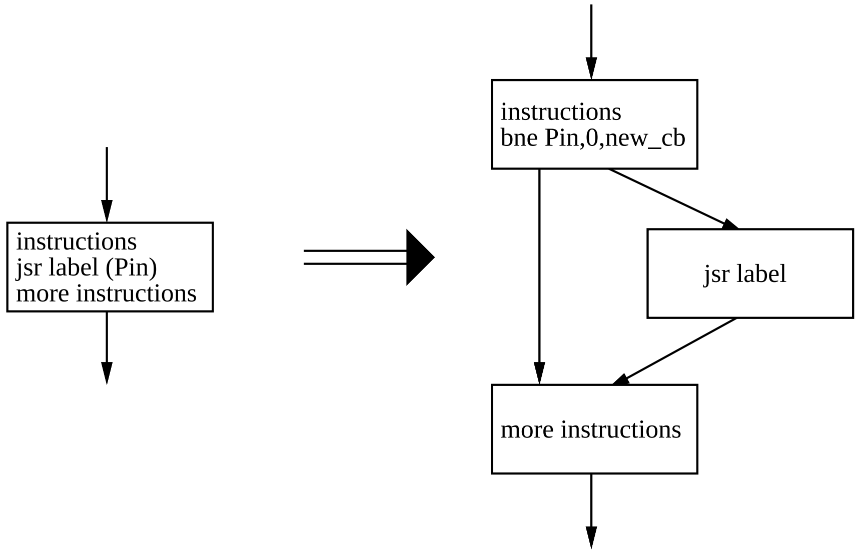


Figure 3.3 Basic conversion for a jsr instruction.

CMP code but not in the FP code.

All of the conversions in Figure 3.2 depend on the existence of non-excepting or silent versions of all of the instructions. Conversions are also possible in the case where these silent instructions do not exist. The conversions in this case, however, have the additional overhead of making sure that speculated instructions do not cause exceptions which could alter the result of the program. Figure 3.4 shows some examples of the basic conversions when non-excepting versions of the instructions are not available. Predicate floating-point instructions can be converted if a floating-point conditional move exists. (Otherwise, branches will have to be inserted through reverse if-conversion.) The conversions for predicated floating-point instructions must produce code that will prevent an overflow or underflow exception when the original predicate is false. This is done by conditionally moving a value, *safe_val*, into one of the operands so that the instruction is guaranteed not to produce an exception. Similarly, the conversions for predicated divides must use a safe value to avoid a divide-by-zero exception.

Fully Predicated Code		Basic Conversions, Non-excepting Instructions	
predicate definition instructions			
pred_lt_f	Pout_U, src1, src2 (Pin)	mov	rtemp1, $safe_val
		cmov_f	rtemp1, src2, Pin
		lt_f	rtemp2, src1, rtemp1
		and	Pout, Pin, rtemp2
arithmetic & logic instructions			
div_f	dest, src1, src2 (Pin)	mov	rtemp1, $safe_val
		cmov_f	rtemp1, src1, Pin
		div_f	rtemp2, src1, rtemp1
		cmov	dest, rtemp2, Pin
memory instructions			
load	dest, addr, off (Pin)	sub	rtemp1, $safe_addr, off
		cmov	rtemp1, addr, Pin
		load	rtemp2, rtemp1, off
		cmov	dest, rtemp2, Pin

Figure 3.4 Basic conversion without non-excepting instructions.

Illegal memory address exceptions must also be prevented in the conversions for predicated loads.

3.3.3 Optimizing the CMOV code

The basic conversions in the previous section focused on converting individual instructions from the FP instruction set to the CMP instruction set. The conversions could be more efficient if they considered more than one instruction at a time. The peephole optimizations presented in this section attempt to use a more global understanding of the basic conversions to increase the efficiency of the CMP code. The instructions generated by the basic conversions are marked as they are generated, since certain assumptions can be made about these instructions which may not necessarily be made in general. The original FP code is already highly optimized, so the peephole optimizations need only focus on eliminating inefficiencies in the basic conversions. Full optimization of the code resulting from the predicate definition conversions requires the ability to complement predicates. Complementing predicates in the CMP instruction set is tricky, so this issue is discussed prior to presenting the peephole optimizations.

3.3.3.1 Complementing predicates

Predicates fundamentally only hold one bit of information. However, in the CMP instruction set, the predicates are stored in integer registers. The comparison instructions in the base instruction set which are used in defining predicates assign a 1 or 0 to their integer destination registers. It would be possible to define a false predicate as the integer 0 and a true predicate as the integer 1, but then bitwise logical inversion would no longer complement the predicate. Fully optimizing the predicate definitions requires the ability to complement predicates with logical instructions. Therefore, predicates are defined to be the value stored in the least significant bit of the integer predicate register. With this definition, bitwise logical inversion will now complement the predicate.

The basic conversions, however, operate under the more restrictive assumption that all bits other than the least significant bit are 0's (i.e., the value in an integer predicate register can be only 0 or 1). This restriction is not a difficulty for the basic conversions because the need to complement the predicate never arises. In reality, only the basic conversions for predicated conditional and unconditional branches rely on this assumption (see Figure 3.2). The instructions resulting from these conversions depend on the integer predicate register containing precisely a 1 or 0. In order to allow predicates to be freely complemented, the instructions which rely on the predicate register containing precisely a 0 or 1 are preceded by an *and* instruction which masks off all but the lowest bit of the integer predicate register. This masking allows predicates to be freely complemented with bitwise logical instructions and thus facilitates the optimization of the predicate definition code. Adding these masks turns out to be mostly for conceptual purposes and ease of implementation, since most of these mask instructions can be removed after optimization of the predicate definition instructions.

3.3.3.2 Peephole optimizations

Classic Optimizations. The FP code often contains multiple predicate definitions which are very similar. The basic conversions handle each predicate definition individually, and as a result, they frequently introduce redundant comparison and logic instructions. Some of these redundancies can be eliminated using common subexpression elimination, copy propagation, and dead code removal. Figure 3.2 gives such a case. The example has two predicate definitions which are identical except for predicate type. In this example, the basic conversions produce two compare instructions, two *and* instructions, and an *or* instruction. The compares and *and* would be redundant.

Redundant Complementary Comparison Elimination. It turns out that there is only one type of redundancy produced by the basic conversions for predicate definitions which cannot be removed by classical common subexpression elimination. If the type of one of the predicate definitions in Figure 3.2 were complemented, the two comparisons on the right would be complementary. These comparisons would usually be redundant. If all uses of the result of one of these comparison instructions can be complemented, then one of the comparisons can be eliminated. A comparison instruction can be complemented if each use of its result can be complemented. *Cmov* instructions and logical instructions are the only two types of instructions that use results of comparisons produced by the basic conversions. The predicate input of a *cmov* instruction can be complemented by changing the *cmov* to a $\overline{cmov}$ instruction. This use of the $\overline{cmov}$ is necessary for full optimization of the predicate definitions and is the main benefit of having a $\overline{cmov}$ instruction in the CMP instruction set. The base instruction set has a full set of bitwise logic instructions including: *and*, *or*, *nand*, *nor*, *and_compl*, and *or_compl*. (Compl means that the bitwise complement is taken of the second input before it is used.) These are the only logic instructions which used the outputs of the comparisons being discussed. The inputs to any of these logical instructions can be individually complemented by using a different instruction in this set. Therefore, the comparisons produced by the basic conversion of predicate definition instructions can always be complemented. An optimization called *redundant complementary comparison elimination* searches for these redundancies and removes them.

Predicate Definition Height Reduction. In the FP instruction set, the execution semantics of OR-type predicate definitions is very efficient. A sequence of OR-type prediction definition instructions which all write the the same destination can be executed in one cycle. However, such a sequence of instructions results in a sequential chain of dependent instructions in the cmov predicated instruction set. The dependence height of this sequential chain of instructions can be reduced by applying associativity rules to the sequential chain of instructions. In this way, a chain of dependent instructions is converted into a binary tree of instructions, and the dependence height is reduced from n to $log_2(n)$. This optimization is called *predicate definition height reduction*, and an example is given in Section 3.4.

Branch Predicate Mask Elimination. After optimization of the predicate definition instructions, almost all of the mask instructions which were introduced to allow complementation of predicates can be removed. The mask instructions were inserted in front of code which expected the value in a predicate register to be exactly a 0 or 1. If the compiler can determine the values of the upper bits of an integer predicate register, then the corresponding mask can be eliminated and the code can be modified, if necessary, to check for a new value.

The values of the upper bits of an integer predicate register can be determined at compile time. In addition, these bits will either be all 1's (*negative polarity*) or all 0's (*positive polarity*). The basic conversions produce code in which the value in an integer predicate register is always initialized to either a 1 or 0. Therefore, the polarity of the values in all predicate registers is originally positive. After initialization, the value in a predicate register is modified only by bitwise logical combination with the values in other predicate registers and bitwise complementation. If the polarity of a value in an integer predicate register is know, then the polarity of its bitwise complement is also know. Similarly, if the polarities of two predicate registers are know, the polarity of their logical combination is know.

The *predicate definition graph* is a structure which enables the polarity of these upper bits to be determined. This graph is a binary tree in which the nodes represent instructions and the arcs represent data flow dependences. The root of the tree is the previous definition of the predicate register of interest. The children of a node are the previous definitions of each each source operand. The polarity of the leaf nodes is always known, and the polarity resulting from any logical operation of registers with known polarities can always be determined. Therefore, the polarity of any predicate register can be determined by building the predicate definition graph and traversing it in depth-first order, determining the polarity of each node along the way. This method only works for predicate registers which are completely defined in the same block in which the predicate register is used. Most of the time this is the case. Through the use of the *predicate definition graph*, most of the mask instructions can be eliminated. This optimization is called *branch predicate mask elimination*.

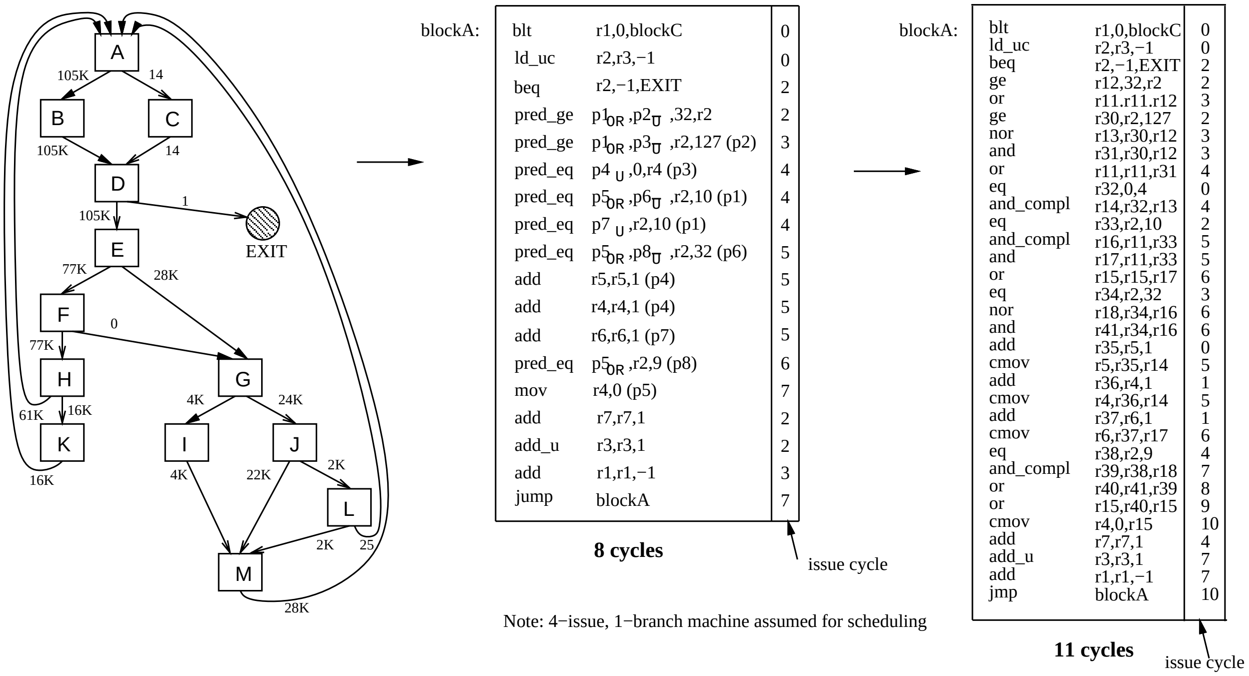
3.4 Benchmark Examples

To more clearly understand the effectiveness of predicated execution support and the performance tradeoffs of FP vs. CMP support, two examples from the set of benchmarks are presented. The first example is from *wc* and the second is from *grep*. These benchmarks were chosen because they are relatively small, yet control-intensive, so they clearly illustrate the effectiveness of FP and CMP support.

