A Comparison of Full and Partial Predicated Execution Support for ILP Processors

Scott A. Mahlke* Richard E. Hank James E. McCormick David I. August Wen-mei W. Hwu

Center for Reliable and High-Performance Computing
University of Illinois
Urbana-Champaign, IL 61801

Abstract

One can effectively utilize predicated execution to improve branch handling in instruction-level parallel processors. Although the potential benefits of predicated execution are high, the tradeoffs involved in the design of an instruction set to support predicated execution can be difficult. On one end of the design spectrum, architectural support for full predicated execution requires increasing the number of source operands for all instructions. Full predicate support provides for the most flexibility and the largest potential performance improvements. On the other end, partial predicated execution support, such as conditional moves, requires very little change to existing architectures. This paper presents a preliminary study to qualitatively and quantitatively address the benefit of full and partial predicated execution support. With our current compiler technology, we show that the compiler can use both partial and full predication to achieve speedup in large control-intensive programs. Some details of the code generation techniques are shown to provide insight into the benefit of going from partial to full predication. Preliminary experimental results are very encouraging: partial predication provides an average of 33% performance improvement for an 8-issue processor with no predicate support while full predication provides an additional 30% improvement.

1 Introduction

Branch instructions are recognized as a major impediment to exploiting instruction-level parallelism (ILP). ILP is limited by branches in two principle ways. First, branches impose control dependences which restrict the number of independent instructions available each cycle. Branch prediction in conjunction with speculative execution is typically utilized by the compiler and/or hardware to remove control dependences and expose ILP in superscalar and VLIW processors [1] [2] [3]. However, misprediction of these branches can result in severe performance penalties. Recent studies have reported a performance reduction of two to more than ten when realistic instead of perfect branch prediction is utilized [4] [5] [6]. The second limitation is that processor resources to handle branches are often restricted. As a result, for control intensive applications, an artificial upper bound on performance will be imposed by the branch resource constraints. For example, in an instruction stream consisting of 40% branches, a four issue processor capable of processing only one branch per cycle is bounded to a maximum of 2.5 sustained instructions per cycle.

Predicated execution support provides an effective means to eliminate branches from an instruction stream. Predicated or guarded execution refers to the conditional execution of an instruction based on the value of a boolean source operand, referred to as the predicate [7] [8]. This architectural support allows the compiler to employ an *if-conversion* algorithm to convert conditional branches into predicate defining instructions, and instructions along alternative paths of each branch into predicated instructions [9] [10] [11]. Predicated instructions are fetched regardless of their predicate value. Instructions whose predicate is true are executed normally. Conversely, instructions whose predicate is false are nullified, and thus are prevented from modifying the processor state.

Predicated execution provides the opportunity to significantly improve branch handling in ILP processors. The most obvious benefit is that decreasing the number of branches reduces the need to sustain multiple branches per cycle. Therefore, the artificial performance bounds imposed by limited branch resources can be alleviated. Eliminating frequently mispredicted branches also leads to a substantial reduction in branch prediction misses [12]. As a result, the performance penalties associated with mispredictions of the eliminated branches are removed. Finally, predicated execution provides an efficient interface for the compiler to expose multiple execution paths to the hardware. Without compiler support, the cost of maintaining multiple execution paths in hardware grows exponentially.

Predicated execution may be supported by a range of architectural extensions. The most complete approach is full

* Scott Mahlke is now with Hewlett Packard Laboratories, Palo Alto, CA.

predicate support. With this technique, all instructions are provided with an additional source operand to hold a predicate specifier. In this manner, every instruction may be a predicated. Additionally, a set of predicate defining opcodes are added to efficiently manipulate predicate values. This approach was most notably utilized in the Cydra 5 minisupercomputer [8] [13]. Full predicate execution support provides the most flexibility and the largest potential performance improvements. The other approach is to provide partial predicate support. With partial predicate support, a small number of instructions are provided which conditionally execute, such as a conditional move. As a result, partial predicate support minimizes the required changes to existing instruction set architectures (ISA's) and data paths. This approach is most attractive for designers extending current ISA's in an upward compatible manner.

In this paper, the tradeoffs involved in supporting full and partial predicated execution are investigated. Using the compilation techniques proposed in this paper, partial predicate support enables the compiler to perform full if-conversion to eliminate branches and expose ILP. Therefore, the compiler may remove as many branches with partial predicate support as with full predicate support. By removing a large portion of the branches, branch handling is significantly improved for ILP processors with partial predicate support. The relatively few changes needed to add partial predicate support into an architecture make this approach extremely attractive for designers.

However, there are several fundamental performance limitations of partial predicate support that are overcome with full predicate support. These difficulties include representing unsupported predicated instructions, manipulating predicate values, and relying extensively on speculative execution. In the first case, for an architecture with only partial predicate support, predicated operations must be performed using an equivalent sequence of instructions. Generation of these sequences results in an increase in the number of instructions executed and requires a larger number of registers to hold intermediate values for the partial predicate architecture. In the second case, the computation of predicate values is highly efficient and parallel with full predicate support. However, this same computation with partial predicate support requires a chain of sequentially dependent instructions, that can frequently increase the critical path length. Finally, the performance of partial predicate support is extensively dependent on the use of speculative execution. Conditional computations are typically represented by first performing the computation unconditionally (speculative) and storing the result(s) in some temporary locations. Then, if the condition is true, the processor state is updated, using one or more conditional moves for example. With full predicate support, speculation is not required since all instructions may have a predicate specifier. Thus, speculation may be selectively employed where it improves performance rather than always being utilized.

The issues discussed in the paper are intended for both designers of new ISA's, as well as those extending existing ISA's. With a new instruction set, the issue of supporting full or partial predicate support is clearly a choice that is available. Varying levels of partial predicate support provide options for extending an existing ISA. For example, introducing *guard* instructions which hold the predicate specifiers of subsequent instructions may be utilized [14].

2 ISA Extensions

In this section, a set of extensions to the instruction set architecture for both full and partial predicate support are presented. The baseline architecture assumed is generic ILP processor (either VLIW or superscalar) with in-order issue and register interlocking. A generic load/store ISA is further assumed as the baseline ISA.

2.1 Extensions for Full Predication

The essence of predicated execution is the ability to suppress the modification of the processor state based upon some condition. There must be a way to express this condition and a way to express when the condition should affect execution. Full predication cleanly supports this through a combination of instruction set and micro-architecture extensions. These extensions can be classified as support for suppression of execution and expression of condition.

Suppression of Execution. The result of the condition which determines if an instruction should modify state is stored in a set of 1-bit registers. These registers are collectively referred to as the predicate register file. The setting of these registers is discussed later in this section. The values in the predicate register file are associated with each instruction in the extended instruction set through the use of an additional source operand. This operand specifies which predicate register will determine whether the operation should modify processor state. If the value in the specified predicate register is 1, or true, the instruction is executed normally; if the value is 0, or false, the instruction is suppressed.

