Incorporating Guarded Execution into Existing Instruction Sets

by

Dionisios N. Pnevmatikatos

A dissertation submitted in partial fulfillment

of the requirements for the Degree of

Doctor of Philosophy

(Computer Sciences)

at the

UNIVERSITY OF WISCONSIN – MADISON

1996

Incorporating Guarded Execution into Existing Instruction Sets

Dionisios N. Pnevmatikatos
UNIVERSITY OF WISCONSIN – MADISON, 1996
Under the supervision of Gurindar Sohi

Guarded execution, or simply guarding, is a powerful and promising concept, with the potential to reduce the unpredictability of the control flow caused by branches, and smoothen the flow of instructions in processor pipeline(s). Guarding also boosts the compiler's ability to expose instruction level parallelism to the processor, while requiring a modest amount of hardware support. These features make guarding attractive for inclusion in an architecture. However, the integration of guarding in an instruction set is not easy, especially when the designer needs to extend an existing instruction set. This thesis address two issues that are critical to the widespread acceptance for guarding: (i) the required instruction set support for guarded instructions, and (ii) the performance on aggressive processor configurations.

This thesis proposes GUARD instructions, a new class of instructions that offer an easy and powerful way to accommodate guarded execution in an instruction set. With the modest requirement of just a few opcodes, GUARD instructions are sufficient to provide efficient support for full guarding.

This thesis evaluates and compares the performance of three ways of supporting guarding: (i) using explicit guard operand fields in each instruction, (ii) using conditional move instructions, and (iii) the newly proposed GUARD instructions. The results of this evaluation show that for all configurations, GUARD instructions perform better than ordinary guarding. They also show that conditional moves can exploit a large fraction of the potential of full guarding, and that hardware mechanisms such as 2-level adaptive branch prediction and out-of-order execution diminish the performance potential of guarded execution.

Dedicated to my sister Maria Christina, who left us too soon

Αφιερωμένο στην αδερφή μου Μαρία Χριστίνα, που μας άφησε πολύ νωρίς

Acknowledgments

Six and a half years and one winter too many in Madison. Thousands of hours of simulations, thousands of lines of code, and even more numbers to gather and interpret. All this work would not have been possible if it weren't for four people: my advisor Guri Sohi who taught me everything I know about research and paper writing, my wife Natalia who loved me, supported me and inspired me more than I could ever ask for, and my parents Nikos and Katerina who made everything in their power to provide me with the best education, even if it meant that we had to be separated. The unconditional support and encouragement of these four people kept me cruising along the long haul of the doctoral dissertation, and for that I will always be grateful. This thesis is as much a tribute to their efforts, as it is to mine.

I would like to thank the members of my thesis committee, Jim Goodman, Jim Smith, David Wood and Parmeswaran Ramanathan for their patience in reading my thesis and for their comments that made this thesis more accurate, more complete and easier to read. I would also like to thank Manolis Katevenis who taught me how much fun computer architecture can be, and Mark Hill who taught me pretty much everything there is to know about caches and introduced me to applied research (see 10+ day long simulations!).

I would also like to thank the members of the multiscalar group: Todd Austin, Scott Breach, Manoj Franklin, Andreas Moshovos, and T. N. Vijaykumar, and other students in architecture: Stefanos Kaxiras, Babak Falsafi with whom I had discussions about everything and anything, even about computer architecture! Todd Austin deserves a special mention both for our close cooperation on the Fast Address Calculation paper, and for his SimpleScalar simulation tools that I used in this thesis.

My life in Madison has been enriched by a number of new close friends: Dimitris Cholevas, Andreas Moshovos, Kyros Koutoulakos, Stefanos Kaxiras, Minos Garofalakis, Lakis Karmirantzos, Spyros Kontogiorgis, Yannis Christou and Yannis Schoinas. Guys, I should have ranked you according to coffee-break performance! Other people that I would like to acknowledge are Yannis Ioannidis, Giorgos Drettakis, Dimitra Vista, and all the members of the Hellenic Student Association of UW–Madison.

I would also like to thank the Wisconsin State Government agencies who's phones are 266-18xx, for making the simple task of answering the phone much more interesting and exciting. I think I will always remember the first "Office of Probation and Parole", "Sargent in Arms" and "Judge Wilcox Chambers" phone calls.

DIONISIOS N. PNEVMATIKATOS

UNIVERSITY OF WISCONSIN – MADISON
May 1996

Contents

List of Tables

List of Figures

Chapter 1
Introduction

In the quest for ever-faster processors, and with a considerable transistor budget per chip, processor designers rely heavily on Instruction Level Parallelism (ILP) to achieve higher performance. Current high-end processors can issue up to 4 or more instructions per cycle, while mainstream processors already can issue two or three instructions per cycle. Meanwhile, efforts to achieve higher clock frequencies force designers to split tasks that traditionally occupied a single pipeline stage (such as instruction decoding) into multiple stages, deepening the processor pipelines.

In these deep and wide pipelines, a continuous instruction supply is critical to sustain a high issue rate. Changes in the control flow of a program can introduce bubbles in the pipeline, resulting in underutilization of the resources, and longer execution times. Furthermore, conditional branches introduce a dependency between the execution of the branch instruction and the fetching of the target instruction. In a deep pipeline, resolving this dependency and fetching the correct target can take several cycles, during which the (multiple) resources of the processor remain idle.

Branches also introduce control dependences which restrict the ability of the compiler to rearrange instructions and achieve better instruction schedules. Control dependent computation (that is, computation who's execution depends on a branch outcome) cannot, in general, be moved across branches, unless it is safe, i.e., unless the compiler can guarantee that the computation will never cause an exception, or unless adequate hardware support is provided to buffer the exceptions until the condition is resolved (e.g., sentinel scheduling [MCH$^+$92], or boosting [SLH90, SHL92]).

1.1 What can we do about branches?

A number of solutions have been proposed to deal with the limitations of branches. The general idea behind many techniques is to predict a *likely* direction for a branch and optimize the execution along that path. This general idea can used both statically during code generation, and dynamically using some information tracking mechanism.

A static manifestation of this general framework is static branch prediction, in which branches are tagged with a direction bit; this bit indicates to the processor the likely branch outcome. Static predictions can also be used by the compiler for scheduling purposes, as in the case of trace scheduling and its variants [Fis81, LFK$^+$93, CNO$^+$88, CMC$^+$91]. In trace scheduling, the compiler predicts a branch and optimizes the code assuming the prediction is correct. To recover the correct state after an incorrect prediction, the compiler generates special fix-up code that will be executed when an incorrect prediction is detected. This fix-up code is responsible for reversing all the effects of the code that was incorrectly executed.