3.4.1 Example loop from wc

Figure 3.5(a) shows the control flow graph for the most important loop segment from the benchmark *wc*. The control flow graph is augmented with the execution frequencies of each control transfer for the measured run of the program. This loop is characterized by small basic blocks and a large percentage of branches. The loop segment contains 13 basic blocks with a total of 34 instructions, 14 of which are branches. The performance of an 8-issue ILP processor without predicated execution support is limited by this high frequency of branches. Overall, a speedup of 2.0 is achieved for an 8-issue processor over a 1-issue processor (See Figure 4.1).

The assembly code after hyperblock formation for the loop segment is shown (b) and (c) of Figure 3.5 for the FP and CMP models, respectively. The issue cycle is given to the right of each assembly code instruction. Note that the assembly code is not reordered based on the issue cycle for ease of understanding. The schedule assumes a 4-issue processor which can issue 4 instructions of any type except branches, which are limited to 1 per cycle. With both FP and



```
blockA:  blt       r1,0,blockC                0
         ld_uc     r2,r3,−1                   0
         beq       r2,−1,EXIT                 2
         pred_ge   p1_OR ,p2_Ū ,32,r2         2
         pred_ge   p1_OR ,p3_Ū ,r2,127 (p2)   3
         pred_eq   p4 _U ,0,r4 (p3)           4
         pred_eq   p5_OR ,p6_Ū ,r2,10 (p1)    4
         pred_eq   p7 _U ,r2,10 (p1)          4
         pred_eq   p5_OR ,p8_Ū ,r2,32 (p6)    5
         add       r5,r5,1 (p4)               5
         add       r4,r4,1 (p4)               5
         add       r6,r6,1 (p7)               5
         pred_eq   p5_OR ,r2,9 (p8)           6
         mov       r4,0 (p5)                  7
         add       r7,r7,1                    2
         add_u     r3,r3,1                    2
         add       r1,r1,−1                   3
         jump      blockA                     7
```

8 cycles

issue cycle

Note: 4−issue, 1−branch machine assumed for scheduling

```
blockA:  blt          r1,0,blockC    0
         ld_uc        r2,r3,−1       0
         beq          r2,−1,EXIT     2
         ge           r12,32,r2      2
         or           r11.r11.r12    3
         ge           r30,r2,127     2
         nor          r13,r30,r12    3
         and          r31,r30,r12    3
         or           r11,r11,r31    4
         eq           r32,0,4        0
         and_compl    r14,r32,r13    4
         eq           r33,r2,10      2
         and_compl    r16,r11,r33    5
         and          r17,r11,r33    5
         or           r15,r15,r17    6
         eq           r34,r2,32      3
         nor          r18,r34,r16    6
         and          r41,r34,r16    6
         add          r35,r5,1       0
         cmov         r5,r35,r14     5
         add          r36,r4,1       1
         cmov         r4,r36,r14     5
         add          r37,r6,1       1
         cmov         r6,r37,r17     6
         eq           r38,r2,9       4
         and_compl    r39,r38,r18    7
         or           r40,r41,r39    8
         or           r15,r40,r15    9
         cmov         r4,0,r15       10
         add          r7,r7,1        4
         add_u        r3,r3,1        7
         add          r1,r1,−1       7
         jmp          blockA         10
```

11 cycles

issue cycle

(a) basic block control flow block

(b) fully predicated hyperblock code

(c) partially predicated code

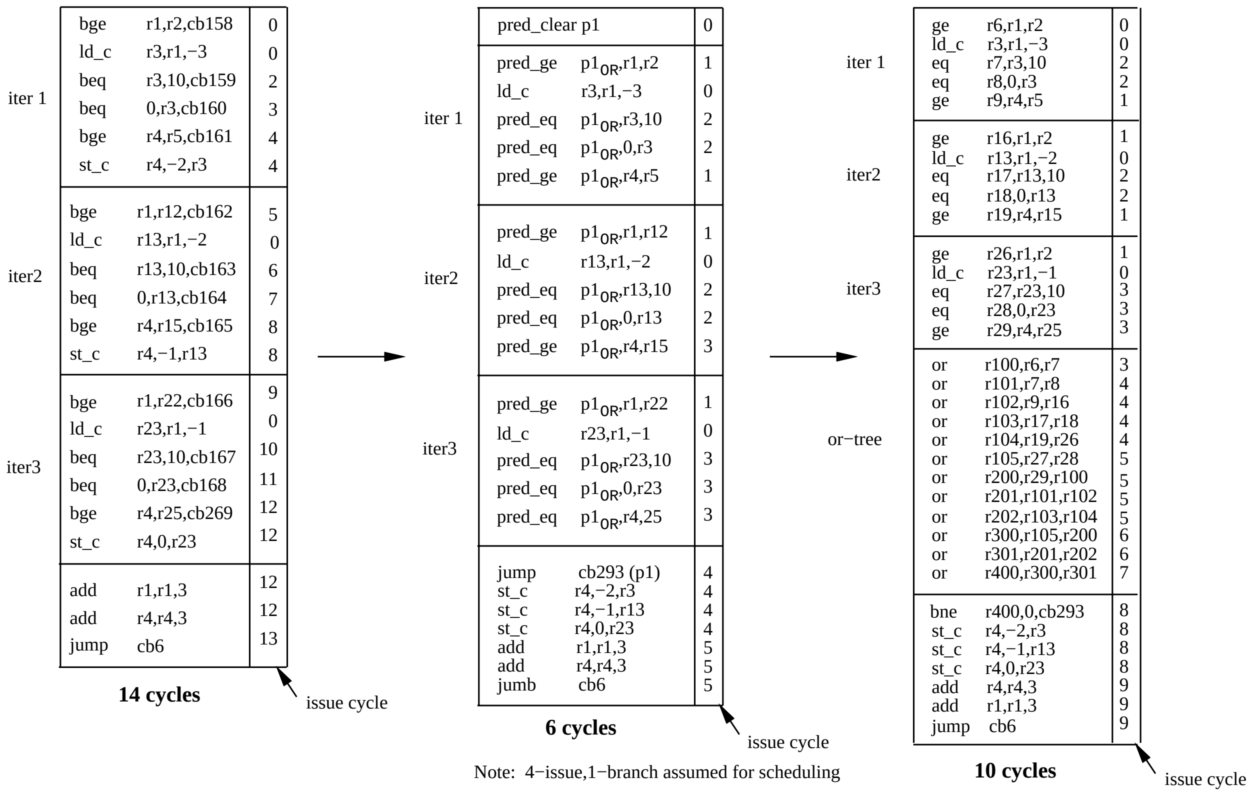
Figure 3.5 Example loop segment from *wc*.

CMP support, all of the branches except three are eliminated using hyperblock formation. The three remaining branches (conditional branch to block C, conditional branch to EXIT, and the loop backedge) are highly predictable. Therefore, virtually all the mispredictions are eliminated with both FP and CMP support in this loop. The resulting performance is increased by 27% with CMP support and 120% with FP support. (See Figure 4.1).

As with many benchmarks, the number of instructions increases significantly going from FP code to CMP code. The FP code in Figure 3.5(b) consists of 18 instructions, and the CMP code in Figure 3.5(c) consists of 33 instructions. In some cases, especially with lower issue widths, this increase in the number of instructions saturates the processor issue resources and increases the execution time. The execution time does increase in this example from 8 cycles to 11 cycles, but the reason for this is a little more subtle. The increase in execution time is a result of an increase in dependence height. The height of the longest dependence chain in the CMP code is 11 cycles. In both the FP and CMP code, the longest dependence chain contains several predicate definitions. These predicate definitions are one instruction each and take one cycle in the FP code. In addition, the dependence distance between two OR-type predicate definition instructions with the same destination register is 0. In other words, two such predicate definition instructions can be executed in the same cycle. In the CMP code, each predicate definition requires from 1 to 3 sequential instructions, and the OR-type predicate definitions do not have any special semantics which allow them to executed more efficiently. The greater effective dependence height of predicate definitions increases the height of the longest dependence chain in the CMP code and thus increases the execution time. For the entire benchmark execution, a similar trend is observed, and the speedup increases from 2.5 for the CMP model to 4.4 for the FP model. (See Figures 4.1 p. 35 and 4.7 p. 41).

3.4.2 Example loop from grep

Figure 3.6 shows the assembly code for the most important loop segment from the benchmark *grep*. The base processor model, which does not support any predicated execution, employs speculative execution in conjunction with superblock ILP compilation techniques to achieve the schedule shown in Figure 3.6(a) [6]. Each of the conditional branches in the figure are infrequently taken, thus the sequence of instructions iterates very frequently. Overall, *grep* is dominated by an extremely high frequency of branches. This high frequency of branches is the performance bottleneck of this loop since only one branch resource is available. However, the branches are highly predictable. Thus, hyperblock compilation techniques focus on reducing this branch bottleneck for processors with limited branch resources.



```
iter 1   bge   r1,r2,cb158    0
         ld_c  r3,r1,−3       0
         beq   r3,10,cb159    2
         beq   0,r3,cb160     3
         bge   r4,r5,cb161    4
         st_c  r4,−2,r3       4

iter2    bge   r1,r12,cb162   5
         ld_c  r13,r1,−2      0
         beq   r13,10,cb163   6
         beq   0,r13,cb164    7
         bge   r4,r15,cb165   8
         st_c  r4,−1,r13      8

iter3    bge   r1,r22,cb166   9
         ld_c  r23,r1,−1      0
         beq   r23,10,cb167   10
         beq   0,r23,cb168    11
         bge   r4,r25,cb269   12
         st_c  r4,0,r23       12

         add   r1,r1,3        12
         add   r4,r4,3        12
         jump  cb6            13
```

```
         pred_clear p1              0

iter 1   pred_ge   p1_OR,r1,r2      1
         ld_c      r3,r1,−3         0
         pred_eq   p1_OR,r3,10      2
         pred_eq   p1_OR,0,r3       2
         pred_ge   p1_OR,r4,r5      1

iter2    pred_ge   p1_OR,r1,r12     1
         ld_c      r13,r1,−2        0
         pred_eq   p1_OR,r13,10     2
         pred_eq   p1_OR,0,r13      2
         pred_ge   p1_OR,r4,r15     3

iter3    pred_ge   p1_OR,r1,r22     1
         ld_c      r23,r1,−1        0
         pred_eq   p1_OR,r23,10     3
         pred_eq   p1_OR,0,r23      3
         pred_eq   p1_OR,r4,25      3

         jump      cb293 (p1)       4
         st_c      r4,−2,r3         4
         st_c      r4,−1,r13        4
         st_c      r4,0,r23         4
         add       r1,r1,3          5
         add       r4,r4,3          5
         jumb      cb6              5
```

Note: 4−issue,1−branch assumed for scheduling

```
iter 1   ge    r6,r1,r2          0
         ld_c  r3,r1,−3          0
         eq    r7,r3,10          2
         eq    r8,0,r3           2
         ge    r9,r4,r5          1

iter2    ge    r16,r1,r2         1
         ld_c  r13,r1,−2         0
         eq    r17,r13,10        2
         eq    r18,0,r13         2
         ge    r19,r4,r15        1

iter3    ge    r26,r1,r2         1
         ld_c  r23,r1,−1         0
         eq    r27,r23,10        3
         eq    r28,0,r23         3
         ge    r29,r4,r25        3

or−tree  or    r100,r6,r7        3
         or    r101,r7,r8        4
         or    r102,r9,r16       4
         or    r103,r17,r18      4
         or    r104,r19,r26      4
         or    r105,r27,r28      5
         or    r200,r29,r100     5
         or    r201,r101,r102    5
         or    r202,r103,r104    5
         or    r300,r105,r200    6
         or    r301,r201,r202    6
         or    r400,r300,r301    7

         bne   r400,0,cb293      8
         st_c  r4,−2,r3          8
         st_c  r4,−1,r13         8
         st_c  r4,0,r23          8
         add   r4,r4,3           9
         add   r1,r1,3           9
         jump  cb6               9
```

(a) superblock code **(b) fully predicated hyperblock code** **(c) partially predicated code**

Figure 3.6 Example loop segment from *grep*.