One way to perform the suppression of an instruction in hardware is to allow the instruction to execute and to disallow any change of processor state in the write-back stage of the pipeline. This method is useful since it reduces the latency between an instruction that modifies the value of the predicate register and a subsequent instruction which is conditioned based on that predicate register. This reduced latency enables more compact schedules to be generated for predicated code. A drawback to this method is that regardless of whether an instruction is suppressed, it still ties up an execution unit. This method may also increase the complexity of the register bypass logic and force exception signalling to be delayed until the last pipeline stage.

An instruction can also be suppressed during the decode/issue stage. Thus, an instruction whose corresponding predicate register is false is simply not issued. This has the advantage of allowing the execution unit to be allocated to other operations. Since the value of the predicate register referenced must be available during decode/issue, the predicate register must at least be set in the previous cycle. This dependence distance may also be larger for deeper pipelines or if bypass is not available for predicate registers. Increasing the dependence distance between definitions and uses of predicates may adversely affect execution time by lengthening the schedule for predicated code. An example of this

P_{in}	Comparison	P_{out}: 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

Table 1: Predicate definition truth table.

suppression model is the predicate support provided by the Cydra 5 [8]. Suppression at the decode/issue stage is also assumed in our simulation model.

Expression of Condition. A set of new instructions is needed to set the predicate registers based upon conditional expressions. These instructions can be classified as those that define, clear, set, load, or store predicate registers.

Predicate register values may be set using predicate define instructions. The predicate define semantics used are those of the HPL Playdoh architecture [15]. There is a predicate define instruction for each comparison opcode in the original instruction set. The major difference with conventional comparison instructions is that these predicate defines have up to two destination registers and that their destination registers are predicate registers. The instruction format of a predicate define is shown below.

pred_<*cmp*> $\text{Pout1}_{<type>}$, $\text{Pout2}_{<type>}$, src1, src2 (P_{in})

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

The predicate <*type*> determines the value written to the destination predicate register based upon the result of the comparison and of the input predicate, P_{in}. For each combination of comparison result and P_{in}, one of three actions may be performed on the destination predicate. It can write 1, write 0, or leave it unchanged. A total of $3^4 = 81$ possible types exist.

There are six predicate types which are particularly useful, the unconditional (U), OR, and AND type predicates and their complements. Table 1 contains the truth table for these predicate types.

Unconditional destination predicate registers are always defined, regardless of the value of P_{in} and the result of the comparison. If the value of P_{in} is 1, the result of the comparison is placed in the predicate register (or its compliment for $\overline{U}$). Otherwise, a 0 is written to the predicate register. Unconditional predicates are utilized for blocks which are executed based on a single condition, i.e., they have a single control dependence.

The OR type predicates are useful when execution of a block can be enabled by multiple conditions, such as logical AND (&&) and OR (||) constructs in C. OR type destination predicate registers are set if P_{in} is 1 and the result of the comparison is 1 (0 for $\overline{OR}$), otherwise the destination predicate register is unchanged. Note that OR type predicates must be explicitly initialized to 0 before they are defined and used. However, after they are initialized multiple OR type predicate defines may be issued simultaneously and in any

(a)	(b)	(c)
if (a&&b)	beq a,0,L1	pred_clear
$j = j + 1;$	beq b,0,L1	pred_eq p1_{OR},$\text{p2}_{\overline{U}}$,a,0
else	add j,j,1	pred_eq p1_{OR},$\text{p3}_{\overline{U}}$,b,0 (p2)
if (c)	jump L3	add j,j,1 (p3)
$k = k + 1;$	L1:	pred_ne p4_{U},$\text{p5}_{\overline{U}}$,c,0 (p1)
else	bne c,0,L2	add k,k,1 (p4)
$k = k - 1;$	add k,k,1	sub k,k,1 (p5)
$i = i + 1;$	jump L3	add i,i,1
	L2:	
	sub k,k,1	
	L3:	
	add i,i,1	

Figure 1: Example of predication, (a) source code, (b) assembly code, (c) assembly code after if-conversion.

order on the same predicate register. This is true since the OR type predicate either writes a 1 or leaves the register unchanged which allows implementation as a wired logical OR condition. This property can be utilized to compute an execution condition with zero dependence height using multiple predicate define instructions.

AND type predicates, are analogous to the OR type predicate. AND type destination predicate registers are cleared if P_{in} is 1 and the result of the comparison is 0 (1 for $\overline{AND}$), otherwise the destination predicate register is unchanged. The AND type predicate is particularly useful for transformations such as control height reduction [16].

Although it is possible to individually set each predicate register to zero or one through the use of the aforementioned predicate define instructions, in some cases individually setting each predicate can be costly. Therefore, two instructions, *pred_clear* and *pred_set*, are defined to provide a method of setting the entire predicate register file to 0 or 1 in one cycle.

Code Example. Figure 1 contains a simple example illustrating the concept of predicated execution. The source code in Figure 1(a) is compiled into the code shown in Figure 1(b). Using if-conversion [10], the code is then transformed into the code shown in Figure 1(c). The use of predicate registers is initiated by a *pred_clear* in order to insure that all predicate registers are cleared. The first two conditional branches in (b) are translated into two *pred_eq* instructions. Predicate register *p1* is OR type since either condition can be true for *p1* to be true. If *p2* in the first *pred_eq* is false the second *pred_eq* is not executed. This is consistent with short circuit boolean evaluation. *p3* is true only if the entire expression is true. The "then" part of the outer if statement is predicated on *p3* for this reason. The *pred_ne* simply decides whether the addition or subtraction instruction is performed. Notice that both *p4* and *p5* remain at zero if the *pred_ne* is not executed. This is consistent with the "else" part of the outer if statement. Finally, the increment of i is performed unconditionally.

2.2 Extensions for Partial Predication

Enhancing an existing ISA to support only partial predication in the form of conditional move or select instructions trades off the flexibility and efficiency provided by full predication in order to minimize the impact to the ISA. Several existing architectures provide instruction set features that reflect this point of view.

Conditional Move. The conditional move instruction provides a natural way to add partial support for predicated execution to an existing ISA. A conditional move instruction has two source operands and one destination operand, which fits well into current 3 operand ISA's. The semantics of a conditional move instruction, shown below, are similar to that of a predicated move instruction.

cmov dest,src,cond
if (cond) dest = src

As with a predicated move, the contents of the source register are copied to the destination register if the condition is true. Also, the conditional modification of the target register in a conditional move instruction allows simultaneous issue of conditional move instructions having the same target register and opposite conditions on an in-order processor. The principal difference between a conditional move instruction and a predicated move instruction is that a register from the integer or floating-point register file is used to hold the condition, rather than a special predicate register file. When conditional moves are available, we also assume conditional move complement instructions (*cmov_com*) are present. These are analogous in operation to conditional moves, except they perform the move when *cond* is false, as opposed to when *cond* is true.