Dynamic branch prediction utilizes information collected at run time to decide which is the most likely direction for a branch. Generally, dynamic branch prediction is based on maintaining a table of counters that record the past behavior of branches [Smi81, YP92, YP93] and a selection mechanism (called a *divisor* by Young *et al.* [YGS95]) that will determine which counter to use for the prediction of each branch. Because the direction of a branch is not by itself sufficient to allow the instruction fetch mechanism to commence fetching new instructions, a Branch Target Buffer is also used to cache branch target address [LS84]. Dynamic branch prediction techniques are significantly more accurate than static ones, and due of their simple, table-based structure, are relatively easy to implement. As a result, virtually every new processor resorts to some form of dynamic branch prediction.

Even the most accurate prediction mechanisms face performance limitations. The behavior of some branches is unpredictable, forcing a lengthy recovery action to be employed by the processor. In addition, branches restrict the ability of the compiler to schedule the code, making it harder for the execution hardware to discover the available parallelism.

One way to alleviate these limitations, is to use *if-conversion* [AKPW83] and *guarded execution* [Hsu86, HD86, Mac93, MLC^{+}92, RYYT89, KSR94]. In guarded execution, instructions in conditional regions of code are augmented with a guard operand. This operand specifies whether the instruction should be executed or not, and the branch controlling the execution of that region can be removed from the code. As a result, the compiler or dynamic scheduler can rearrange these instructions arbitrarily, as long as the dependences are respected. Furthermore, regardless of the direction of the branch, the control flow is always sequential through the if-converted part of the code, allowing a fast, sequential instruction fetching, and eliminating the pipeline bubbles due to branches. Since the if-converted branches are not predicted, they never require corrective actions. However, these advantages come at some cost. The guarded instructions that the compiler generates may be dynamically converted into NOPs, in which case they will not contribute to the useful computation of the program. However the processor will still have to fetch and decode them, possibly preventing other useful instructions from executing.

1.2 Guarding Background

Guarded execution (or simply *guarding*), has been proven effective in several contexts. Originally, it was proposed by Dijkstra [Dij75] as a high level programming construct, meant to simplify the expression of algorithms and to allow the formal verification of the derived algorithms. Dijkstra allowed an arbitrary Boolean conditions to be used as the guard condition of a high level statement.

Vector processors such as the CDC STAR-100 [HT72], TI ASC [Wat72] and Cray-1 [Rus78] have long benefited from guarded execution expressed through the use of *vector masks*. A vector mask is an N-bit wide control register, where N is the number of elements in a vector register. Each bit in the vector mask controls whether the corresponding element of a destination vector register should be updated or not. Essentially the vector mask stores multiple guard conditions. Using a vector mask, loops that contain if-statements can be vectorized by breaking the loop in two phases: one in which vector instructions are used to compute the bits of the vector mask, and one in which vector instructions perform the necessary computation, with the newly computed vector mask masking the results that should not be written to their destinations. The Cray-1 also includes a scalar version of the vector mask, and a *merge* instruction which is a superset of the conditional move instruction.

Allen *et al.* formalized and generalized the idea of vector masks introducing *if-conversion* [AKPW83], a general code transformation technique that facilitates the vectorization of floating point intensive applications. As part of PFC, the Parallel Fortran Converter, if-conversion was responsible for the simplification of the control flow of loops with conditional statements by converting the control dependences introduced by branches into data dependences, which are easier for the compiler to handle. The overall operation of PFC is to convert the loops into the two phases described for use with a vector mask: first compute the appropriate conditions, and then do the actual computation guarded with the appropriate condition. A later phase of PFC was responsible for mapping these converted loops into vector instructions using the vector masks if they were supported by the underlying architecture.

VLIW machines, for example Cydra-5 [RYYT89, DHB89], and the IBM VLIW machine [Ebc88], have also used if-conversion and guarded execution to facilitate the software pipelining of loops with conditional branches. The general methodology is to apply if-conversion to loop bodies so they become free of control dependences, and then to apply other loop scheduling technique such as software pipelining [RG91, BRRP82, Lam88, Ebc87]. These machines provided instruction set and hardware support for guarded execution. For example, the Cydra-5 contained a *predicate* field per operation, and the IBM VLIW machine supported *tree* instructions, which would evaluate an expression tree and depending on the exact conditions would nullify the appropriate operations along the branches of the tree.

Hsu and Davidson [Hsu86, HD86] used guarding on a scalar processor to allow better scheduling of decision trees. In the context of a decision tree, the conditional branches are essential because they steer the flow of control to the correct branch of the tree. These diverging control structures are not amenable to if-conversion, and guarding was used as a general purpose technique to fill multiple architectural branch delay slots. However, the only current commercial scalar instruction set to include support for guarded execution is the ARM instructions set [Mac93].

The Illinois IMPACT group [CMC$^+$91, MCH$^+$92, MLC$^+$92, MHB$^+$94, MHM$^+$95] has developed several compilation techniques to improve the performance of guarded execution for ordinary programs. The main compilation construct they use is the *Hyperblock*, which is a compilation region that may contain ordinary, as well as guarded instructions. A hyperblock is defined to have a single entry and may have multiple exits. The IMPACT compiler performs an extensive set of code transformation techniques, such as loop restructuring, unrolling, loop peeling, etc. in order to enlarge the amount of computation that can be encapsulated in a hyperblock. The compiler chooses the most appropriate code transformation technique using profiling information.

1.3 Contributions of this thesis

Why isn't guarded execution a part of many instructions set architectures? The answer is not simple; however, we can identify two key factors that hinder the general acceptance of guarding:

- It requires substantial support in the instruction set, for guard operands and new instruction specifiers. Guarding requires new opcodes and instruction formats, and perhaps additional registers or register files to hold guard conditions. While this support can be trivially added to a new instruction set, extending an existing one is considerably harder. In an existing instruction set the unused space in the instruction encoding is very limited, especially for instructions with immediate fields such as load and store instructions.

- The expected performance in an ordinary development environment is unknown. Most research in the area of guarding has been done for numerical intensive programs, or under very controlled environments, where plenty of information about the compiled programs is available. Under every-day use, such a controlled environment may not be possible.

This thesis focuses on these two practical issues concerning guarded execution. Our contribution is twofold.

First, we propose a new way of supporting guarding which requires only small additions to an existing (or new) instruction set architecture (ISA). This is achieved through a new class of instructions, the GUARD instructions. GUARD instructions augment the semantics of ordinary computation to add the guard operands. GUARD instructions can also reduce the number of instructions required to perform condition evaluation and can sustain better performance than other methods for guarding.

Second, we identify a set of guarding alternatives for existing instruction sets and we evaluate their performance potential. We perform the evaluation in two steps. In the first step, we evaluate the impact of guarding on the dynamic characteristics (such as the instruction count and branch behavior) of the programs; these characteristics provide insight in the effectiveness of guarding. In the second step, we perform detailed simulations to determine the execution time for each of our guarding alternatives. The context of the evaluation is a MIPS-like processor. We carry out the evaluation assuming varying levels of support for guarding, using either the traditional specification method, or using GUARD instructions.