With FP support, the compiler is able to combine the branches into a single exit branch using OR-type predicate definitions. Since OR-type predicate definitions can be issued simultaneously, an extremely tight schedule can be achieved. The execution time is dramatically reduced from 14 to 6 cycles with full predicate support. With CMP support, the same transformations are applied. Therefore, the same number of branches is eliminated. However, the representation of OR-type predicates is less efficient with CMP support. In particular, the logical *or* instructions cannot be simultaneously issued. The *predicate definition height reduction* discussed previously in Section 3.3.3.2 is applied to reduce the dependence height of the sequence and improve performance. In the example, CMP support improves performance from

14 to 10 cycles. Overall for the final benchmark performance, CMP support and FP support improve performance by 47% and 93% respectively, over the base code.

CHAPTER 4

EXPERIMENTAL EVALUATION

4.1 Methodology

The techniques discussed in this thesis are evaluated using emulation-driven simulation. The benchmarks studied are 023.eqntott, 026.compress, 052.alvinn, 056.ear, and 072.sc from the SPEC92 suite and the UNIX utilities cccp, cmp, eqn, grep, lex, qsort, tbl, wc, and yacc. The benchmarks are compiled into the generic instruction set with varying levels of support for predication (i.e., the baseline instruction set, the FP instruction set, and the CMP instruction set). Register allocation and scheduling are performed, producing code which could be run on an architecture with the corresponding instruction set. Then the code is prepared for emulation on a HP PA-RISC processor by running a second phase of register allocation and generating PA-RISC assembly code. The emulation of the varying levels of predicate support is done using the bit manipulation and conditional nullification capabilities of the PA-RISC instruction set [8]. Predicates in the FP model are emulated by reserving n of the callee-saved registers and accessing them as 32 x n 1-bit registers. The emulation of the code verifies correct execution and generates an instruction trace for the simulator. The simulator uses the trace information generated by the emulator in addition to detailed modeling of the prefetch and issue unit, the instruction and data caches, the branch target buffer, and hardware interlocks in order to provide accurate performance information. We call this technique *emulation-driven simulation*.

Three processor models are evaluated in the next section. The baseline processor, referred to as *Superblock*, is a k-issue processor. It can issue any mix of k instructions, with the exception that it can only issue one branch per cycle. The instruction set for this processor is a generic load/store ISA. The instruction and data caches are either *perfect* or *16K* direct mapped caches with 32 byte blocks. The data cache is a blocking cache with a write through, no write allocate policy and a miss penalty of 12 cycles. For dynamic branch prediction, the baseline processor has a 1K entry BTB with a 2-bit counter predictor and a misprediction penalty of 2 cycles. The baseline processor also has the option of a 10-cycle misprediction penalty, referred to as *large_mp*, or a *two-level* branch prediction scheme. This processor is assumed to have an unlimited number of registers. It does not have any support for predicated execution, but it does have support for speculative execution in the form of non-excepting or silent versions of all instructions. The code compiled for this processor is obtained using Superblock ILP compilation techniques.

The processor model with the most extensive support for predicate is referred to as the *Full Prediction* (FP) model. The FP processor has, in addition to the features of the baseline processor, the extensive architectural support for predicated execution described in Section 2.2. The code compiled for this processor is obtained using the Hyperblock compilation techniques discussed in Section 3.2.

The third processor model is referred to as the *Conditional Move Predication* (CMP) model. The CMP processor has in addition to the features of the baseline processor predicated execution support in the form of the *cmov* and $\overline{cmov}$ instructions as discussed in Section 2.3. The code compiled for this processor is converted from the code for the FP processor using the basic conversions and optimizations of Section 3.3.

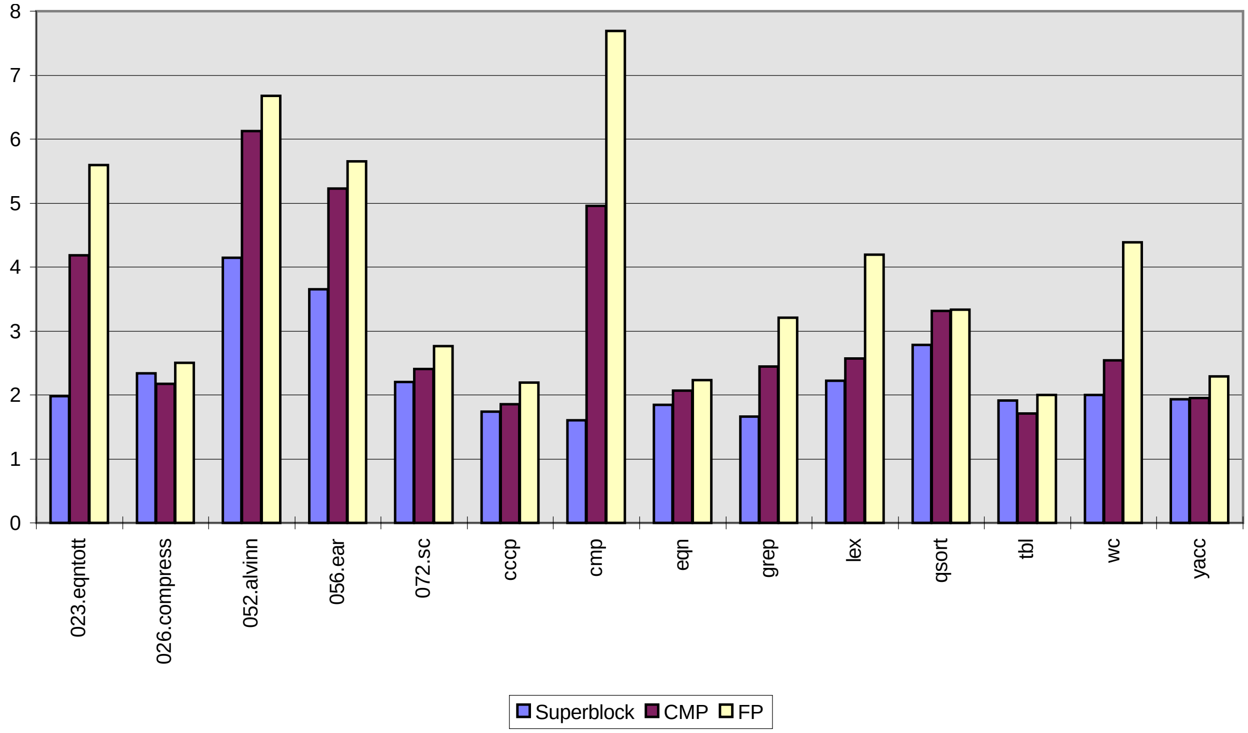


Figure 4.1 Effectiveness of FP and CMP support for an 8-issue, 1-branch processor.

Performance information is provided for these models in terms of speedup, which is calculated by dividing the cycle count for a 1-issue baseline processor by the cycle count of a k-issue processor of the specified model.

4.2 Results

4.2.1 Benefits of predicated execution

Figure 4.1 shows the performance results on an 8-issue, 1-branch architecture for each level of predication support. For the results in this figure, perfect caches, a 2-bit counter branch predictor, and a 2-cycle branch misprediction penalty are assumed. The speedup for the CMP model is on average[1] 38% higher than the speedup for the Superblock model, and the speedup

[1] Averages reported refer to the arithmetic mean. When average percentages are reported, the percentage is computed for each benchmark before the arithmetic mean is taken.

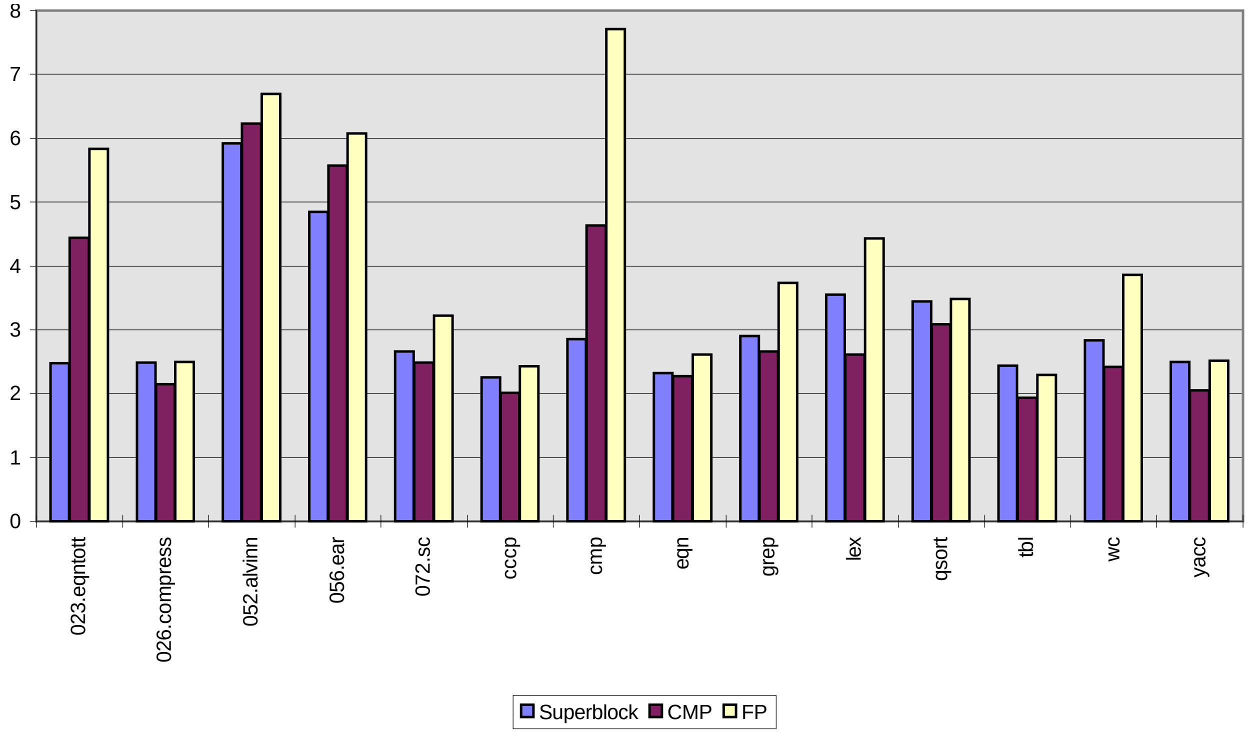


Figure 4.2 Effectiveness of FP and CMP support with 2-branch issue capability.

for the FP model is on average 79% higher than the speedup for the Superblock model. Clearly, support for predicated execution has some major performance benefits. All of the other figures in this section will vary by one parameter from Figure 4.1.

Some of the benefits of predicated execution are discussed in Section 1.1.3. One of these benefits is the ability to eliminate branches from the dynamic instruction stream. Since many processors are issue-limited by branches (as is the processor in Figure 4.1), eliminating branches from the dynamic instruction stream can significantly increase the issue rate. Figure 4.2 demonstrates this with performance results for an 8-issue, 2-branch architecture. CMP and FP have speedups 2% and 34% greater than Superblock, respectively. Addition of a second branch issue capability helped Superblock model narrow the performance gap with the predication models. Although all of the models actually perform better with the 2-branch issue capability, the per-

Table 4.1 Comparison of branch statistics: number of branches (BR), mispredictions (MP), and misprediction rate (MPR).