The Sparc V9 instruction set specification and the DEC Alpha provide conditional move instructions for both integer and floating point registers. The HP Precision Architecture [17] provides all branch, arithmetic, and logic instructions the capability to conditionally nullify the subsequent instruction. Currently the generation of conditional move instructions is very limited in most compilers. One exception is the DEC GEM compiler that can efficiently generate conditional moves for simple control constructs [18].

Select. The select instruction provides more flexibility than the conditional move instruction at the expense of pipeline implementation. The added flexibility and increased difficulty of implementation is caused by the addition of a third source operand. The semantics of the select instruction are shown below.

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

Unlike the conditional move instruction, the destination register is always modified with a select. If the condition is true, the contents of **src1** are copied to the destination, otherwise the contents of **src2** are copied to the destination register. The ability to choose one of two values to place in the destination register allows the compiler to effectively choose between computations from "then" and "else" paths of conditionals based upon the result of the appropriate comparison. As a result, select instructions enable more efficient transformations by the compiler. This will be discussed in more detail in the next section. The Multiflow Trace 300 series machines supported partial predicated execution with select instructions [19].

3 Compiler Support

The compiler eliminates branch instructions by introducing conditional instructions. The basic transformation is known as if-conversion [9] [10]. In our approach, full predicate support is assumed in the intermediate representation (IR) regardless of the the actual architectural support in the target processor. A set of compilation techniques based on the hyperblock structure are employed to effectively exploit predicate support in the IR [11]. For target processors that only have partial predicate support, unsupported predicated instructions are broken down into sequences of equivalent instructions that are representable. Since the transformation may introduce inefficiencies, a comprehensive set of peephole optimizations is applied to code both before and after conversion. This approach of compiling for processors with partial predicate support differs from conventional code generation techniques. Conventional compilers typically transform simple control flow structures or identify special patterns that can utilize conditional moves or selects. Conversely, the approach utilized in this paper enables full if-conversion to be applied with partial predicate support to eliminate control flow.

In this section, the hyperblock compilation techniques for full predicate support are first summarized. Then, the transformation techniques to generate partial predicate code from a full predicate IR are described. Finally, two examples from the benchmark programs studied are presented to compare and contrast the effectiveness of full and partial predicate support using the these compilation techniques.

3.1 Compiler Support for Full Predication

The compilation techniques utilized in this paper to exploit predicated execution are based on a structure called a *hyperblock* [11]. A hyperblock is a collection of connected basic blocks in which control may only enter at the first block, designated as the entry block. Control flow may leave from one or more blocks in the hyperblock. All control flow between basic blocks in a hyperblock is eliminated via if-conversion. The goal of hyperblocks is to intelligently group basic blocks from many different control flow paths into a single block for compiler optimization and scheduling.

Basic blocks are systematically included in a hyperblock based on two, possibly conflicting, high level goals. First, performance is maximized when the hyperblock captures a large fraction of the likely control flow paths. Thus, any blocks to which control is likely to flow are desirable to add to the hyperblock. Second, resource (fetch bandwidth and function units) are limited; therefore, including too many blocks may over saturate the processor causing an overall performance loss. Also, including a block which has a comparatively large dependence height or contains a hazardous instruction (e.g., a subroutine call) is likely to result in performance loss. The final hyperblock consists of a linear sequence of predicated instructions. Additionally, there are explicit exit branch instructions (possibly predicated) to any blocks not selected for inclusion in the hyperblock. These branch instructions represent the control flow that was identified as unprofitable to eliminate with predicated execution support.

	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 2: Example of predicate promotion.

3.2 Compiler Support for Partial Predication

Generating partially predicated code from fully predicated code involves removing predicates from all instructions which are not allowed to have a predicate specifier. The only instruction set remnants of predication in the partially predicated code are conditional move or select instructions. Transforming fully predicated code to partially predicated code is essentially accomplished by converting predicated instructions into speculative instructions which write to some temporary location. Then, conditional move or select instructions are inserted to conditionally update the processor state based on the value of the predicate. Since all predicated instructions are converted to speculative instructions, the efficiency of the partially predicated code is heavily dependent on the underlying support for speculation provided by the processor. In this section, the code generation procedure chosen to implement the full to partial predication transformation is described. The procedure is divided into 3 steps, predicate promotion, basic conversion, and peephole optimization.

Predicate Promotion. The conversion of predicated instructions into an equivalent set of instructions that only utilize conditional moves or selects introduces a significant amount of code expansion. This code expansion is obviously reduced if there are fewer predicated instructions that must be converted. Predicate promotion refers to removing the predicate from a predicated instruction [11]. As a result, the instruction is unconditionally executed. By performing predicate promotion, fewer predicated instructions remain in the IR that must be converted.

An example to illustrate the effectiveness of predicate promotion is presented in Figure 2. The code sequence in the upper left box is the original fully predicated IR. Straightforward conversion to conditional move code, as will be discussed in the next subsection, yields the code in the upper right box. Each predicated instruction is expanded into two instructions for the partial predicate architecture. All the conditional moves in this sequence, except for the last, are unnecessary if the original destination registers of the predicated instructions are temporary registers. In this case, the predicate of the first two instructions can be promoted, as shown in the lower left box of Figure 2. The *add* instruction is the only remaining predicated instruction. Finally, conversion to conditional move code after promotion yields the code sequence in the bottom right box of Figure 2. In all, the number of instructions is reduced from 6 to 4 in this example with predicate promotion.

It should be noted that predicate promotion is also effective for architectures with full predicate support. Predicate promotion enables speculative execution by allowing predicated instructions to execute before their predicate is calculated. In this manner, the dependence between the predicate definition and the predicated instruction is eliminated. The hyperblock optimizer and scheduler utilize predicate promotion when the predicate calculation occurs along a critical dependence chain to reduce this dependence length.

Basic Conversions. In the second step of the transformation from fully predicated code to partially predicated code, a set of simple transformations, referred to as basic conversions, are applied to each remaining predicated instruction independently. The purpose of the basic conversions is to replace each predicated instruction by a sequence of instructions with equivalent functionality. The sequence is limited to contain conditional moves as the only conditional instructions. As a result, most instructions in the sequence must be executed without a predicate. These instructions thus become speculative. When generating speculative instructions, the compiler must ensure they only modify temporary registers or memory locations. Furthermore, the compiler must ensure the speculative instructions will not cause any program terminating exceptions when the condition turns out to be false. Program terminating exceptions include illegal memory address, divide-by-zero, overflow, or underflow.

The basic conversions that may be applied are greatly simplified if the underlying processor has support full support for speculative execution. In particular, non-excepting or silent, instructions allow for the most efficient transformations. For such an architecture, the basic conversions for the main classes of instructions are summarized in Figure 3. The simplest conversion is used for predicated arithmetic and logic instructions and also for memory loads. The conversion, as can be seen in Figure 3, is to rename the destination of the predicated instruction, remove the predicate, and then conditionally move the result into the original destination based on the result of the predicate.