The thesis is organized as follows: Chapter 2 describes in detail guarded execution, its use and potential, its hardware support requirements and its ISA requirements. Chapter 3 describes the compiler support required for the effective use of guarded execution. Chapter 4 introduces GUARD instructions, a mechanism to allow the introduction of guarding in existing instruction sets, and discusses the required changes in the instruction set architecture level and at the execution hardware level. Chapter 5 presents the effect of guarding on the dynamic program characteristics. Chapter 6 presents the execution time evaluation of guarding, and Chapter 7 concludes the thesis and summarizes the implications of this work.

Chapter 2
Guarded Execution

Guarded execution is the use of guarded instructions to express conditional program structures. In this chapter, we define the exact semantics and the key characteristics of guarded instructions, and we briefly describe *if-conversion*, a method that uses guarded instructions to express conditional computation. Then we discuss the advantages and limitations of using guarding execution and we discuss the implications of supporting guarded execution on the instruction set architecture and on the implementation of the execution engine.

2.1 Semantics and use of guarded instructions

A guarded instruction is a ordinary instruction augmented with a guard condition specifier. The semantics of a guarded instruction are as follows: evaluate the guard condition and if it evaluates to true, then execute the instruction, otherwise treat the instruction as a NOP. For example, a guarded add instruction can be written as:

```
g_add gcond ?  dst, src1, src2
```

and its semantics would be: if `gcond` evaluates to true, perform the addition of `src1` and `src2` into `dst`, otherwise leave `dst` unmodified.

The conditional nature of guarded instructions allows the expression of conditional computation structures without the use of conditional branches, since the guard operands will make sure that only the correct results are committed to the architectural state of the processor.

2.1.1 Use of guarded instructions

To understand why guarded execution can be a useful feature in a processor, let us first consider how a program is compiled and executed. A common program representation form used by compilers is a control flow graph (CFG). A control flow graph is a directed graph where each node is a basic block and each arc corresponds to the possible flow of control through that node. The compilation of a program can be thought of as a set of traversals of the CFG during which instructions are rearranged. During code generation, the nodes of the CFG are placed in a linear fashion in the program memory. The execution of a program can be thought of as a traversal of the linearized CFG during which the processor decides which instructions must be executed. The existence of conditional structures in the CFG gives rise to *control dependences*. A CFG node X is said to be control dependent on CFG node Y, when (i) Y has two (or more) outgoing arcs, and (ii) when the flow of control reaches node Y, node X may or may not be executed, according to the condition that selects the outgoing arc of node Y.

Figure 2.1 illustrates the control dependences in a CFG. Figure 2.1(a) shows the C-code for a small loop containing an if-then-else statement. Figure 2.1(b) shows the corresponding control flow graph, annotated (using dashed lines) with the control dependences. In this example node 2 and 3 are

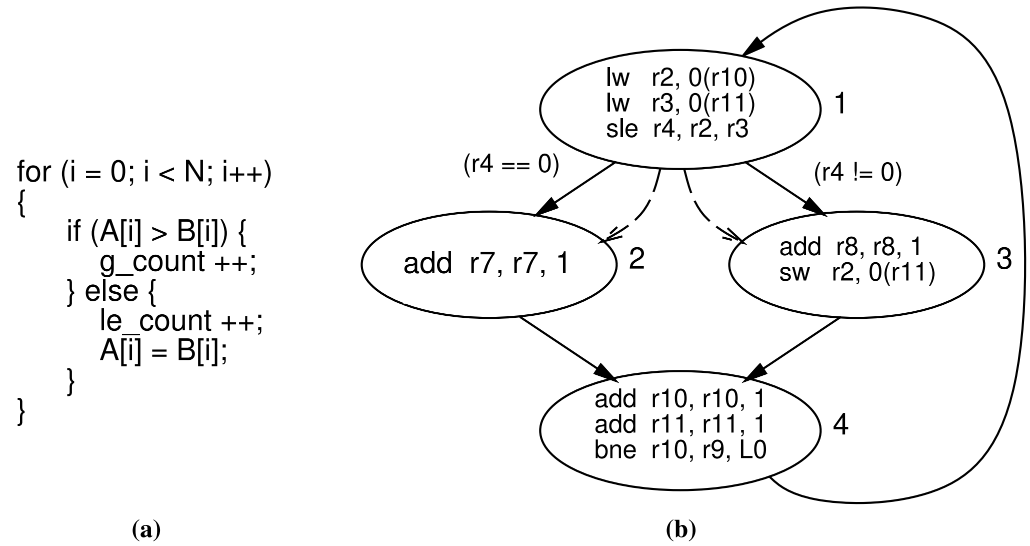


Figure 2.1: A small loop and its corresponding CFG. The dashed arrows in the CFG indicated control dependences.

control dependent on node 1. Note that node 4 is not control dependent on node 1, since node 4 will be executed regardless of the outcome on the branch in node 1. Node 4 is *not* control dependent on nodes 2 and 3 since these nodes have a single exit arc.

In a traditional instruction set, the conditional structures in the CFG are expressed in the assembly code using conditional branches to express the decision points (node 1) and unconditional branches to redirect the control flow to the reconvergence points (node 4) as shown in Figure 2.2(a). During the execution of the generated code, the processor navigates through the CFG by means of executing the conditional and unconditional branches.

If the instruction set supports guarded instructions, a process known as *if-conversion* can be used to express the conditionally executed parts of the CFG using guarded instructions as shown in Figure 2.2(b). (In Figure 2.2(b), the prefix "g_" denotes the guarded version of an instruction, so g_add is a guarded add, g_sw is a guarded store, etc; the first operand of a conditional instruction is the condition register.) If-conversion works on loop-free subsets of the CFG of a program in three steps. First, it assigns a guard condition register to each conditionally executed node. Second, it identifies where each guard condition should be set and inserts code that sets it appropriately. As a last step, if-conversion transforms all the instruction in the conditionally executed nodes into guarded instructions using the corresponding guard condition. (A detailed description of an algorithm for if-conversion will be presented in Chapter 3.) After these three steps, the control dependences are transformed into data dependences which are explicitly expressed as the settings and the uses of the guard registers.