Benchmark	Superblock			Conditional Move			Predicated Move			Full Predication		
	BR	MP	MPR	BR	MP	MPR	BR	MP	MPR	BR	MP	MPR
023.eqntott	313M	43M	13.85%	55M	6818K	12.33%	54M	7139K	13.27%	50M	6919K	13.81%
026.compress	12M	1283K	10.67%	9333K	873K	9.36%	9272K	863K	9.31%	9229K	846K	9.17%
052.alvinn	461M	1075K	0.23%	75M	1122K	1.49%	74M	1050K	1.41%	74M	1050K	1.41%
056.ear	1540M	65M	4.19%	446M	15M	3.34%	442M	15M	3.36%	443M	16M	3.64%
072.sc	22M	1193K	5.39%	11M	896K	8.29%	12M	861K	7.36%	12M	878K	7.42%
cccp	921K	65K	7.09%	538K	66K	12.27%	535K	65K	12.20%	538K	65K	12.15%
cmp	530K	4395	0.83%	27K	31	0.12%	27K	31	0.12%	27K	31	0.12%
eqn	7707K	574K	7.45%	4535K	530K	11.70%	4498K	500K	11.12%	4535K	536K	11.82%
grep	663K	9660	1.46%	171K	20K	11.73%	171K	20K	11.73%	171K	20K	11.73%
lex	14M	235K	1.65%	3091K	209K	6.76%	2996K	186K	6.21%	2994K	193K	6.46%
qsort	8781K	1357K	15.45%	6059K	617K	10.18%	6073K	614K	10.11%	6129K	609K	9.93%
tbl	610K	38K	6.31%	432K	39K	9.12%	417K	38K	9.09%	417K	38K	9.02%
wc	478K	33K	6.85%	224K	57	0.03%	223K	57	0.03%	224K	57	0.03%
yacc	12M	523K	4.47%	5811K	436K	7.50%	5896K	427K	7.24%	6065K	435K	7.18%

formance gap between the Superblock model and the predication models narrowed because the Superblock code has many more branches. As shown in Table 4.1, the hyperblock formation techniques have removed many branches from the CMP and FP code.

Another benefit of predicated execution is the ability to reduce the number of branch mispredictions. Table 4.1 shows the number of branches, the number of mispredictions, and the misprediction rate for each model in Figure 4.1. The number of branches and mispredictions is significantly lower for the two models with predication support than it is for the Superblock model. The misprediction rate is usually somewhat higher for the CMP and FP models because more dynamic branches than mispredictions are removed for these models.

Figure 4.3 shows that the effectiveness of predicated execution increases as the misprediction penalty increases. The misprediction penalty in Figure 4.3 is 10 cycles while the misprediction penalty in Figure 4.1 is only 2 cycles. For this larger misprediction penalty, the speedups for the CMP and FP models average 51% and 92% better than the speedup for the Superblock model. The main reason for this significant widening in the performance gap between the Superblock

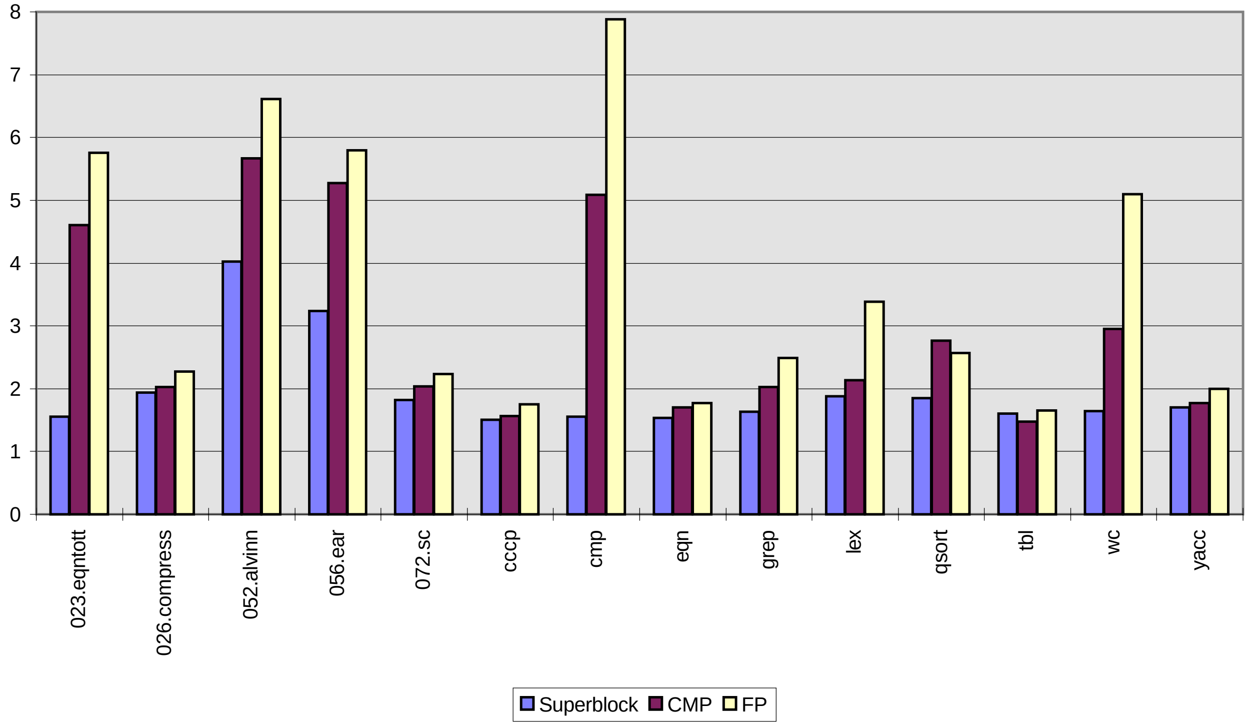


Figure 4.3 Effectiveness of FP and CMP support with 10-cycle branch misprediction penalty.

model and the models with predication support is that the Superblock model has to do many more branch predictions. As a result, its performance dropped much more with an increase in the branch misprediction penalty than did the others. As processor frequencies continue to rise, the level of pipelining tends to increase, usually with the effect of increasing the branch misprediction penalty. For instance, Intel's Pentium Pro processor has a branch misprediction penalty of 10 or more cycles. Some, but not all, of the problem of a large branch misprediction penalty can be alleviated by using a better branch predictor. On the other hand, very high quality branch prediction requires considerable hardware and tends to reinforce the limit of one branch per cycle.

Figure 4.4 shows performance results for an architecture with a two-level branch predictor. The better predictor does slightly narrow the performance gap between the Superblock model

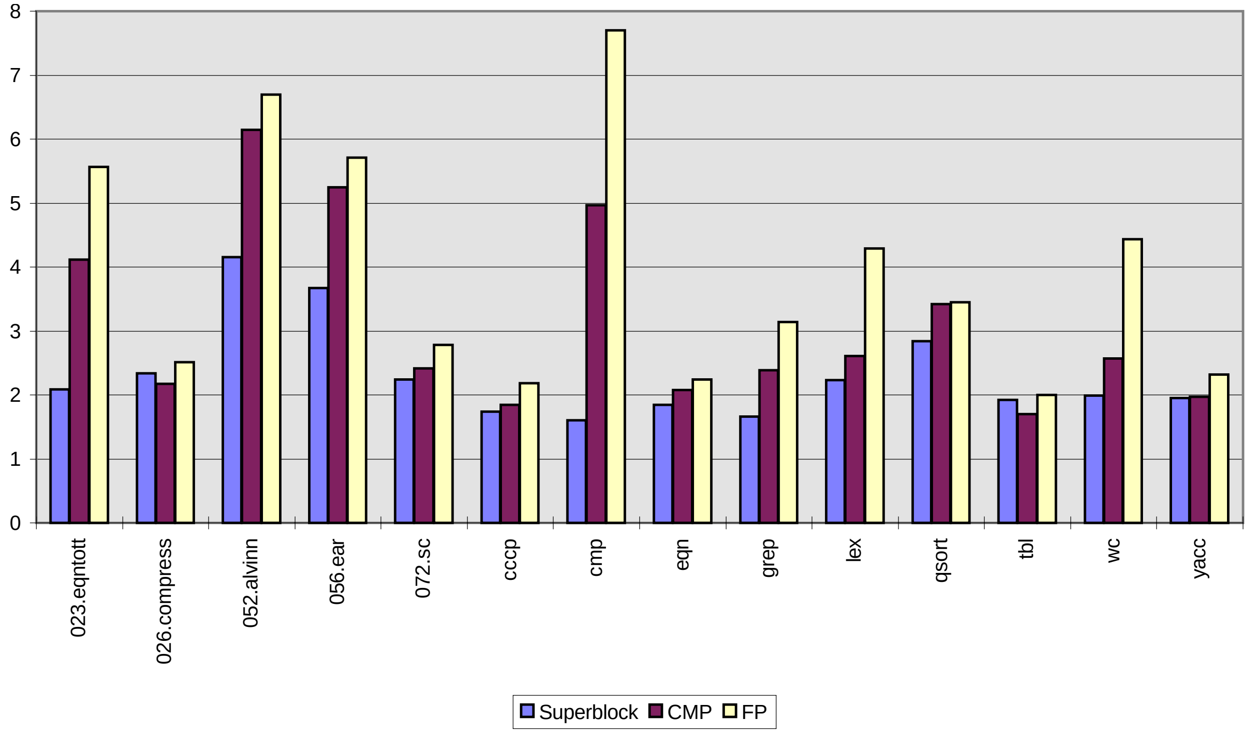


Figure 4.4 Effectiveness of FP and CMP support with better (two-level) branch predictor.

and the two predication models. The speedups for the CMP and FP models are on average 37% and 78% better, respectively, than the speedup for the Superblock model in this case.

Figure 4.5 shows the effect using 16k (instead of perfect) instruction and data caches. In general, smaller caches hurt the performance of the CMP and FP models a little more than the Superblock model. This is mainly due to the larger number of instructions, the more extensive speculation, and shorter schedules of the code for the predication models. On average, the CMP and FP models perform 30% and 59% better than the Superblock model does.

Figure 4.6 shows that the FP model retains a significant advantage at lower issue rates. This figure will be discussed in more detail in the next section.

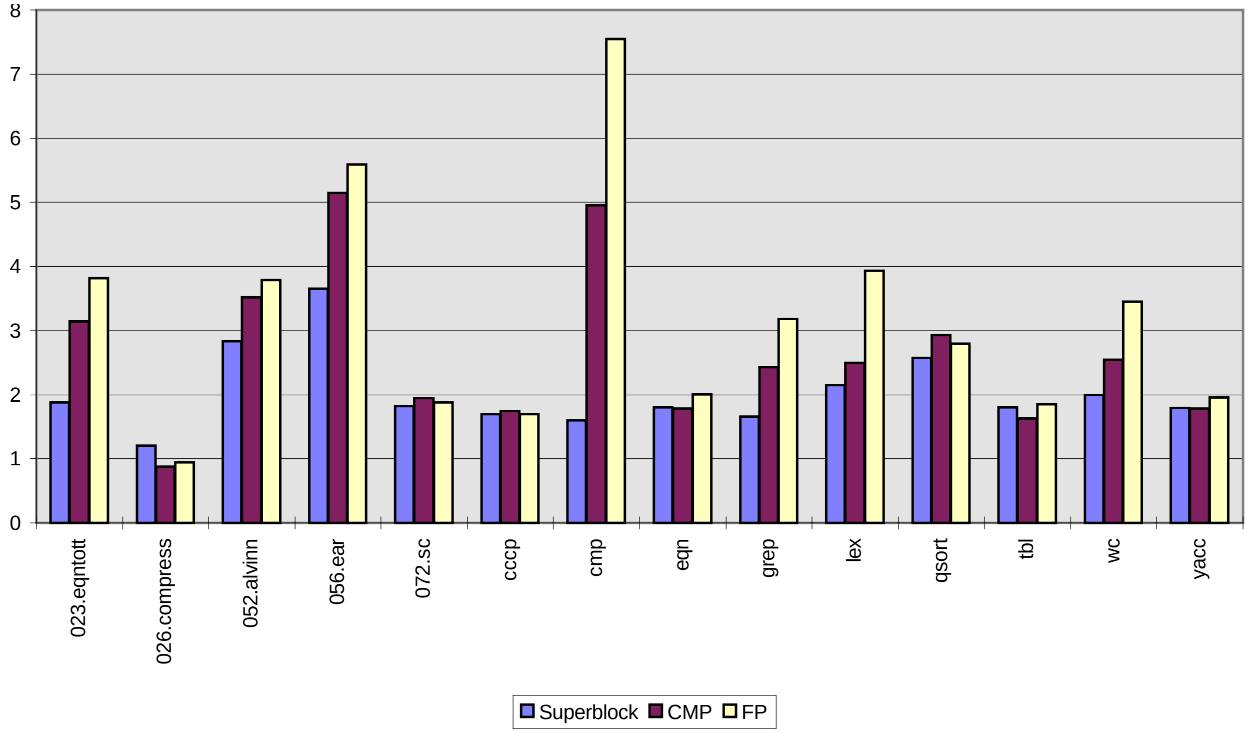


Figure 4.5 Effectiveness of FP and CMP support with 16K instruction and data caches.