The basic conversions for memory store instructions are similar. Since the destination of a store instruction is a memory location instead of a register, a different technique must be used to insure that the an invalid value is not written to the original destination of the store. Figure 3 shows that the address of the store is calculated separately. Then a conditional move is used to replace the address of the store with *$safe_addr* when the predicate of the store is false. The macro *$safe_addr* refers to a reserved location on the stack.

The conversions for predicate definition instructions are the most complicated because predicate definitions have rather complicated logic capabilities. The conversions for two representative predicate definition instructions are shown in Figure 3. The predicate definition instructions are identical except for the type on the destination predicate register. The transformation for the *OR* type predicate

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

Figure 3: Basic conversions assuming non-excepting instructions available in the architecture.

produces three instructions. The first instruction performs the *lt* comparison of *src1* and *src2*, placing the result in a temporary register. Each predicate definition transformation generates such a comparison instruction. The second instruction performs a logical AND which clears the temporary register if the predicate Pin is false. This clearing instruction is generated only if the predicate definition instruction is predicated. The third instruction performs a logical OR of the value in the temporary register with the previous value of the OR type predicate $Pout$ and deposits the result t in $Pout$. For an AND type predicate, the result would be stored with a logical AND. For an unconditional predicate, a separate depositing instruction is not necessary.

The basic conversions for branches are relatively straight forward and are left to the reader. Predicated subroutine calls are handled by branching around them when the predicate is false since conditional calls were not assumed in the architecture.

Conversions are also possible if no speculation support is provided. However, in addition to insuring that registers or memory locations are not illegally modified, the basic conversions must also prevent exceptions when the original predicate is false. Figure 4 shows three typical conversions. The non-excepting versions of these appeared in Figure 3. Note that the excepting versions produce more instructions than the corresponding conversions for non-excepting instructions. For predicate definition, arithmetic, and logic instructions, the only difference in the conversions is that a value that is known to prevent an exception is conditionally moved into one of the source operands of the previously predicated instruction. These values, which depend on the type of instruction, are referred to as *$safe_val* in the figure.

Fully Predicated Code	Basic Conversions, Excepting Instructions
predicate definition instructions	
pred_lt_f Pout$_{U}$,src1,src2 (Pin)	mov temp_src,src2 cmov_com temp_src,$safe_val,Pi lt_f temp_dest,src1,temp_src and Pout,temp_dest,Pin
arithmetic & logic instructions	
div_f dest,src1,src2 (Pin)	mov temp_src,src2 cmov_com temp_src,$safe_val,Pin div_f temp_dest,src1,temp_src cmov dest,temp_dest,Pin
memory instructions	
load dest,addr,off (Pin)	add temp_addr,addr,off cmov_com temp_addr,$safe_addr,Pin load temp_dest,temp_addr,0 cmov dest,temp_dest,Pin

Figure 4: Basic conversions without non-excepting instructions available in the architecture.

The conversions for floating point conditional branch instructions are similar. Conversion for load instructions is also similar, only an address known not to cause an illegal memory access is moved into the address source of the load.

Peephole Optimizations. The basic transformations of the previous section introduce some inefficiencies since each instruction is considered independently. Many of these inefficiencies can be removed by applying a set of peephole optimizations after the basic transformation. The goal of these optimizations is to reduce the instruction count and dependence height of the partial predicate code. The optimizations find opportunities for improving code efficiency by investigating the interactions of the various transformations, exploiting special cases, and utilizing the additional functionality of the *select* instruction over the conditional move. Some of the optimizations in this section rely on the existence of complementary AND and OR instructions (*and_not* and *or_not*). These instructions are simply logical instructions in which the second source operand is complemented. The existence of these instructions is assumed in the base instruction set.

Basic conversions of predicate defines introduce redundant comparison and logic instructions. For predicates which only differ in predicate type (U, OR, AND), the comparisons are obviously redundant. Applying common subexpression elimination, copy propagation, and dead code removal after conversion effectively eliminates these redundancies. In some cases, the transformations of similar predicate definitions result in opposite comparisons. If one of these comparisons can be inverted, then one of the comparisons may be eliminated. A comparison can be inverted when each use of the result of this comparison can be inverted without the addition of an instruction. The result of a comparison in a predicate definition instruction used only by *and*, *and_not*, *or*, *or_not*, *cmov*, *cmov_com*, *select*, or a conditional branch may be inverted. The only two non-invertible sources which might contain the result of a predicate definition conversion are the non-inverted inputs of *and_not* and *or_not*. Therefore, in most cases, one of two complementary comparisons resulting from similar predicate definitions can be eliminated.

The use of *OR* type predicates is extremely efficient for architectures with full predicate support. Sequences of *OR* type predicate definitions which all write to the same destination predicate may be simultaneously executed. However, with partial support, these sequences of *OR* type predicate definitions result in a sequential chain of dependent instructions. These strict sequential dependences may be overcome using associativity rules to reduce the height of the dependence chain. The dependence height of the resulting code is $log_2(n)$, where n is the number of *OR* type predicate definitions. An example of OR-Tree optimization is presented in Section 3.3.

Some additional optimizations are possible if a *select* instruction is available. The functionality of the *select* instruction is described in Section 2.2. Through the use of a *select* instruction, one instruction from the sequences used for excepting arithmetic and memory instructions shown in Figure 4 can be eliminated. The detailed use of selects is not discussed in this paper due to space considerations.

3.3 Benchmark Examples

In order to more clearly understand the effectiveness of predicated execution support and the performance tradeoffs of full versus partial 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 they are very control-intensive so they clearly illustrate the effectiveness of full and partial predicate support.

Example Loop from Wc. Figure 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 2.3 is achieved for an 8-issue processor over a 1-issue processor (see Figure 8).

The assembly code after hyperblock formation for the loop segment with full and partial predicate support is shown in Figures 5(b) and (c), 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 full and partial predicate 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 full and partial predicate support in this loop. The resulting performance is increased by 17% with partial predicate support and an additional 88% with full predicate support (see Figure 8).

The performance difference between full and partial predicate support comes from the extra instructions required to represent predicate defines and predicated instructions. As a result, the issue resources of the processor are over saturated with partial predicate support. In the example in Figure 5, the number of instructions is increased from 18 with full predicate support to 31 with partial predicate support. This results in an increase in execution time from 8 to 10 cycles. For the entire benchmark execution, a similar trend is observed. The number of instructions is increased from 1526K with full predicate support to 2999K with partial predicate support, resulting in a speedup increase of 2.7 with partial support to 5.1 with full support (see Figure 8 and Table 2).