The positive effects of guarded execution can be seen in Figure 2.2 (a) and (b). Comparing the two figures we find that two static branches were eliminated (corresponding to the if-then-else construct in the C-code), and that the basic blocks are considerably larger: the MIPS-like assembly contains eight non-branch and three branch instructions, while the guarded version contains 8 non-branch and 1 branch instructions. The instruction level parallelism is increased considerably with guarding. In Figure 2.2(a), the maximal dependence path is five or six instructions per iteration depending on

```
L0: lw    r2, 0(r10) # A[i]
    lw    r3, 0(r11) # B[i]   r2
    sle   r4, r2, r3
    bne   r4, r0, L1          r4
    add   r7, r7, 1           CT
    j     L2
L1: add   r8, r8, 1           CT
    sw    r2, 0(r11)
L2: add   r10, r10, 1
    add   r11, r11, 1         r10
    bne   r10, r9, L0
```

(a)

```
L0: lw              r2, 0(r10)  # A[i]
    lw              r3, 0(r11)  # B[i]   r2
    sle             r4, r2, r3
    g_add    !r4?   r7, r7, 1            r4
    g_add     r4?   r8, r8, 1
    g_sw      r4?   r2, 0(r11)
    add             r10, r10, 1
    add             r11, r11, 1          r10
    bne             r10, r9, L0
```

(b)

Figure 2.2: MIPS-like assembly for the code in Figure 2.1. Conditional computation is implemented using branches in part (a) and guarded instructions in part (b). The arrows indicate the worst dependence path(s) through the code, and are tagged with the register that causes the dependency. The CT tag stands for "Control Transfer".

whether the conditional branch is taken or not. (The arrows in the right of the assembly code indicate the worst path corresponding to the not-taken case.) In Figure 2.2(b), the maximal dependence is three instructions per iteration. Assuming adequate resources, data cache hits and equal probability for each direction of the conditional branch, the execution the non-guarded version of the code will take 6.5 cycles on the average (the two possible paths are 5 and 6 instructions long plus one cycle for the load latency), while the execution of the guarded version of the code will take 4 cycles (a critical path of three instructions plus one cycle for the load latency), corresponding to a 62% speedup.

2.1.2 Advantages of guarding

The previous example illustrated that a seemingly small difference, the conversion of control dependences into data dependences, had a significant impact on the execution time of the code. The reason is that control dependences have a negative effect both on the compiler and on the processor which was relieved by the conversion.

When if-conversion is used, several basic blocks of the CFG are merged together. The combined basic block contains more instructions and presents more scheduling opportunities and flexibility. A compiler for an instruction level parallel processor generally has to first identify the set of instructions that are independent, and then schedule them so they are executed in parallel. Under realistic assumptions, the processor resources are finite and the compiler has the additional task of scheduling resource usage among the competing instructions. The larger basic blocks after if-conversion generally contain larger amounts of parallelism, permitting the compiler to produce instructions schedules that are more parallel and match the processor resources better.

Furthermore, after if-conversion, instructions from different control paths can be freely intermixed, allowing instruction schedules that overlap different control paths during execution, a flexibility that is shown to have significant performance potential; Lam and Wilson [LW92] found that allowing the execution of multiple flows of control almost tripled the amount of parallelism exposed to their abstract machine models. In essence, the ability to overlap different control paths is similar to *global scheduling* techniques [Nic85, Smi92]. The removal of control dependences also can increase the effectiveness of other code transformation techniques such as software pipelining [RG91, BRRP82, Lam88, Ebc87], modulo scheduling [Rau94], etc.

For the instruction fetch unit of a processor, the larger, if-converted basic blocks allow for higher instruction fetch efficiency. Small basic blocks cause frequent changes in the flow of control, resulting in under-utilization of the instruction fetch bandwidth and the processor resources. The execution of if-converted straight-line code benefits from an instruction memory system that provides high bandwidth, but not necessarily short latency. High bandwidth from the instruction memory to the processor, is relatively cheap for single chip implementations, since the processor can take advantage of a wide on-chip instruction cache. Furthermore, prefetching can be used to shorten the latency due to instruction cache misses during sequential parts of instruction fetch.

Processors employing dynamic branch prediction, can also benefit from guarding. For programs with small basic blocks, multiple predictions per cycle may be required to find enough instructions and to fully utilize the pipeline resources [YMP93]. The larger basic blocks of the if-converted code keep the pipeline more full, reducing the importance of multiple predictions per cycle. A smaller number of predictions usually generates fewer mispredictions. For every misprediction, the processor has to invoke a recovery action to undo the effects of the mis-speculated instructions and commence the instruction fetching from the correct target. Since these recovery actions can cost several cycles, reducing the number of mispredictions will reduce the total execution time of programs. The advantages of guarding are summarized in the top part of Table 2.1.

2.1.3 Limitations of guarding

However guarded execution is not without limitations. First, guarded execution requires adequate support in the Instruction Set Architecture (ISA) level. This support that may range from a few new instructions, to a complete overhaul of the architecture and the instruction set. Providing adequate ISA support for guarding may be relatively easy when designing a new architecture, but extending an existing architecture is a much more challenging endeavor. Section 2.2 of this chapter will discuss in detail the instruction set requirements of guarding.

A second limitation of guarding is that it increases the total number of instructions executed dynamically. In general, instructions from both paths (traversed and not traversed) of a branch instruction are transformed into guarded instructions; the processor has to fetch and decode all these instructions, since it has no prior knowledge of which instructions will be useful until the instructions are fetched and examined, and the corresponding conditions (if any) are evaluated. After the condition evaluation, instructions from the not-traversed path are transformed into NOPs in the earlier stages of the pipeline (these NOPs need not be executed, but they will consume part of the fetch and decode bandwidth of the processor). These extra instructions, from the not-traversed path, may be scheduled to execute in parallel with other useful computation, if the processor has a sufficient number of resources. If sufficient resources do not exist, the additional instructions can actually increase the overall execution time. The execution time may also increase if the paths are of unequal lengths: when the longer path cannot be scheduled in parallel with other useful computation, the shorter path might have to be lengthened and performance along that path will suffer.

A third limitation of guarding is that the addition of guard operand(s) to each guarded instruction requires additional read port(s) in the register file. The additional read ports are used to read the

Advantages	Fewer control dependencies allow better compiler schedules Larger Basic Blocks present more parallelism to compiler Compiler can overlap execution of different control paths Improves or facilitates other compiler optimizations (S/W pipelining, etc.) Large sequential blocks of instructions Fewer control flow changes Less important to predict multiple branches per cycle Fewer mispredictions and fewer recovery actions
Limitations	Requires ISA support Increases the dynamic instruction count Requires additional read port(s) in the register file Consumes architecturally visible registers Requires adequate compiler support

Table 2.1: Advantages and limitations of guarding.

guard operand(s) of a guarded instructions in parallel with its regular source operands. The additional read ports will increase the size of the register file and can make it slower.

Finally, the conversion of control dependences to data dependences consumes architecturally visible registers. Without guarding, the register that holds the condition is used once to decide the branch outcome and set the correct PC value. With guarding, the condition register is used as a source operand in all the instructions it covers. Therefore, the lifetime of this register must extend to the last guarded instruction, thus increasing the register pressure. The problem is exacerbated by the instruction scheduler which, by rearranging instructions to increase parallelism, can increase the register lifetimes. One possible solution to this problem is to add a separate predicate register file [MLC$^+$92, RYYT89, KSR94], to relieve the pressure on the general purpose registers. This solution, however, is a major architectural change and may not be easily incorporated into existing architectures. Table 2.1 summarizes the advantages and limitations of guarding.

2.2 Instruction set support for guarding

The introduction of guarded instructions in a new or existing instruction set gives rise to a number of tradeoffs between opcode space, performance and flexibility. As described earlier, if-conversion modifies the program by introducing instructions that evaluate guard conditions, and instructions that use these conditions as guard operands. The extent of the instruction set support for these two operations will determine the performance of the generated guarded code. Two questions arise naturally at this point: (i) how can we evaluate the guard conditions, and (ii) how can we specify the guard conditions and the guarded computation?