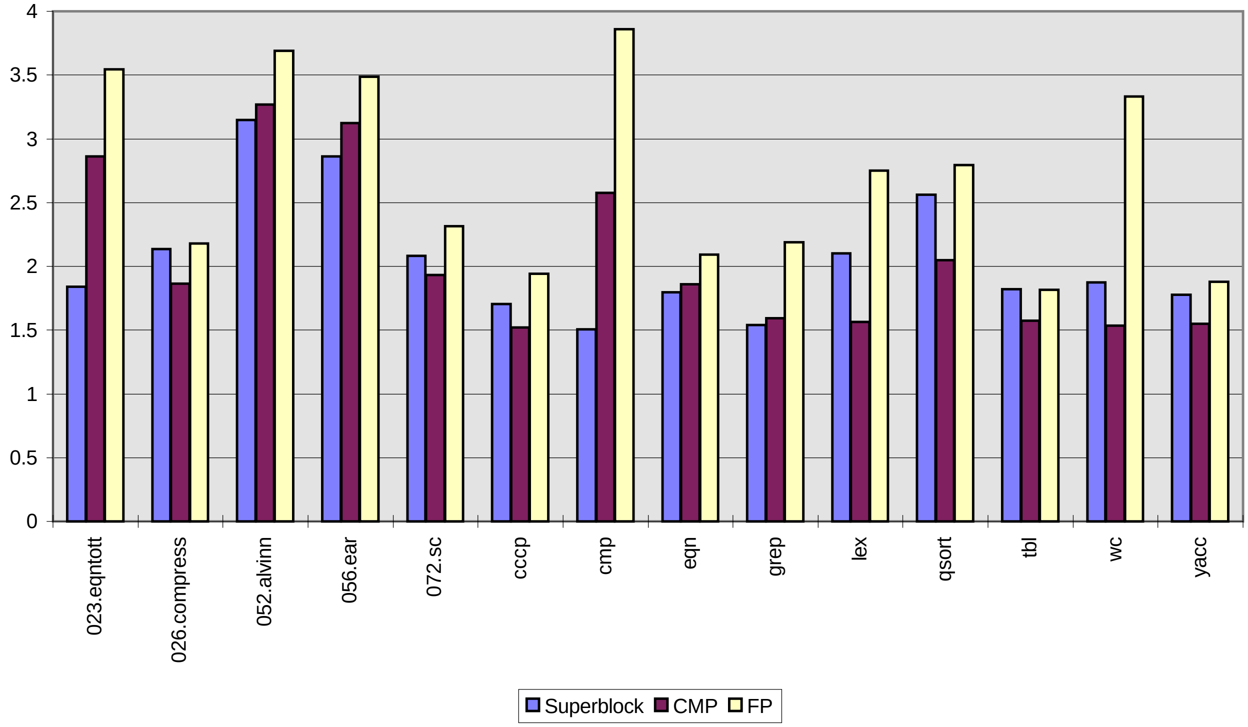


Figure 4.6 Effectiveness of FP and CMP support with only 4-issue capability.

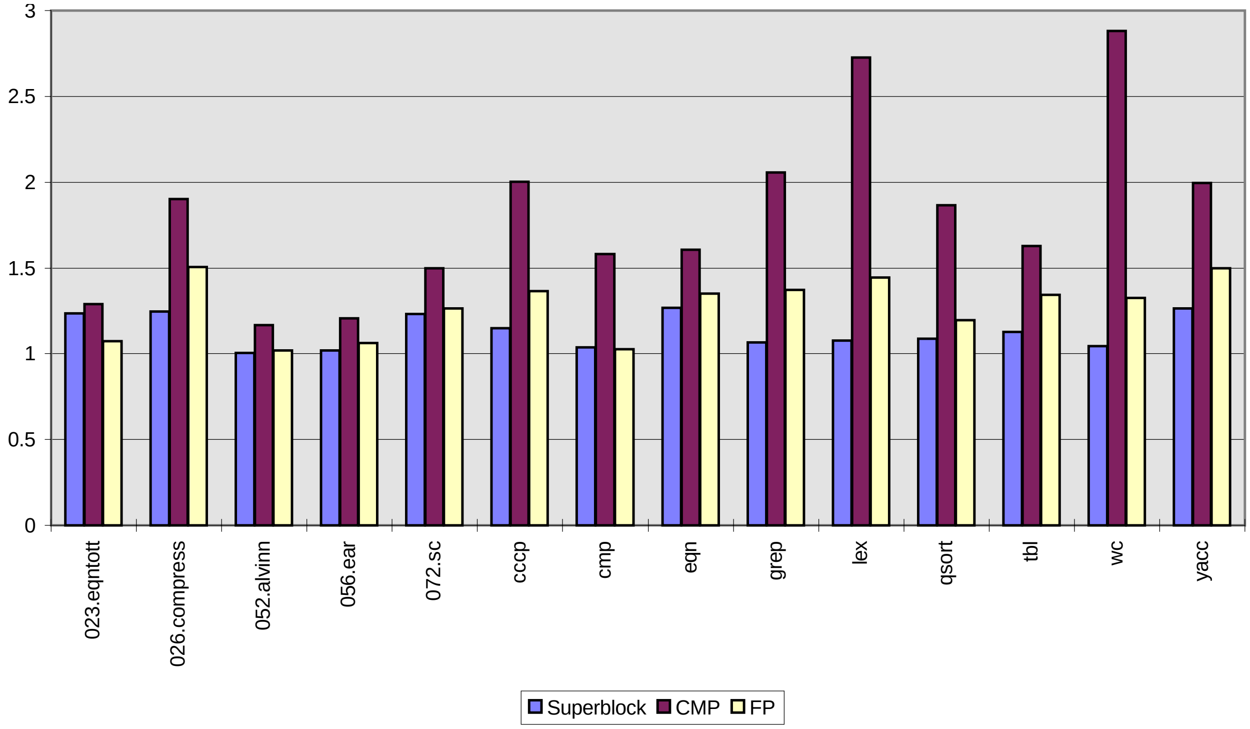


Figure 4.7 Dynamic instruction count comparison.

4.2.2 Conditional move vs. full predication

Figure 4.1 shows that the FP model has the best performance of the three and that the CMP model has, on average, performance about halfway between that of the FP and Superblock models. The difference in performance between the two models with predication support is a result of two factors. The two factors are the increase in dynamic instruction count and the increase in dependence height that result from the basic conversions.

Figure 3.2 p. 19 shows that the basic conversion for both predicated instructions and predicate definition instructions increase the number of instructions. Figure 4.7 compares the number of dynamic instructions executed by each model. (Note that the numbers have been normalized so that the number of instructions executed by the baseline processor is one.) This figure shows a number of interesting things. All of the models execute more instructions than the baseline processor. An increase in the number of dynamic instructions is usually one of the tradeoffs for exposing ILP. The FP model executes on average 13% more instructions than the Superblock model. The CMP model executes on average 62% more instructions than the Superblock model. For lex and wc, the CMP model actually executes more than twice as many instructions as the Superblock model. Such a large increase in the number of instructions executed can saturate the issue capabilities of even an 8-issue machine. The fact that the CMP model achieves only about half of the speedup of the FP model is partially due to larger numbers of instructions in the CMP model. Decreasing the issue rate magnifies this effect. Figure 4.6 shows that for a lower issue rate, the number of instructions in the CMP code saturates the issue slots and significantly decreases the performance for the CMP code. For the 4-issue machine, the CMP model performs worse than the Superblock model on most of the benchmarks. Since the CMP model performs quite a bit better on a few benchmarks, it averages 1.8% more speedup than the Superblock model. The FP model does not have the same problems. The performance of the FP model does decrease significantly going from an 8-issue to a 4-issue architecture. However, the FP model does not have to execute nearly as many instructions as the CMP model, and its advantage over the Superblock still averages 24% for the 4-issue architecture. The CMP model would benefit in this case from a less aggressive hyperblock formation algorithm. The basic conversions start with hyperblocks which were targeted at the FP model and which are therefore more aggressive than is desirable for the CMP model in this case.

The second factor in the performance difference between the two predication models is the increase in dependence chain lengths in the CMP code due to the basic conversions. In addition to increasing the number of instructions, the basic conversions for both predicated instructions

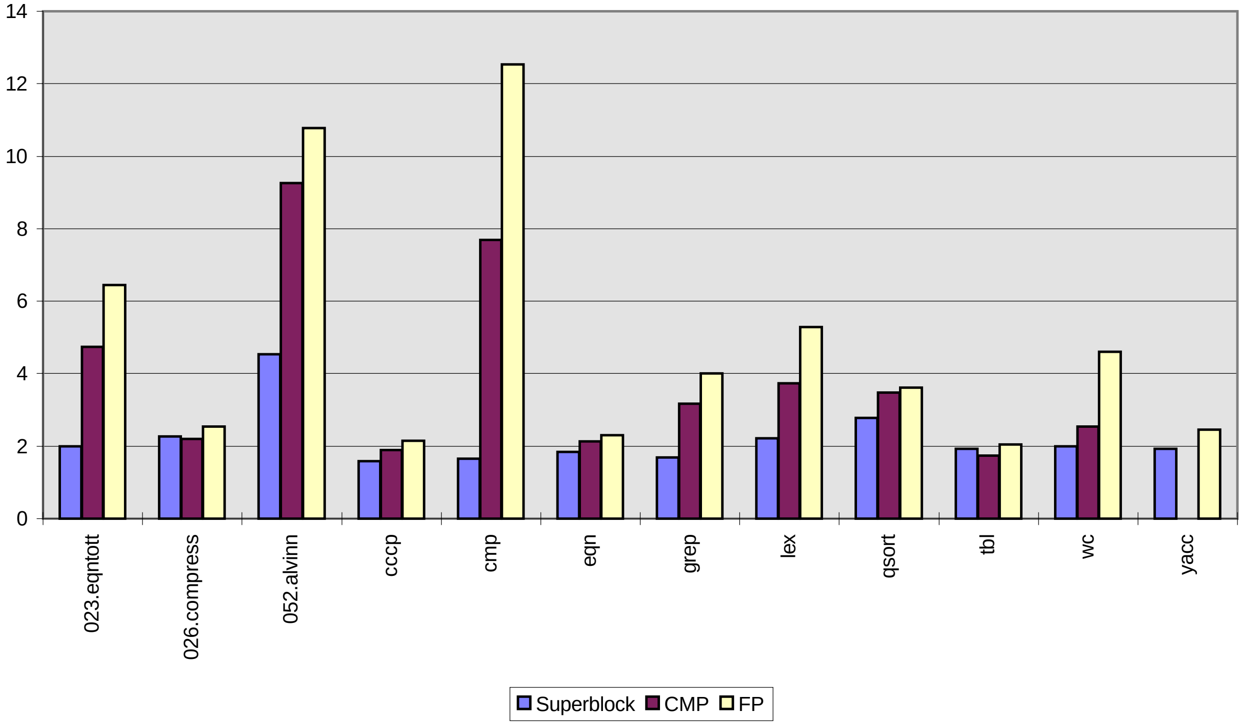


Figure 4.8 Effectiveness of FP and CMP support with 16-issue capability.

and predicate definition instructions increase dependence heights of the paths containing the instruction being converted. This effect is especially critical for predicate definition instructions since they are so prevalent and efficient in the FP code. The performance results for a machine with a significantly larger issue rate make it apparent that much of the performance difference between the CMP and FP models is due to dependence height. Figure 4.8 shows the results for a 16-issue, 1-branch processor. The performance advantage still maintained by the FP model over the CMP model attests to the significance of the dependence height increase.

CHAPTER 5

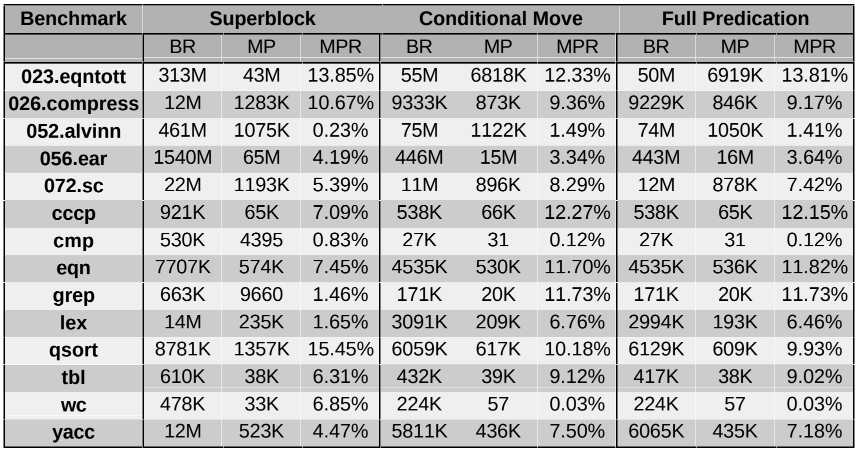
AN INTERMEDIATE DESIGN POINT

5.1 Motivation

As shown in Chapter 4, FP support has some significant performance advantages over CMP support. The CMP model also has another difficulty which was not addressed in Chapter 4. In those experiments, the models were assumed to have an infinite number of registers. The CMP model uses integer registers to store predicates and thus will increase register pressure significantly. On the other hand, the CMP architectural extensions have the advantage of being much easier to fit into an existing ISA. A natural question would be: With the knowledge we now have, can we find a level of predication support which is easier than FP support is to fit into an existing ISA but results in higher performance than does CMP support?