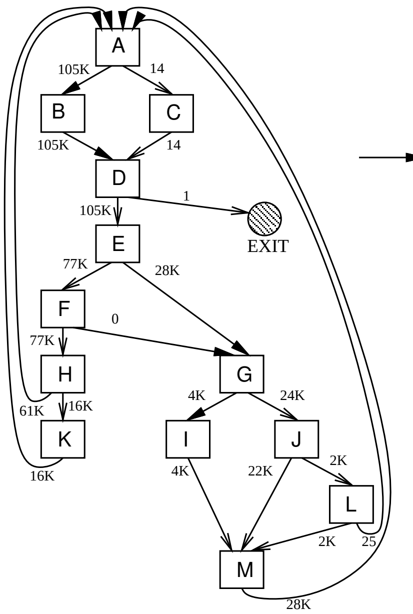
Example Loop from Grep. Figure 6 shows the assembly code for the most important loop segment from the benchmark *grep*. The base processor model, which does not support any predicated execution, employs speculative execution in conjunction with superblock ILP compilation techniques to achieve the schedule shown in Figure 6(a) [20]. Each of the conditional branches in the figure are very 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 1 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.

With full predicate 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 partial predicate support, the same transformations are applied. Therefore, the same number of branches are eliminated. However, the representation of *OR* type predicates is less efficient with partial predicate support. In particular, the logical OR instructions cannot be simultaneously issued. The or-tree optimization discussed previously in Section 3.2 is applied to reduce the dependence height of the sequence and improve performance. In the example, partial predicate support improves performance from 14 to 10 cycles. Overall for the final benchmark performance, partial predicate support improves performance by 46% over the base code and full predicate support further improves performance by 31%.

4 Experimental Evaluation

4.1 Methodology

The predication techniques presented in this paper are evaluated through emulation-driven simulation. The benchmarks studied consist of *008.espresso, 022.li, 023.eqntott, 026.compress, 052.alvinn, 056.ear, and 072.sc* from SPEC-92, and the Unix utilities *cccp, cmp, eqn, grep, lex, qsort, wc, and yacc*. The benchmark programs are initially compiled to produce intermediate code, which is essentially the instruc-



(a) basic block control flow block

```
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

(b) fully predicated hyperblock code