2.2.1 Support for guard condition evaluation

When a conditional branch instruction is removed by if-conversion, it is replaced by an instruction sequence that evaluates the guard condition and sets the appropriate the guard registers for the subsequent

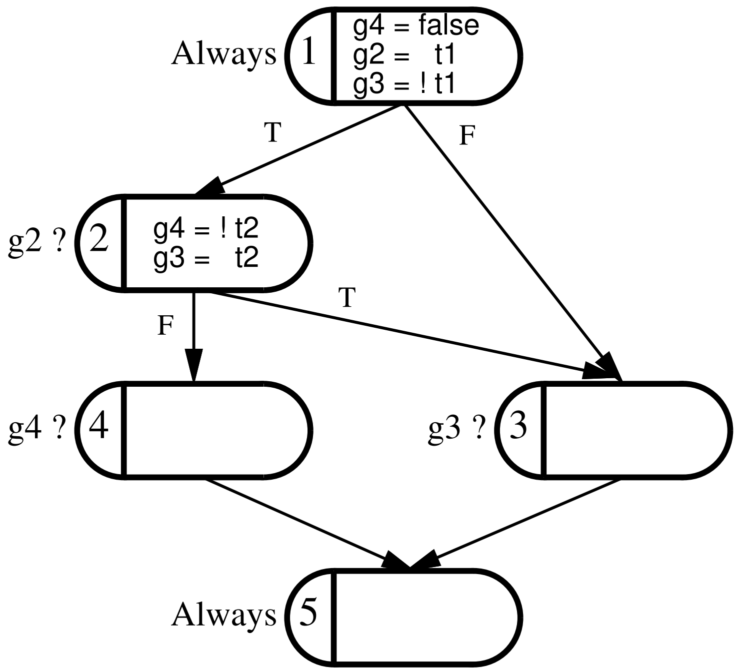


Figure 2.3: An example of multiple guard register sets and guard register initializations.

guarded computation. Depending on the complexity of the guard condition, its evaluation can range from a simple `set` instruction when the condition is a simple comparison of two values, to a sequence of logic instructions when the condition is a complex Boolean expression. Generally, an instruction set provides an adequate (if not complete) set of both bit-wise logic evaluation instructions such as logical `and`, `or`, etc, and comparison instructions such as `set_less_than`, `set_equal`, etc, that operate on the general purpose register file. These instructions are adequate to evaluate any arbitrary guard condition without requiring any further instruction set support. If a separate guard register file is used, the instruction set must include a set of condition evaluation instructions that will operate on the guard registers, as well as instructions to transfer values between the guard and the general purpose register files.

When the possible successors of a node A in the CFG are assigned distinct guard condition registers, each of these registers will have to be set in A according to A's condition. For example, in Figure 2.3, node 1 has two successors, nodes 2 and 3. These nodes are assigned different guard registers, which introduces two distinct set instructions in node 1. In addition, to ensure that no guard register can be used before it is set (which would yield an unknown value), some guard condition registers may need to be initialized to a known value before the condition evaluation begins. This situation arises for the inner-most nodes in nested conditional structures. For example, in Figure 2.3 node 4 is assigned guard register g4. The only set instruction for register g4 is in node 2, which is itself conditional on condition t1 (through the guard register g2). If t1 is false, the `set` instruction in node 2 will not be executed, and the value of g4 would be undefined. Therefore, g4 must be initialized to False in the beginning of node 1.

The initialization and set instructions are bursty in nature and in many cases in the critical path

		Value written to dst							
g	$Comparison$	U	$\overline{U}$	C	$\overline{C}$	OR	$\overline{OR}$	AND	$\overline{AND}$
0	0	0	0	–	–	–	–	–	–
0	1	0	0	–	–	–	–	–	–
1	0	0	1	0	1	–	1	0	–
1	1	1	0	1	0	1	–	–	0

Table 2.2: Predicate Definition Truth Table for *pred_set* instructions in the PlayDoh instruction set. A dash in an entry indicates that the destination register is left unmodified.

of the computation. The PlayDoh instruction set [KSR94] addresses these potential problems by supporting a rich set of predicate setting instructions. PlayDoh, supports a special `pred_clear` instruction which initializes all the guard registers in a single cycle. PlayDoh defines a separate guard register file consisting of sixty four 1-bit registers, so clearing all of them is a relatively simple task. PlayDoh also supports a set of predicate-set instructions which specify two sources for the condition evaluation and two distinct destinations. The syntax of a predicate set instruction is:

```
pred_set.<cmp> dst1, A1, dst2, A2, src1, src2 ? g
```

where `<cmp>` is the comparison to be performed on the source operands, `dst1` and `dst2` are guard registers, `src1` and `src2` are the source operands for the comparison, `g` is a guard register guarding the pred_set instruction and `A1` and `A2` are action specifiers that determine how the corresponding destination guard register will be set. Depending on the value of the guard register `g`, the result of the comparison and the action specifier, the `pred_set` instruction will either write a 1, write a 0 or leave the destination register unchanged. The possible actions are Unconditional, Conditional, OR and AND, as well as the complement of these four actions. Table 2.2 defines the outcome for all the combinations of predicate and predicate values, and action specifiers; a dash in an entry indicates that the destination register should not be written. For example, the instruction:

```
pred_set.<  t1, U,  t2, C,   a, b ? t3
```

will perform a "less-than" comparison of a and b, and will set the registers t1 unconditionally (i.e., regardless of the value of t3), and t2 conditionally, that is, only if t3 is true.

The various actions provided in the `pred_set` instruction are used to closely match the source level condition evaluation to the set of predicate set instructions avaliable. For example, if the source level condition is the OR of several variables, the compiler can use the OR action in the generated `pred_set` instructions that evaluate the condition. The `pred_set` instructions also facilitate the control-tree height reduction technique that will be described in the next Chapter, in Section 3.3.4.

Since a single `pred_set` instruction can set up to two guard registers, the number of instructions required to set the guard registers be reduced significantly. This is an important capability, considering the bursty nature of condition evaluation. Without two destinations, the `set` instructions can deplete the issue resources of a processor, excluding the actual computation from executing. Similarly, the `pred_clear` instruction reduces the number of guard register initializations that may be needed to ensure correct condition evaluation. While these architectural additions certainly improve the performance of guarded code, they are hard to incorporate in an instruction set. The `pred_set` instruction specifies 5 register operands, a comparison function and two set actions. With a total of six possible set actions and assuming that each register specifier is 5 bits (in PlayDoh it is 6 bits), we need 31 bits just to encode the parameters of the `pred_set` instruction, without specifying the actual comparison that should be performed, or the `pred_set` opcode. Clearly such an instruction cannot be supported in any of the current 32-bit instruction sets. Furthermore, allowing an instruction to have more than one destination can double the number of write-ports in the register file.