The predication support in the FP instruction set has two major components. One component is the extensive set of complex predicate definition instructions. Both the number and complexity of these instructions make them difficult to fit into an existing instruction set. However, a significant performance penalty is paid in the CMP model for not supporting these predicate definition instructions. Up to five instructions are required to define a predicate in the CMP instruction set. In addition, the instructions corresponding to predicate definitions in the CMP instruction set are more sequential than the original predicate definitions. The number of predicate definition instructions in the fully predicated code corresponds roughly to the number of branches eliminated by if-conversion (see Table 5.1). Therefore, the set of

Table 5.1 Comparison of branch statistics: number of branches (BR), mispredictions (MP), and misprediction rate (MPR).



Benchmark	Superblock			Conditional Move			Full Predication		
	BR	MP	MPR	BR	MP	MPR	BR	MP	MPR
023.eqntott	313M	43M	13.85%	55M	6818K	12.33%	50M	6919K	13.81%
026.compress	12M	1283K	10.67%	9333K	873K	9.36%	9229K	846K	9.17%
052.alvinn	461M	1075K	0.23%	75M	1122K	1.49%	74M	1050K	1.41%
056.ear	1540M	65M	4.19%	446M	15M	3.34%	443M	16M	3.64%
072.sc	22M	1193K	5.39%	11M	896K	8.29%	12M	878K	7.42%
cccp	921K	65K	7.09%	538K	66K	12.27%	538K	65K	12.15%
cmp	530K	4395	0.83%	27K	31	0.12%	27K	31	0.12%
eqn	7707K	574K	7.45%	4535K	530K	11.70%	4535K	536K	11.82%
grep	663K	9660	1.46%	171K	20K	11.73%	171K	20K	11.73%
lex	14M	235K	1.65%	3091K	209K	6.76%	2994K	193K	6.46%
qsort	8781K	1357K	15.45%	6059K	617K	10.18%	6129K	609K	9.93%
tbl	610K	38K	6.31%	432K	39K	9.12%	417K	38K	9.02%
wc	478K	33K	6.85%	224K	57	0.03%	224K	57	0.03%
yacc	12M	523K	4.47%	5811K	436K	7.50%	6065K	435K	7.18%

predicate definition instructions plays a very important role in the performance of the FP code. Another component in the FP instruction set is the extra predicate operand that each instruction has. Fitting an extra operand on each instruction in an existing instruction set is a big problem. Most instructions simply do not have enough unused bits in the instruction word to accommodate another operand. The performance cost of not supporting an extra predicate operand on each instruction is the increase in the number of instructions and the height of some dependence chains resulting from the basic conversions of predicated instructions.

The full predication architecture also has a micro-architectural feature of great benefit, the predicate register file. This prevents the predicates from increasing the register pressure in the integer register file. The predicate register file is also a more efficient use of space, since each predicate register has to be only one bit wide. Mahlke et al. [15] uses 16 to 32 predicate registers.

5.2 Architectural Extensions and Compiler Support

The model described in this section, called the Predicated Move Predication (PMP) model, will be used to help clarify the components of the performance gap between the FP model and the CMP model. The features were also chosen in an attempt to maximize the performance of the model while minimizing the impact on existing instruction sets. Since predicate definitions are so frequent in predicated code, I concluded that support of predicated definition instructions is a key factor in the performance gap between the CMP and FP models. Similarly, I concluded that keeping predicates in integer registers will cause a significant increase in register pressure that will counteract some of the speedup for the CMP model. I thought that performance benefits of supporting for an extra predicate operand on each instruction were not worth the difficulty of providing this support. Therefore, the PMP instruction set supports all of the predicate definition instructions in the FP instruction set but supports only one predicated operation, the *predicated move*, or *pmov*. It turns out that the compilation techniques presented in Section 3.3 are useful as a basis for converting from the FP instruction set to a variety of instruction sets with less support for predication. A subset of these compilation techniques is used to convert FP code to PMP code.

5.3 Experimental Evaluation

Figure 5.1 shows that for an 8-issue, 1-branch processor, the PMP mode achieves most of the speedup of the FP model. On average, the FP, PMP, and CMP models achieve respective speedups of 79%, 68%, and 38% greater that the Superblock model. The CMP model achieves roughly half of the speedup of the FP model, while the PMP model achieves about 85% of the

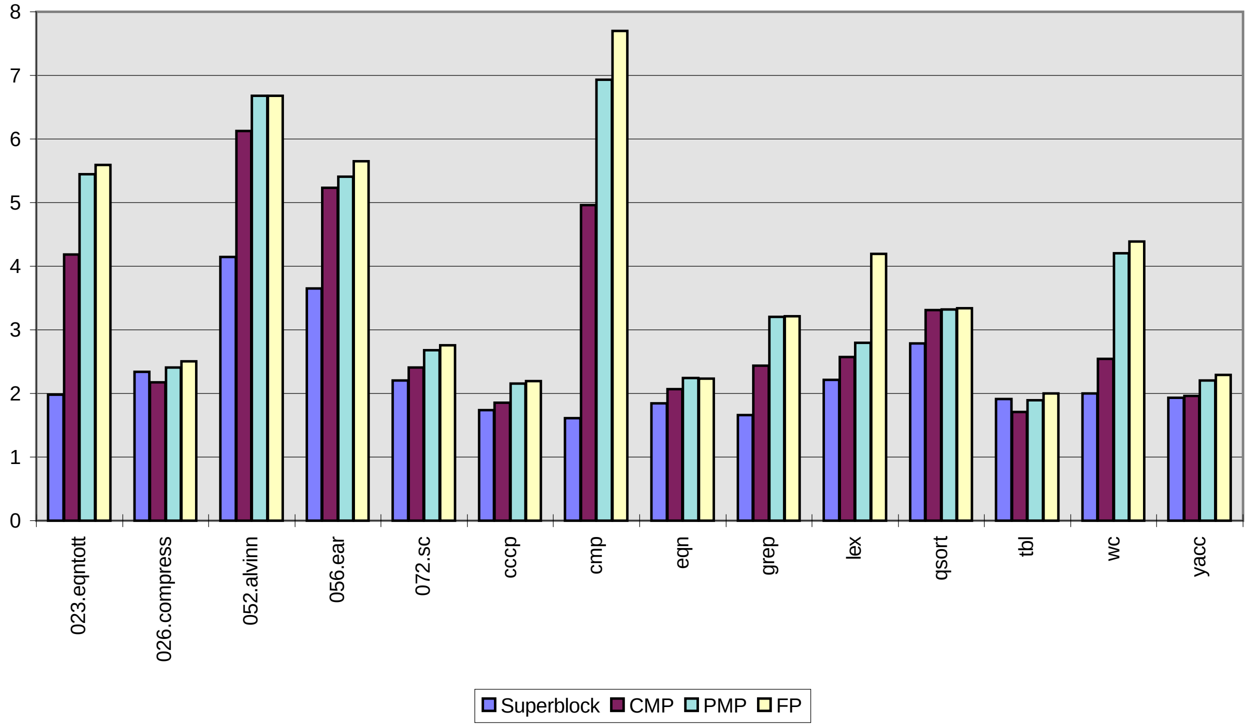


Figure 5.1 Effectiveness of FP and CMP support for an 8-issue, 1-branch processor.

speedup of the FP model. Clearly, the support of predicate definition instructions is much more of a factor in these experiments than the ability to predicate every instruction.

One of the benefits of supporting predicate definition instructions is that predicate definitions can be performed with relatively few instructions. Figure 5.2 shows the relative number of instructions executed by each model. The number of instructions executed by the CMP model is significantly more than the number executed by the other models. The number of instructions executed by the PMP model is much more similar to the number executed by the FP than the CMP model. Therefore, the conversion of predicate definition instructions into simple logic instructions accounts for a much larger portion of the extra instructions executed by the CMP model than the conversion of predicated instructions does. Figure 5.3 shows that, unlike the CMP model, the PMP model still achieves most of the speedup of the FP model.

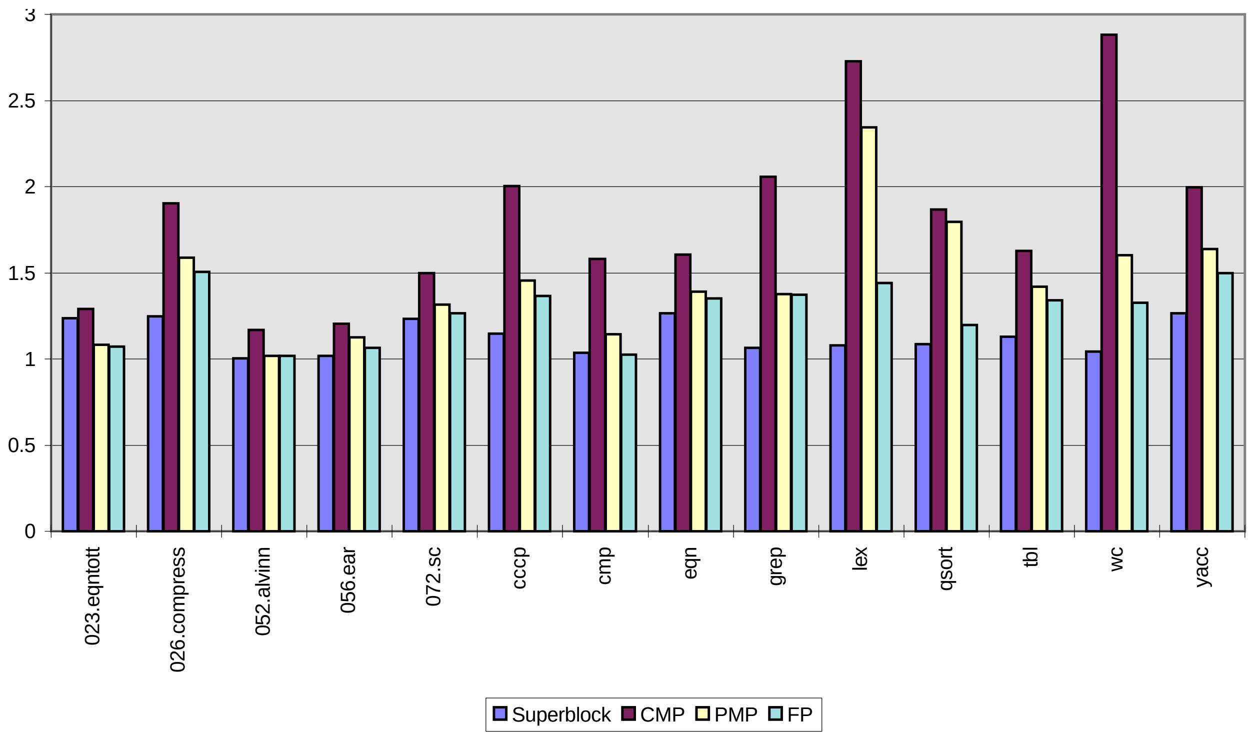


Figure 5.2 Dynamic instruction count comparison.