```
blockA: blt       r1,0,blockC     0
        ld_uc     r2,r3,−1        1
        beq       r2,−1,EXIT      3
        lt        r12,32,r2       3
        ge        r33,r2,127      3
        and_not   r13,r12,r33     4
        and       r31,r33,r12     4
        or_not    r11,r31,r12     5
        eq        r34,0,r4        1
        and       r14,r34,r13     5
        eq        r36,r2,10       3
        and_not   r16,r11,r36     6
        and       r17,r36,r11     6
        eq        r37,r2,32       4
        and_not   r18,r16,r37     7
        and       r55,r37,r16     7
        add       r25,r5,1        1
        cmov      r5,r25,r14      6
        add       r24,r4,1        1
        cmov      r4,r24,r14      6
        add       r26,r6,1        1
        cmov      r6,r26,r17      7
        eq        r35,r2,9        4
        and       r65,r35,r18     5
        or        r45,r17,r65     7
        or        r15,r65,r45     8
        cmov      r4,0,r15        9
        add       r7,r7,1         2
        add_u     r3,r3,1         2
        add       r1,r1,−1        2
        jump      blockA          9
```

10 cycles issue cycle

(c) partially predicated code

Figure 5: Example loop segment from *wc*.

tion set of an architecture with varying levels of support for predicated execution. Register allocation and code scheduling are performed in order to produce code that could be executed by a target architecture with such support. To allow emulation of the code on the host HP PA-RISC processor, the code must be modified to remove predication, while providing accurate emulation of predicated instructions.

Emulation ensures that the optimized code generated for each configuration executes correctly. Execution of the benchmark with emulation also generates an instruction trace containing memory address information, predicate register contents, and branch directions. This trace is fed to a simulator for performance analysis of the particular architectural model being studied. We refer to this technique as emulation-driven simulation. The simulator models, in detail, the architecture's prefetch and issue unit, instruction and data caches, branch target buffer, and hardware interlocks, providing an accurate measure of performance.

Predicate Emulation. Emulation is achieved by performing a second phase of register allocation and generating PA-RISC assembly code. The emulation of the varying levels of predicate support, as well as speculation of load instructions is done using the bit manipulation and conditional nullification capabilities of the PA-RISC instruction set [17]. Predicates are emulated by reserving n of the callee-saved registers and accessing them as $32 \times n$ 1-bit registers.

The instruction sequence required to emulate a predicate define instruction is dependent upon the predicate types of the destination predicate registers. As an example, consider the predicated predicate define instruction (1) in Figure 7. In this example, predicate registers *p1*, *p2*, and *p3* have been assigned bits 1,2, and 3 of general register *%r3*, respectively. Instruction (1) is defining predicate register *p1* as *OR* type and *p3* as unconditional complement. The first instruction in the five instruction assembly code sequence, places a 0 in bit 3 of register *%r3*, unconditionally setting *p3* to 0. The second instruction will branch around the remaining instructions if the predicate *p2* is 0. If *p2* is 1 the third instruction then performs the comparison, and using the conditional nullification capabilities of that instruction, determines which of the next two instructions will be executed. If the contents of *%r24* is 0, then only the fifth instruction will be executed, writing a 1 to bit 1 of *%r3*, setting *p1* to 1. Otherwise, only the fourth instruction will be executed, writing a 1 to bit 3 of *%r3*, setting *p3* to 1.

Predicated instructions are emulated by extracting the bit from one of the reserved registers that corresponds to the predicate for that instruction. The value of that bit is used to conditionally execute the predicated instruction. For example, instruction (2) in Figure 7 is predicated on *p3*. Thus, bit 3 is extracted from *%r3* and is used to conditionally nullify the increment of *%r25*.

Conditional Move Emulation. The emulation of conditional move and select instructions is done in a similar fashion. Instruction (3) in Figure 7 is a conditional move of *r6* into *r5* if the contents of *r8* is non-zero. Emulation requires two instructions. The first performs the comparison and nullifies the subsequent copy of *6* into *5* if *r8* is zero. Instruction (4) in Figure 7 is a select instruction. As with the conditional move instruction, the first instruction performs a comparison to determine the contents of *r8*. If *r8* is zero, *r7* will be copied into *r5*, otherwise *r6* is copied into *r5* as described in Section 2.2.

Processor Models. Three processor models are evaluated this paper. The baseline processor is a k-issue processor, with no limitation placed on the combination of instruc-

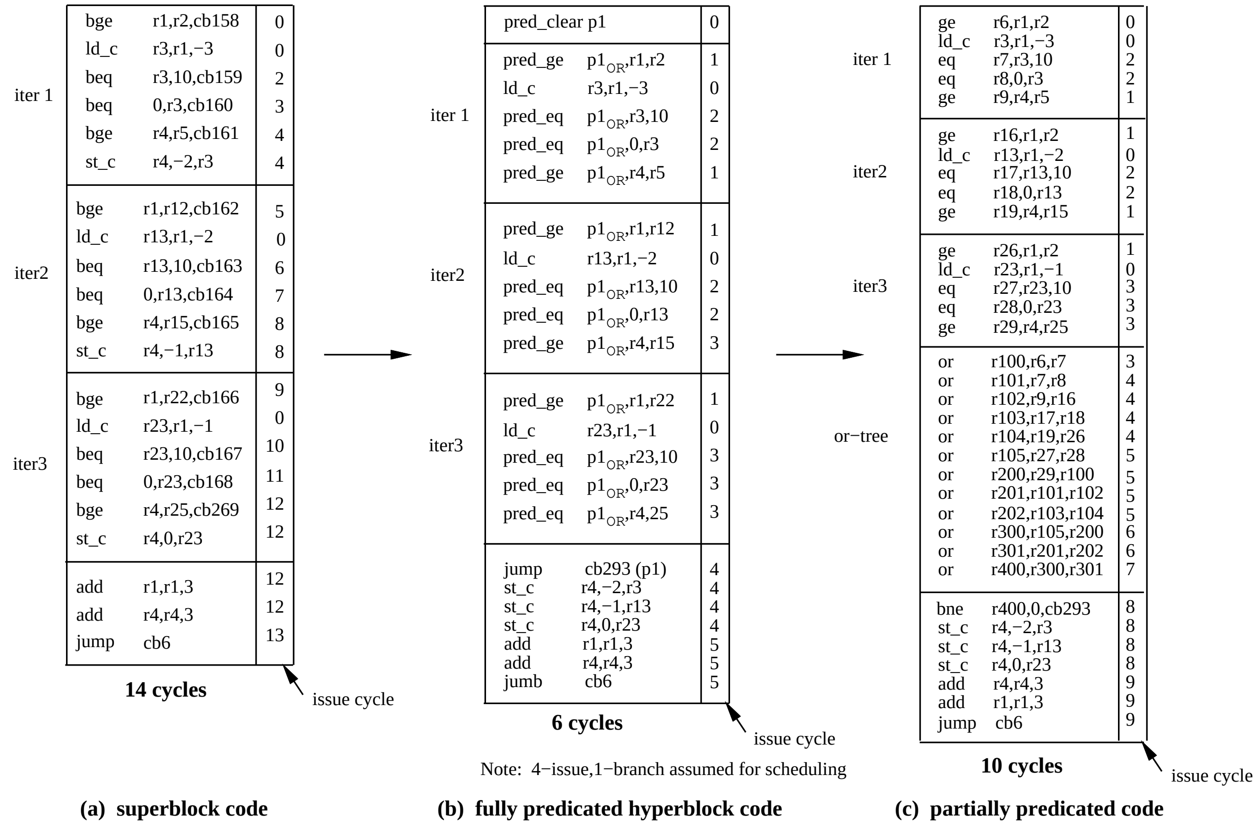


Figure 6: Example loop segment from *grep*.

tions which may be issued each cycle, except for branches. The memory system is specified as either perfect or consists of a 64K directed mapped instruction cache and a 64K direct mapped, blocking data cache; both with 64 byte blocks. The data cache is write-through with no write allocate and has a miss penalty of 12 cycles. The dynamic branch prediction strategy employed is a 1K entry BTB with 2 bit counter with a 2 cycle misprediction penalty. The instruction latencies assumed are those of the HP PA-RISC 7100. Lastly, the baseline processor is assumed to have an infinite number of registers. The baseline processor does not support any form of predicated execution. However, it includes non-excepting or silent versions of all instructions to fully support speculative execution. Superblock ILP compilation techniques are utilized to support the baseline processor [20]. The baseline processor is referred to as *Superblock* in all graphs and tables.

For partial predicate support, the baseline processor is extended to support conditional move instructions. Note that since non-excepting versions of all instructions are available, the more efficient conversions are applied by the compiler for partial predication (Section 3.2). The partial predicate support processor is referred to as *Conditional Move*. The final model is the baseline processor extended to support full predication as described in Section 2.1. This model is referred to as *Full Predication*. For this model, hyperblock compilation techniques are applied. Performance of the 3 models is compared by reporting the speedup of the particular processor model versus the baseline processor. In

(1) pred_eq	$p1_{OR}$, $p3_{\overline{U}}$,r24,0 (p2)	DEPI	0,3,1,%r3
		BB,>=,N	%r3,2,$pred_0
		COMCLR,=	%r0,%r24,%r0
		DEPI,TR	1,3,1,%r3
		DEPI	1,1,1,%r3
		$pred_0	
(2) add	r25,r25,1 (p3)	EXTRU,EV	%r3,3,1,%r0
		ADDI	1,%r25,%r25
(3) cmov	r5,r6,r8	COMCLR,=	%r8,%r0,%r0
		COPY	%r6,%r5
(4) select	r5,r6,r7,r8	COMCLR,=	%r8,%r0,%r0
		OR,TR	%r6,%r5
		COPY	%r7,%r5

Figure 7: HP PA-RISC emulation of predicate support.

particular, speedup 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