2.2.2 Support for specifying guarded computation

Once the guard condition is evaluated, guarded instructions use it to determine whether to commit their results. Compared to ordinary instructions, guarded instructions must specify their guard condition registers and a comparison function that will determine, depending of the value of the guard registers, whether the overall guard condition is true. The specification of the each guard register operand as well as the specification of the condition consume instruction bits, a precious resource that should be used cautiously and efficiently. Given a fixed instruction budget, a designer has to balance the complexity of the permitted guard conditions, and the number of guarded instructions supported in the instruction set.

Guard Condition Complexity

Allowing more powerful guard expressions in a single guarded instruction can shorten the critical paths through the code by speeding up the evaluation of the guard condition. For example, if the instruction set allows the conjunction of two registers to be used as a guard expression (in a manner similar to Hsu and Davidson's work [Hsu86, HD86]), the guard condition $A\&B$ can be attached to a guarded instruction directly. For example an add instruction guarded by that condition would be written as:

```
g_add (A & B) ? Rd, Rs1, Rs2
```

If the instruction set restricts the guard condition to a single register, a complex guard condition must first be computed in a temporary register before it can be used. The above guarded add for example would have to be expanded into:

```
and            R1, A,   B
g_add  R1 ? Rd, Rs1, Rs2
```

A more complicated guard condition clearly requires more bits for its specification and therefore consumes more opcode space. In the remainder of this thesis we assume that only one guard register can be specified per guarded instruction, as we feel that the specification of more than one guard operand would be impractical for an existing instruction set. Even in the context of a new instruction set, committing two operand specifiers to the guard condition will either reduce the number of operands of the actual computations, or will add read ports to the register file.

Guarding general computation

The extent of support for guarded instructions is another manifestation of the space–flexibility trade-off, and is especially important when extending an existing instruction set. To allow maximum flexibility to the compiler, every instruction must be available in a guarded form, allowing any arbitrary conditional code construct to be transformed in a guarded form by attaching the guard condition to it. This symmetry simplifies the compiler construction since no case analysis is needed to determine whether a code sequence can be expressed using the existing set of guarded instructions. Proposed methods for specifying guarded execution use an additional operand field for each instruction [MLC$^+$92, RYYT89, Mac93, KSR94]. An additional bit in the opcode specifies whether the guard condition should be true or false for the instruction to execute. This specification method allows maximum flexibility to the compiler but consumes a significant fraction of the total instruction bits: for an architecture with 32 registers, 5 unused instruction bits must be dedicated to the guard condition operand.

Using conditional moves to support guarding

Existing instruction sets have already partitioned the opcode space in a certain way and may not have enough unused opcode bits to allocate to specify guarded instructions. The opcode space limitations forced designers to introduce only a small number of guarded instructions. Many instruction set architectures such as the Cray-1 [Rus78], the DEC Alpha [Com88], the SPARC V9 [Cas93], the MIPS R10000 [Pri94], the PowerPC [Cor94] and the Intel P6 [Gwe95] include a *conditional move* (`cmov`) instruction. The `cmov` instruction is almost trivial to add: since a move uses only one source specifier, the second specifier is used to specify the guard condition. Having only two source operands, the conditional move instruction does not require the additional register file read port that guarding usually requires.

Conditional move instructions have an additional desirable property: they can be used to synthesize other guarded instructions. This is achieved using ordinary unconditional instructions to compute results in temporary registers and then committing the results in the real destination registers using conditional moves. For example, a guarded add instruction will be synthesized using a ordinary add followed by a conditional move instruction. In general, the guarded form of instructions that may cause exceptions (such as loads, stores, integer division or floating point instructions) are harder to synthesize using `cmovs`. For example, synthesizing a guarded load using a load into a temporary register followed by a conditional move may incorrectly generate exceptions. The synthesis of guarded versions of these instructions can only be allowed if additional code is inserted to check the instruction operands and verify that the operation will not cause any exceptions.

The Cray compilers have always been able to deal with this problem. Unlike the STAR-100 and the ASC vector processors that define a general purpose vector mask register, in the Cray-1 architecture the vector mask can only be used in a vector move (called "merge") instruction. To synthesize the "masked" version of other vector instructions, the compiler used instruction sequences that would initialize the masked element of the operands, so that the subsequent vector instruction would not cause an exception. For example, for a vector division, the compiler would first set all the masked elements of the divisor to 1, ensuring that the subsequent vector division would only cause an exception only for the non-masked elements. A representative subset of the scalar equivalent of these sequences is listed in the second column of Table 2.3. Mahlke *et al.* [MHM$^+$95] used these sequences to synthesize the guarded form of all instructions using `cmovs`. The code sequences in Table 2.3 are between two and four (dependent) instructions in length, and can be used effectively only when the guarded condition and the source operands are available well in advance of the uses of the guarded result. When the compiler synthesizes the guarded version for an expression tree, it can optimize the sequences by removing the conditional moves for all the intermediate (temporary) registers. The sequences can also be optimized using *predicate promotion*, a code transformation that will be described in the next chapter, in Section 3.3.7.

The synthesized sequences can be shortened by the use of *non-excepting* instructions. The idea is that, if non-excepting instructions are available in an instruction set, the guarded form of any instruction can be synthesized simply using the non-excepting version of the instruction storing its result in a temporary register, followed by a conditional move to commit the result in the destination register, without the need for instructions that verify that an exception will not be generated. For example, a guarded load can be synthesized by a non-excepting load followed by a conditional move. As can be seen in the last column of Table 2.3, the sequences that use non-excepting instructions are all two instructions in length, a significant improvement over the sequences using ordinary instructions.

While the code sequences that use non-excepting instructions are guaranteed not to cause any spurious exception, they will not generate any exceptions even when the original code would have generated one. To ensure that an exception should only be generated when the guard condition is true, the processor must buffer the exception and propagate it through the computation until the guard condition is verified to be true. To implement the exception buffering, the processor can maintain an exception tag along each register. A non-excepting instruction that suppresses an exception would set the exception tag and other computation instructions would propagate it to their destination registers. The tag can be checked later by the conditional move instruction if the condition evaluates to true and if so, the exception should be serviced. This functionality is very similar to sentinel scheduling proposed by Mahlke *et al.* [MCH$^+$92] and the Rogers and Li's hardware support for speculative loads [RL92]. However, the storage required to postpone the handling of exceptions (for example, the exception tags along each register) is part of the architectural state of the processor. Therefore, this storage must be saved and restored in contexts switched, external interrupts etc, requiring additional mechanisms to support these operations. Overall, non-excepting instructions can be used to facilitate the synthesis of guarded forms of instructions only in environments that ignore exceptions, or on hardware platforms that provide enough hooks to allow the use of non-excepting instructions without compromising exception handling.