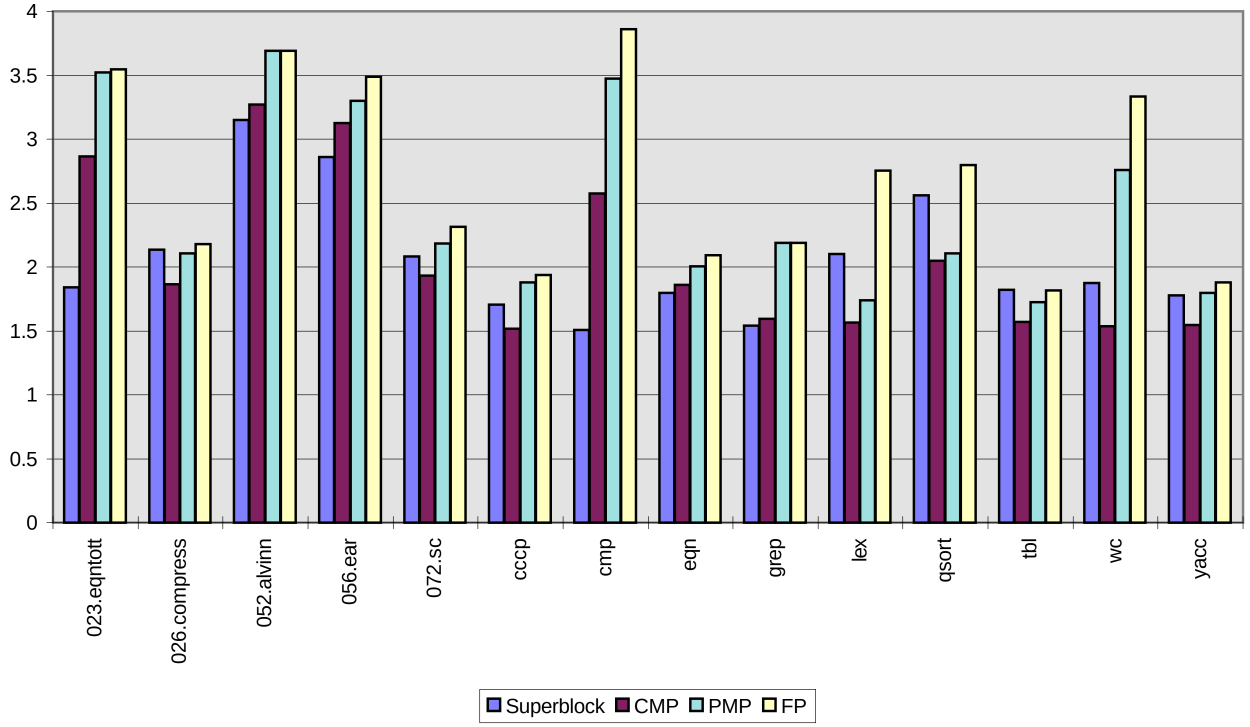


Figure 5.3 Effectiveness of FP and CMP support with only 4-issue capability.

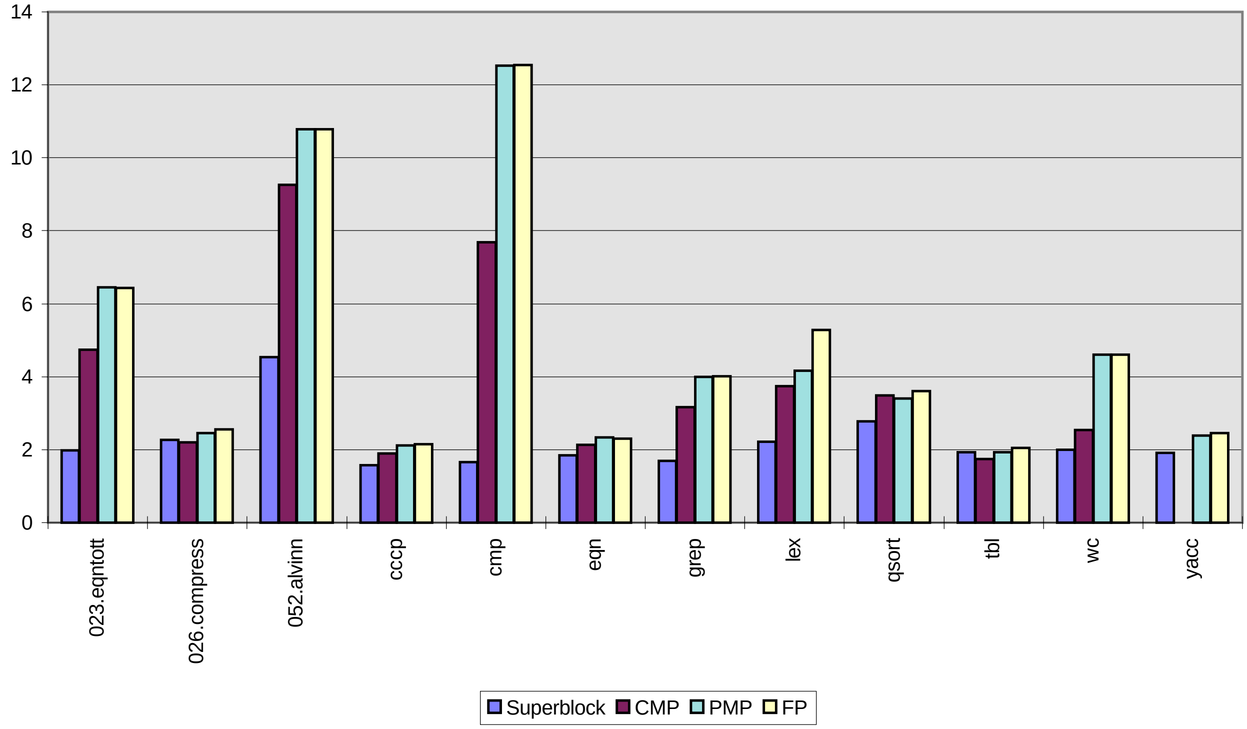


Figure 5.4 Effectiveness of FP and CMP support with 16-issue capability.

This implies that the PMP model has not saturated the issue resources of the processor as the CMP model has done, further supporting the significance of the existence of predicate definition instructions in decreasing the instruction count. On average, the PMP model maintains almost 70% of the advantage of the PMP model over the Superblock model.

Another benefit of supporting predicate definition instructions is that they reduce the dependence height of chains of predicate definitions. Figure 5.4 allows significant insight into the difference that predicate definition instructions make in the area of dependence height. In this figure the performance of the PMP model is almost identical to that of the FP model. On the other hand, the CMP model does not perform as well. These data imply that the dependence height increases that result from the basic conversions for predicated instructions are not nearly as important as the dependence height increases resulting from the basic conversions of predicate definition instructions.

In summary, support of the predicate definition instructions encompasses most of the differences in performance between the CMP and FP models in both categories of instruction count and dependence height. It appears, then, that providing support which allows all instructions to be predicated does not have much of an impact on performance (as long as the underlying architecture has good support for speculation).

Figures 5.5 - 5.8 and Table 5.1 are included for comparison with the results in Chapter 4. The trend of the PMP model achieving most of the performance of the FP model holds up for each configuration. An oddity that is worthy of note is the fact that in a very few cases the expected performance order of the four models is disturbed. This occurs sometimes when a dynamic system effect is factored into similar performance results. Since the code varies significantly for each model, cache and BTB access patterns can be considerably different. In a few cases, these differences cause a performance ordering which is different than the usual ordering.

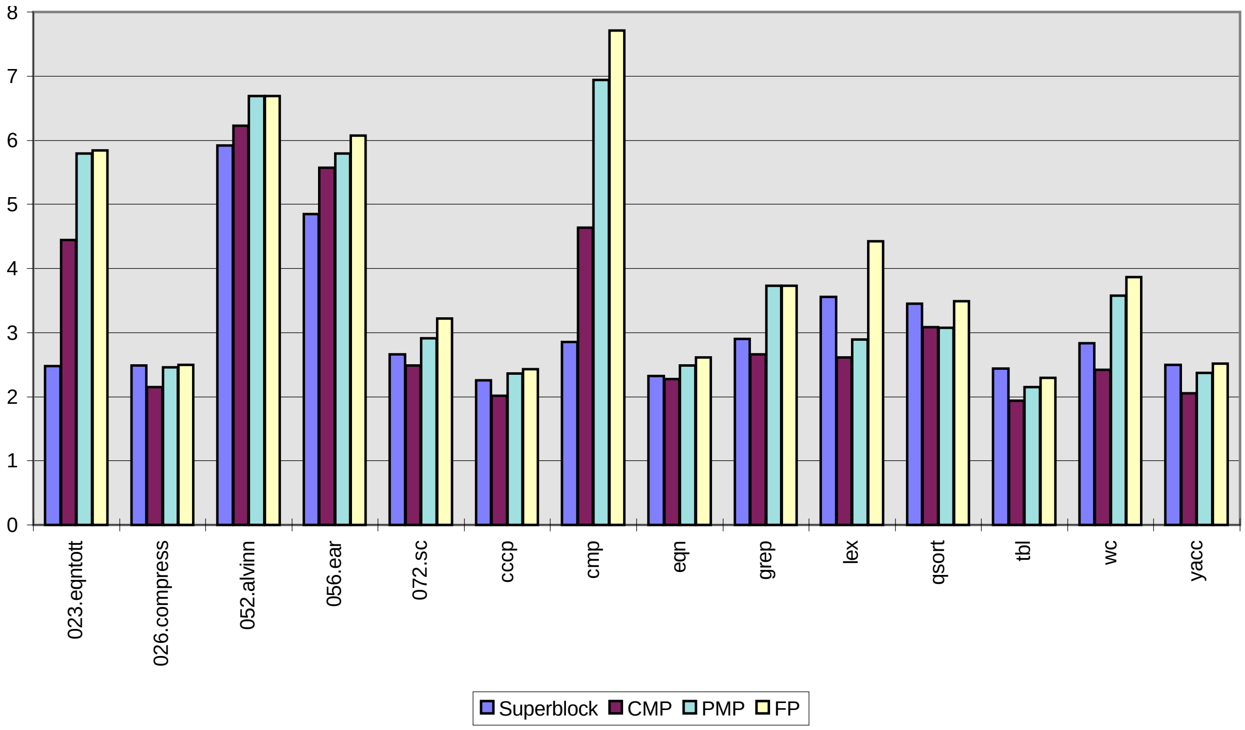


Figure 5.5 Effectiveness of FP and CMP support with 2-branch issue capability.

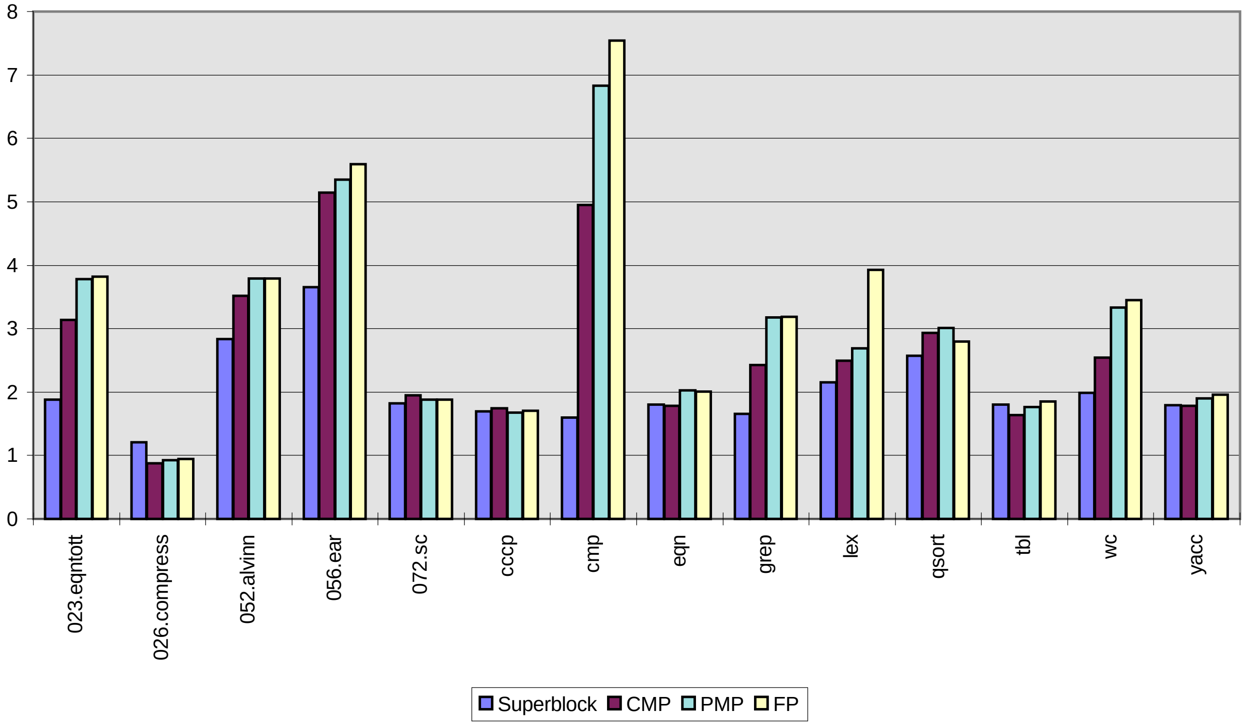


Figure 5.6 Effectiveness of FP and CMP support with 16K instruction and data caches.