Figure 8 shows the relative performance achieved by superblock, conditional move, and full predication for an issue-8, 1-branch processor. Full predication performed the best in every benchmark with an average speedup of 63% over superblock.[1] Speedup with conditional move code fell between superblock and full predication for all benchmarks except *072.sc* which performed slightly below superblock

[1]Averages reported refer to the arithmetic mean.

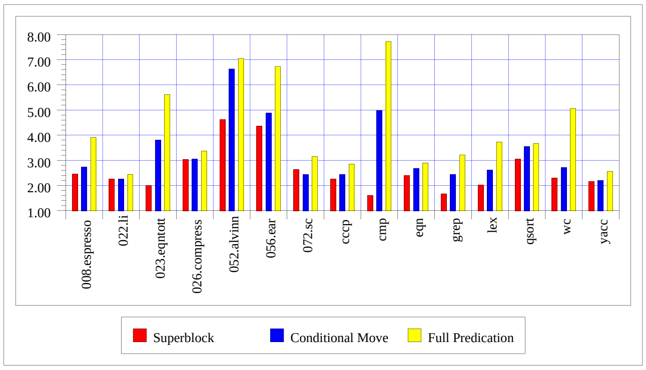


Figure 8: Effectiveness of full and partial predicate support for an 8-issue, 1-branch processor with perfect caches.

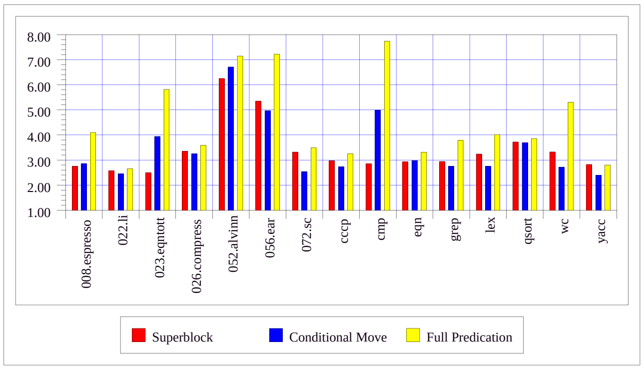


Figure 9: Effectiveness of full and partial predicate support for an 8-issue, 2-branch processor with perfect caches.

code's performance. The unusual behavior of *072.sc* was primarily due to increased dependence chain lengths caused by the conditional move transformations. On average, though, conditional move code had a speedup of 33% over superblock. The speedup for conditional move code is very substantial. Most researchers and product developers have reported small gains except for certain special cases with conditional moves. However, utilizing the hyperblock techniques in conjunction with the conditional move transformations yields consistent performance improvements.

Full predication also achieved performance gain on top of the conditional move model. This illustrates that there is significant performance gain possible provided by the ISA changes to support full predication. In particular, the efficiency of representing predicated instructions, the reduced dependence heights to represent predicated instructions, and the ability to simultaneously execute *OR* type predicate defines provided full predicate support with the additional performance improvement. On average, a gain of 30% over the conditional move model was observed.

Increasing the branch issue rate from 1 to 2 branches per cycle provides interesting insight into the effectiveness of predicated execution. Figure 9 shows the performance result of an 8-issue processor that can execute 2 branches per cycle. The performance improvement of conditional move code and full predicate against superblock code is reduced. This is attributal to improving the performance of superblock. The

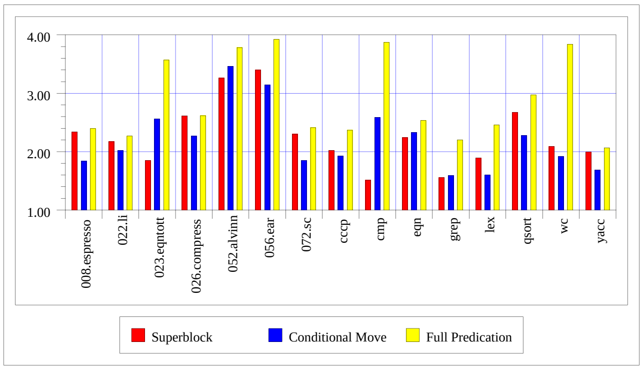


Figure 10: Effectiveness of full and partial predicate support for an 4-issue, 1-branch processor with perfect caches.

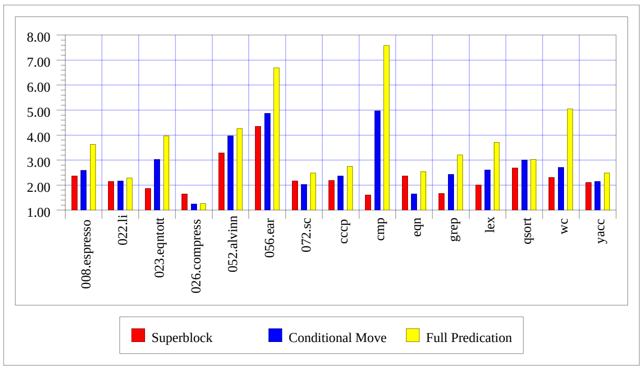


Figure 11: Effectiveness of full and partial predicate support for an 8-issue, 1-branch processor with 64K instruction and data caches.

conditional move and full predication code has had many of the branches removed with hyperblock formation. Therefore, increasing the number of branches does not noticeably improve the performance of conditional move and full predication code. On average, conditional move performed only 3% faster than superblock while full predication performed 35% faster than superblock.

Figure 10 shows performance of the benchmarks on a 4 issue processor that can issue 1 branch per cycle. The most noticeable trend across these benchmarks is that while full predication consistently beats superblock code, conditional move code performs worse than superblock in the majority of benchmarks. Since support for predication in the condition move code is limited, the compiler must compensate by creating many more instructions than it would with full predicate support. These extra instructions are absorbed by the 8 issue machine, but saturate the 4 issue machine creating poor results. These results indicate a more conservative hyperblock formation algorithm needs to be employed for the conditional move model with a 4-issue processor. For full predication, substantial performance gain is still possible for the 4-issue processor, with an average of 33% speedup over superblock.

To evaluate the cache effects associated with predicated execution, Figure 11 is presented. As expected all three methods were affected by a realistic cache model. However,

Benchmark	Superblk	Cond. Move	Full Pred.
008.espresso	489M	812M (1.66)	626M (1.28)
022.li	31M	38M (1.23)	32M (1.04)
023.eqntott	1030M	1230M (1.19)	885M (0.86)
026.compress	90M	128M (1.41)	108M (1.20)
052.alvinn	3574M	4003M (1.12)	3603M (1.01)
056.ear	11225M	13838M (1.23)	11073M (0.99)
072.sc	91M	85M (0.93)	75M (0.83)
cccp	3701K	5077K (1.37)	3855K (1.04)
cmp	932K	1422K (1.53)	922K (0.99)
eqn	44M	49M (1.11)	44M (0.99)
grep	1282K	2467K (1.92)	1647K (1.28)
lex	36M	75M (2.10)	46M (1.29)
qsort	44M	70M (1.61)	49M (1.11)
wc	1493K	2999K (2.01)	1526K (1.02)
yacc	43M	66M (1.53)	50M (1.16)

Table 2: Dynamic instruction count comparison.

two benchmarks stand out. The real cache significantly reduced the performance of *026.compress* in all three models. Conditional move and full predication code increased data memory traffic more by performing speculative execution using predicate promotion. Since these promoted instructions often caused cache misses, the performance of conditional move and full predication code dropped significantly. *Eqn* also exhibited an interesting result. Conditional move performed poorly while full predication and superblock remained proportionally the same. This is a side effect of the increased instruction cache miss rate due to conditional move's larger instruction count. This is evidenced by the dynamic instruction count of *eqn* in Table 2. On average predicated code still yielded good results over superblock with full predication performing 54% faster than superblock and conditional moves performing 24% faster than superblock.

The dynamic instruction count of all benchmarks with respect to a processor model is shown in Table 2. Full predication code can increase dynamic instruction count over superblock as is executes both paths of an *if-then-else* construct. Superblock code can increase dynamic instruction count over full predication by unnecessary speculated instructions into frequently executed paths. Therefore the overall relation in instruction count between full predication and superblock can vary as the results indicate. Conditional move code's dynamic instruction count is hit hardest; however, since it suffers from executing code normally hidden by branches combined with the inefficiencies associated with not having full predicate support. Conditional move code had an average of 46% more dynamic instructions than superblock, while full predication had only 7% dynamic instruction.

Finally, as shown in Table 3, the number of branches in partially and fully predicated code is substantially less than in the superblock code. Much of the speedup of full and partial predication comes from the elimination of branches. Mispredicted branches incur a significant performance penalty. With fewer branches in the code, there are fewer mispredictions. Also, in many architectures, because of the high cost of branch prediction, the issue rate for branches is less than the issue rate for other instructions. Therefore, fewer branches in the code can greatly increase the available ILP. Partially and fully predicated code have very close to the same number of branches, with fully predicated code often having just a few less. The small difference in the number of branches is a result of adding branches around predicated subroutine calls in partially predicated code. The differences in the misprediction ratios for partially and fully predicated code is also a result of predicated subroutine calls.