Restricted guarding

Integrating guarded execution in an existing instruction set is a non-trivial task: the number of unused opcode bits is usually small and the opcode space is already partitioned in a certain way. One way to add partial support for guarded execution in an existing instruction set without modifying the existing instruction formats, is to provide the guarded form only for the instructions whose encoding is sparse enough to accommodate the guard operand(s). For example, imagine trying to add a 5-bit guard operand to the MIPS R2000 instruction set. Figure 2.4 shows the three major instruction formats. In Figure 2.4 we see that the *Immediate* and *Jump* formats use all the available bits, leaving no space for the addition of a guard condition. (A guarded jump instruction would, of course, be equivalent to a conditional branch instruction, with the sole difference of a larger target range.) On the other hand, all the instructions in the Register format contains 5 unused bits, (in the MIPS instruction set, all instructions that use the `sa` field leave the `rs` field unused), exactly as many as needed to encode the guard

Guarded instruction	Instruction sequences using CMOVs	Instruction sequences using non-excepting instructions
`g_add   cond? dst, src1, src2`	`add          tmp, src1, src2` `cmov  cond? dst, tmp`	same

`g_div   cond? dst, src1, src2`	`mov          tmp2, src2`
	`cmov  cond? tmp2, 1`
	`div          tmp, src1, tmp2`
	`cmov  cond? dst, tmp`
		`div        tmp, src1, src2`
		`cmov cond? dst, tmp`

`g_load  cond? dst, addr(base)`	`add          tmpaddr, base, addr`
	`cmov !cond? tmpaddr, $safe_addr`
	`load         tmp, tmpaddr(0)`
	`cmov  cond? dst, tmp`
		`load       tmp, addr(base)`
		`cmov cond? dst, tmp`

`g_store cond? dst, addr(base)`	`add          tmpaddr, addr, base`
	`cmov !cond? tmpaddr, $safe_addr`
	`store        tmp, 0(tmpaddr)`
		same

Table 2.3: Synthesis of general guarded statements using conditional move instructions, with and without non-excepting instructions. The $safe_addr register contains an memory address guaranteed not to cause an exception.The guarded form of floating point instructions are synthesized in a manner similar to the `div` instruction.

Format						
	6 bits	5 bits	5 bits	16 bits		
I−Type (Immediate)	opcode	rs	rt	immediate		
	6 bits	24 bits				
J−Type (Jump)	opcode	target				
	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits
R−Type (Register)	opcode	rs	rt	rd	sa	function

Figure 2.4: The MIPS R2000 instruction formats.

register specifier. Based on this observation, Pnevmatikatos and Sohi [PS94] defined *restricted guarding* to include the guarded form only for ALU operations. However, their results indicate that the lack of guarded forms of memory operations and operations with immediate values, limits the performance potential of restricted guarding.

2.3 Hardware support for guarding

The hardware required to support guarded execution are fairly straightforward. For each guarded instruction the execution engine has to perform two actions: (i) read the guard register and evaluate the guard condition and (ii) squash the instruction if the guard condition is false. The first action can be implemented by adding an extra read port in the register file so that each guarded instruction reads its guard operand in parallel with its source operands. The implementation of the second action is more involved.

In a pipelined processor without guarded execution, by the end of the decode stage the control logic has determined whether the instruction will generate a result to be written in the destination register. The forwarding logic paths are set so that once a destination is declared (that is, when the instruction with that destination register enters the pipeline), all subsequent reads of this register are forwarded from this instruction, until the register file is updated with the new value. A guarded instruction however may or may not produce a result, depending on its guard condition. This conditional behavior mandates careful modifications in the forwarding structures of the processor.

2.3.1 Implementing guarding using select logic

A simple solution is to force the guarded instruction to *always* produce a result, either a new value, or the old value of the destination register. This solution is simple and requires no changes in the forwarding logic, as the invariant that every instruction generates a (correct) result is maintained. Essentially, this solution converts the Write-After-Write hazard that exists between the previous write into the destination register and the current (potential) write by the guarded instruction into a Write-After-Read hazard, since the older value is read instead of just being overwritten.

To be able to select between the new and the old value of the destination register, each guarded instruction must read the value of its destination register along with its source and guard register during the decode stage of the pipeline. Treating the destination register as a source operand adds one more read port in the register file, and brings the total number of read ports to four per instruction issued; this additional read port makes the register file bigger and possibly slower. Furthermore, the old value of the destination register may be a result of an instruction that is still in flight in the pipeline. To handle this case correctly, one more forwarding path must be added in the pipeline to forward the old value of the destination register.

In addition, the implementation of this solution requires a multiplexor at the end of each stage where the result of an instruction is produced, to select between old and new value. For example, for a 5-stage pipeline a multiplexor is needed at the end of the EX stage for ALU instructions and at the end of the MEM stage for load instructions, as shown in Figure 2.5. These multiplexors are usually incorporated in the computation logic or in the pipeline latches. However, if the multiplexors cannot be merged with other pre-existing circuits, they may increase the critical path of the instruction execution. The total amount of hardware devoted to the support of guarding is not trivial as can be seen in the high level pipeline schematic in Figure 2.5.

2.3.2 Implementing guarding by overloading the forwarding logic

If reading the destination register of guarded instructions is impractical, guarding can be implemented by overloading the guarding functionality on the forwarding logic and paths. Consider the following instruction sequence:

```
g_add   r9 ? r3, r1, r2
sub          r5, r4, r3
```

During decoding, the `sub` instruction will read its operands from the register file, reading the old value for register `r3`, as the new value is not yet produced by the `g_add` instruction. If the guard condition of `g_add` is true, the new value of `r3` can be forwarded to the `sub` instruction using the existing forwarding paths. If the guard condition is false, the `sub` instruction is supposed to use the old value of `r3`, which is already read. If the forwarding logic *does not* forward the result from the guarded `add`, the `sub` will execute using read the correct (old) value of register `r3`. Thus, if the forwarding logic is modified so that it only forwards the result of a guarded instruction when the guard condition is true, the extra register file read port is not required. The modification of the forwarding logic obviates the need for a multiplexor to select the correct value: we use the multiplexors in the forwarding paths to achieve that function. Figures 2.6 and 2.7 detail the pipelined execution of the above instruction sequence for the guard condition evaluating to true and false respectively.

Figure 2.8 outlines the additions to a pipeline that are required to support guarding. Comparing Figures 2.5 and 2.8 we see that implementing guarding by overloading the existing forwarding logic and paths is significantly less demanding in hardware. Nevertheless, it may introduce new critical paths in the pipeline control logic.