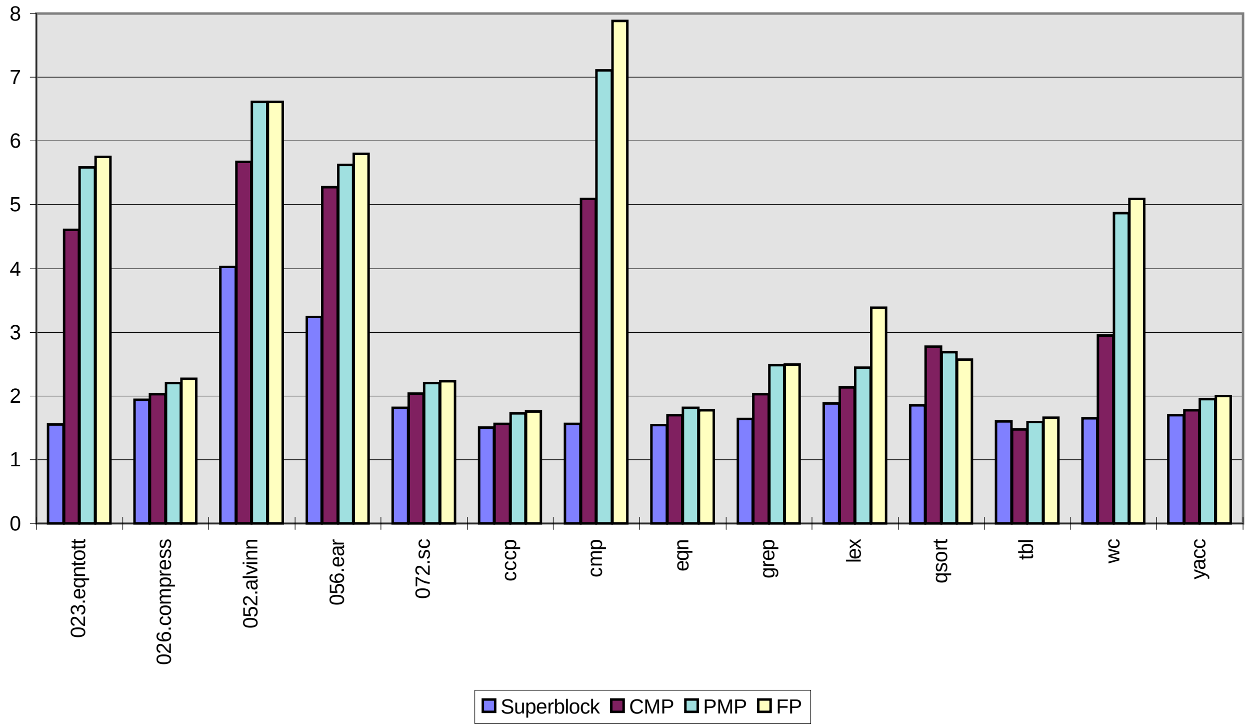


Figure 5.7 Effectiveness of FP and CMP support with 10-cycle misprediction penalty.

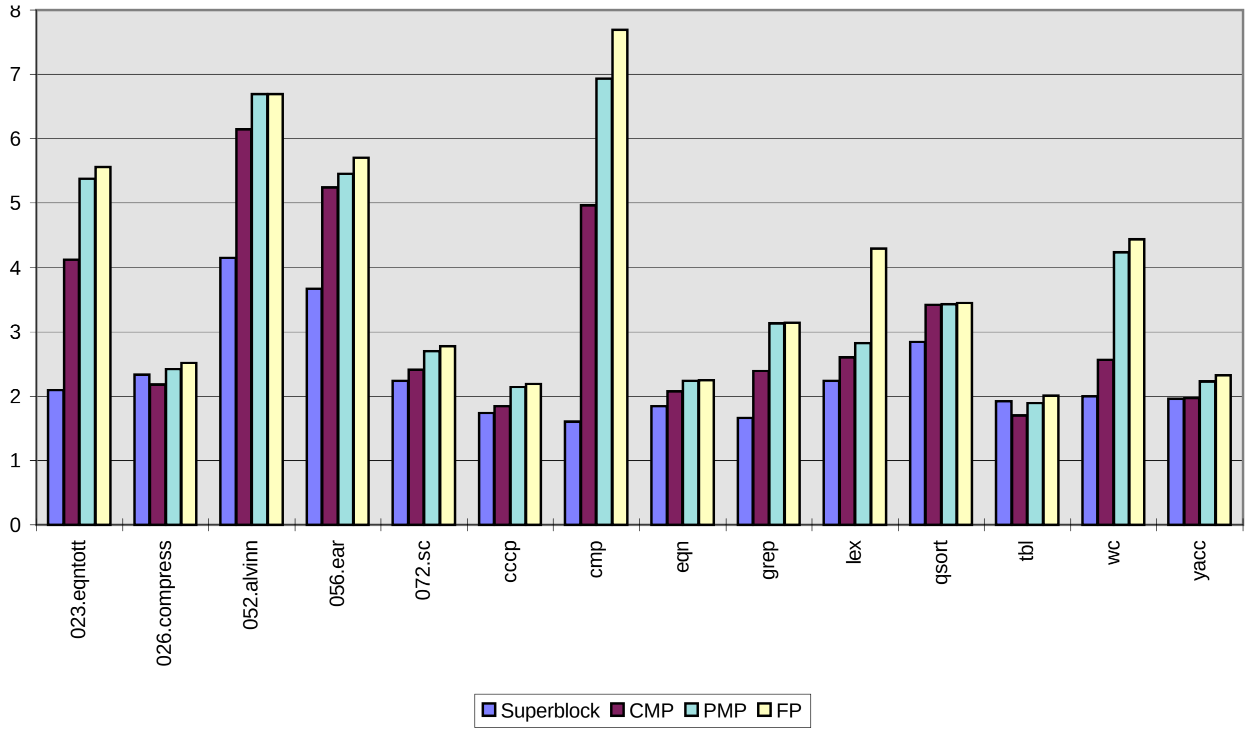


Figure 5.8 Effectiveness of FP and CMP support with a better (two-level) branch predictor.

CHAPTER 6

CONCLUSIONS AND SUGGESTIONS FOR FURTHER RESEARCH

6.1 Conclusions

Predicated execution is a very useful tool for exposing and exploiting ILP. Extensive support for predication as with the FP model provides the most performance benefits. Unfortunately, such support can be costly and very difficult to fit into existing instruction sets. Lower levels of support for predicated execution can still be quite useful. With just a conditional move instruction and compiler support, the CMP model was able to achieve a significant performance advantage over a model with no support for predicated execution.

New compiler techniques were introduced which allow conversion of code from an architecture with extensive support for predication to various architectures with lower levels of support from predication. The advantage of compiling for a low level of predication via an intermediate representation which has a high level of support for predication is apparent from the good performance results achieved for the CMP model compared to previous efforts at exploiting a conditional move instruction.

It was shown that a significant portion of the advantage of extensive support for predicated execution comes from an extensive set of predicate definition instructions. Implementing predicate definitions without a very specialized set of instructions greatly increases the instruction count and the dependence height of the code. The fact that the PMP model achieved almost the same performance results as the FP model showed that allowing every instruction to be predicated is not that important.

6.2 Suggestions for Further Research

This section contains some suggestions for additional research in this area. First, variations of the techniques in this thesis could be applied to a system that does not have architectural support for speculative execution. The baseline architecture in this thesis is assumed to have non-excepting versions of all instructions. This support allowed many instructions to be efficiently speculated. Because many instructions are speculated in the code generated by the basic conversions in Section 3.3.2, elimination of the non-excepting versions of instructions may significantly impact the performance of this code. Since it also contains speculated instructions, the FP code would also be affected by such a change.

Second, the effects of register pressure were not considered in this research. An infinite number of registers was assumed in the baseline architecture. Predicates are stored in integer registers in the CMP model making register pressure a significant issue for this model. The increase in register pressure due to storing predicates should be measured and ways for dealing with it should be explored. One way to alleviate the crowding of the integer register file in the CMP model would be to create a separate register file for the predicate registers. However, with this modification, special instructions or some kind of special treatment would then be needed to define predicates.

Third, determining a good set of predicate definition instructions for a three-operand ISA is an open area of research. The PMP support for predication presented in this thesis is appealing because it would be easier to fit into an existing instruction set while still maintaining most of the performance of the FP model. However, accommodating the set of predicate definition instructions in the PMP model would be very difficult in an existing ISA. One problem is the sheer number of different predicate definition instructions. Another more significant problem is the number of operands assumed for the predicate definition instructions. A predicate definition instruction in the PMP model can have two predicate register definition operands, two integer source operands, and one predicate register source operand for a total of five operands. Many ISAs have three-operand instructions, and almost none have more than three. Aside from the difficulty of modifying the hardware to handle up to five operands, actually fitting five register operands into a typical 32-bit instruction word can be difficult. I believe that efficiency of the predicate definition instructions in the PMP model is inherently tied to the large number of operands which these instructions have. If the predicate definition instructions in the PMP model were limited to three operands, the resulting code would probably not be much better than the CMP code. Developing and studying the performance of a relatively efficient set of three-operand predicate definition instructions would be a significant contribution to this area of research.

Fourth, the convert-to-NOP method used in this research for squashing instructions with false predicates may be infeasible for machines with longer pipelines or out-of-order issue. The other choice for squashing instructions is the data selection model given in Section 2.1.

REFERENCES

[1] S. A. Mahlke, R. E. Hank, J. McCormick, D. I. August, and W. W. Hwu, "A comparison of full and partial predicated execution support for ILP processors," in *Proceedings of the 22th International Symposium on Computer Architecture*, pp. 138–150, June 1995.

[2] J. E. Smith, "A study of branch prediction strategies," in *Proceedings of the 8th International Symposium on Computer Architecture*, pp. 135–148, May 1981.

[3] J. Lee and A. J. Smith, "Branch prediction strategies and branch target buffer design," *IEEE Computer*, pp. 6–22, January 1984.

[4] T. Y. Yeh and Y. N. Patt, "A comparison of dynamic branch predictors that use two levels of branch history," in *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pp. 257–266, May 1993.

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

[6] W. W. Hwu, S. A. Mahlke, W. Y. Chen, P. P. Chang, N. J. Warter, R. A. Bringmann, R. G. Ouellette, R. E. Hank, T. Kiyohara, G. E. Haab, J. G. Holm, and D. M. Lavery, "The Superblock: An effective technique for VLIW and superscalar compilation," *The Journal of Supercomputing*, vol. 7, pp. 229–248, January 1993.

[7] M. S. Lam, "Software pipelining: An effective scheduling technique for VLIW machines," in *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation*, pp. 318–328, June 1988.

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

[9] G. R. Beck, D. W. Yen, and T. L. Anderson, "The Cydra 5 minisupercomputer: Architecture and implementation," *The Journal of Supercomputing*, vol. 7, pp. 143–180, January 1993.

[10] B. R. Rau, D. W. L. Yen, W. Yen, and R. A. Towle, "The Cydra 5 departmental supercomputer," *IEEE Computer*, vol. 22, pp. 12–35, January 1989.

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

[12] D. N. Pnevmatikatos and G. S. Sohi, "Guarded execution and branch prediction in dynamic ILP processors," in *Proceedings of the 21st International Symposium on Computer Architecture*, pp. 120–129, April 1994.

[13] J. R. Allen, K. Kennedy, C. Porterfield, and J. Warren, "Conversion of control dependence to data dependence," in *Proceedings of the 10th ACM Symposium on Principles of Programming Languages*, pp. 177–189, January 1983.

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

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

[16] N. J. Warter and W. W. Hwu, "Reverse if-conversion for enhanced modulo scheduling," Tech. Rep. in preparation, Center for Reliable and High-Performance Computing, University of Illinois, Urbana, IL, November 1992.