An odd behavior is observed for *grep* in Table 3. The number of mispredictions for the conditional move and full predication models are larger than that of the superblock model. This is caused by a branch combining transformation employed for hyperblocks by the compiler which is heavily applied for *grep*. With this transformation, unlikely taken branches are combined to a single branch. The goal of the transformation is to reduce the number of dynamic branches. However, the combined branch typically causes more mispredictions than the sum of the mispredictions caused by the original branches. As a result, the total number of mispredictions may be increased with this technique.

5 Concluding Remarks

The code generation strategy presented in this paper illustrates the qualitative benefit of both partial and full predication. In general, both allow the compiler to remove a substantial number of branches from the instruction stream. However, full predication allows more efficient predicate evaluation, less reliance on speculative execution, and fewer instructions executed. As shown in our quantitative results, these benefits enable full predication to provide more robust performance gain in a variety of processor configurations.

For an eight issue processor that executes up to one branch per cycle, we show that conditional move allows about 30% performance gain over an aggressive base ILP processor with no predication support. This speedup is very encouraging and shows that a relatively small architectural extension can provided significant performance gain. Full predication offers another 30% gain over conditional move. The performance gains of full and partial predication support illustrate the importance of improving branch handling in ILP processors using predicated execution.

Results based on a four issue processor illustrate the advantage of full predication support. Full predication support remains substantially superior even in the presence of a low issue rate. This is due to its efficient predicate evaluation and low instruction overhead. This contrasts with conditional move support where the extra dynamic instructions over utilize the processor issue resources and result in a sizable performance degradation for the majority of the benchmarks. Nevertheless, the substantial performance gain for two of the benchmarks suggests that conditional move could be a valuable feature even in a low issue rate processor. However, this does indicate that a compiler must be extremely intelligent when exploiting conditional move on low issue rate processors.

All of the results presented in this paper are based on a 2-cycle branch prediction miss penalty. This was chosen to show conservative performance gains for predicated execution. For machines with larger branch prediction miss penalties, we expect the benefits of both full and partial prediction to be much more pronounced. Furthermore, when more advanced compiler optimization techniques become available,

Benchmark	Superblock			Conditional Move			Full Predication		
	BR	MP	MPR	BR	MP	MPR	BR	MP	MPR
008.espresso	75M	3402K	4.55%	38M	2066K	5.38%	33M	1039K	3.15%
022.li	7457K	774K	10.38%	6169K	694K	11.25%	6110K	702K	11.5%
023.eqntott	315M	42M	13.47%	53M	6732K	12.66%	51M	6931K	13.57%
026.compress	12M	1344K	10.9%	9269K	864K	9.32%	9240K	867K	9.38%
052.alvinn	463M	1091K	0.24%	74M	896K	1.23%	74M	1032K	1.38%
056.ear	1539M	66M	4.3%	443M	16M	3.52%	442M	15M	3.4%
072.sc	22M	1232K	5.49%	11M	1044K	9.19%	11M	934K	8.26%
cccp	921K	66K	7.19%	537K	65K	12.17%	534K	65K	12.15%
cmp	530K	4395	0.83%	26K	31	0.12%	26K	31	0.12%
eqn	7470K	1612K	8.2%	4506K	514K	11.4%	4495K	511K	11.37%
grep	663K	9660	1.46%	171K	20K	11.7%	171K	20K	11.73%
lex	14M	232K	1.65%	3070K	201K	6.55%	3030K	196K	6.46%
qsort	8847K	1332K	15.06%	6092K	597K	9.79%	6066K	610K	10.06%
wc	478K	33K	6.85%	224K	57	.025%	224K	57	.025%
yacc	12M	517K	4.31%	5944K	445K	7.48%	5900K	431K	7.31%

Table 3: Comparison of branch statistics: number of branches (BR), mispredictions (MP), and miss prediction rate (MPR).

we expect the performance gain of both partial and full predication to increase. We also feel it would be interesting to explore the range of predication support between conditional move and full predication support.

Acknowledgements

The authors would like to thank Roger Bringmann and Dan Lavery for their effort in helping put this paper together. We also wish to extend thanks to Mike Schlansker and Vinod Kathail at HP Labs for their insightful discussions of the Playdoh model of predicated execution. Finally, we would like to thank Robert Cohn and Geoff Lowney at DEC, and John Ruttenberg at SGI for their discussions on the use of conditional moves and selects. This research has been supported by the National Science Foundation (NSF) under grant MIP-9308013, Intel Corporation, Advanced Micro Devces, Hewlett-Packard, SUN Microsystems and AT&T GIS.

References

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

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

[3] 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.

[4] M. D. Smith, M. Johnson, and M. A. Horowitz, "Limits on multiple instruction issue," in *Proceedings of the 3rd International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 290–302, April 1989.

[5] D. W. Wall, "Limits of instruction-level parallelism," in *Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 176–188, April 1991.

[6] M. Butler, T. Yeh, Y. Patt, M. Alsup, H. Scales, and M. Shebanow, "Single instruction stream parallelism is greater than two," in *Proceedings of the 18th International Symposium on Computer Architecture*, pp. 276–286, May 1991.

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

[8] 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.

[9] 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.

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

[11] 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.

[12] S. A. Mahlke, R. E. Hank, R. A. Bringmann, J. C. Gyllenhaal, D. M. Gallagher, and W. W. Hwu, "Characterizing the impact of predicated execution on branch prediction," in *Proceedings of the 27th International Symposium on Microarchitecture*, pp. 217–227, December 1994.

[13] 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.

[14] 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.

[15] V. Kathail, M. S. Schlansker, and B. R. Rau, "HPL playdoh architecture specification: Version 1.0," Tech. Rep. HPL-93-80, Hewlett-Packard Laboratories, Palo Alto, CA 94303, February 1994.

[16] M. Schlansker, V. Kathail, and S. Anik, "Height reduction of control recurrences for ILP processors," in *Proceedings of the 27th International Symposium on Microarchitecture*, pp. 40–51, December 1994.

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

[18] D. S. Blickstein *et al.*, "The GEM optimizing compiler system," *Digital Technical Journal*, vol. 4, pp. 121–136, 1992.

[19] P. G. Lowney *et al.*, "The Multiflow trace scheduling compiler," *The Journal of Supercomputing*, vol. 7, pp. 51–142, January 1993.

[20] W. W. Hwu *et al.*, "The Superblock: An effective technique for VLIW and superscalar compilation," *The Journal of Supercomputing*, vol. 7, pp. 229–248, January 1993.