2.3.3 Implications of out-of-order execution

When a processor supports out-of-order execution, the decoding and the execution of an instruction are decoupled, and it is possible that the guard register of a guarded instruction is not available during decoding. To continue decoding instructions, the processor must allow guarded instructions to enter the pipeline, even if it is not known at decode time whether they will commit their results or not. However, the asynchronous nature of the out-of-order execution disallows the use of the forwarding logic

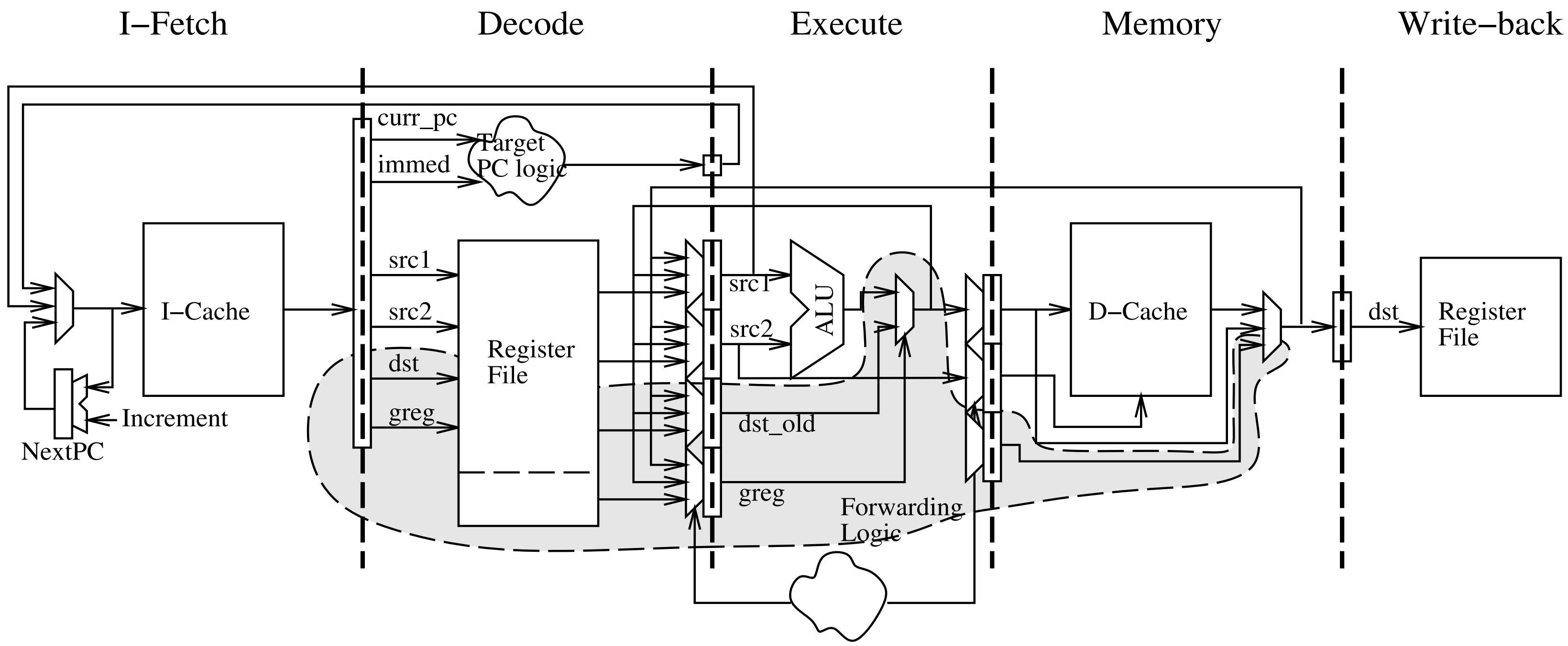


Figure 2.5: A 5-stage pipeline implementing guarding as a select between the (potential) new and the old value of the destination register. The shaded portion of the pipeline is the portion that needs to be added to support guarded execution while the dashed line in the register file indicates the possibility of a dedicated guard register file.

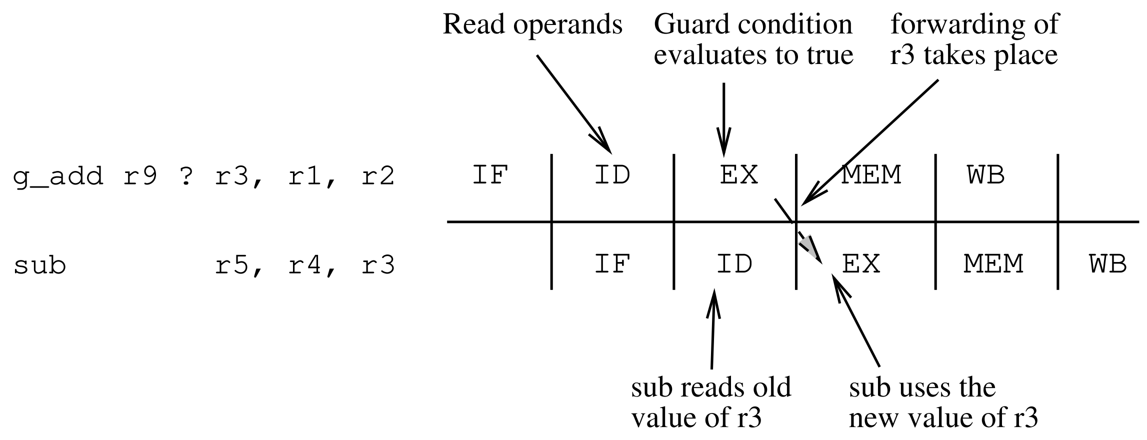


Figure 2.6: Pipeline operation when the guard condition of a guarded instruction evaluates to true. When the guard condition of a guarded instruction evaluates to true, its result is forwarded to subsequent instructions using the existing forwarding paths.

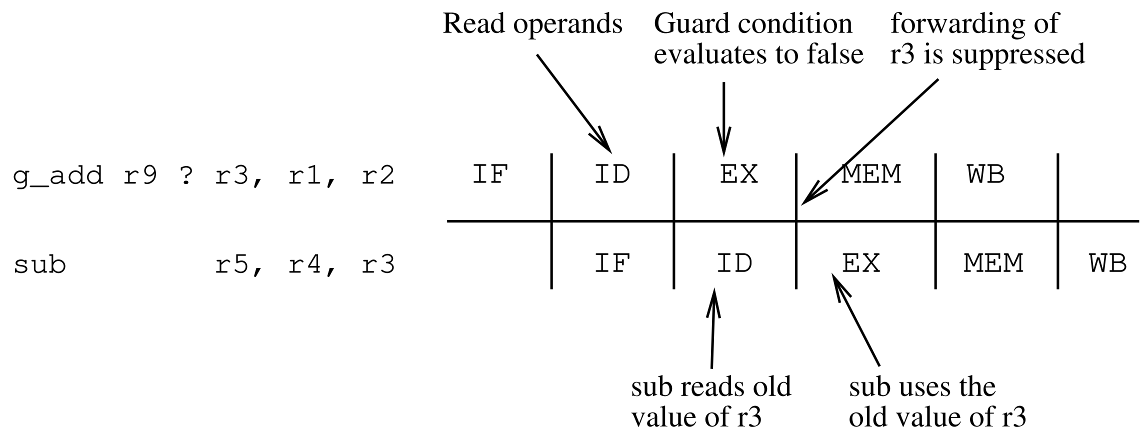


Figure 2.7: When the guard condition of a guarded instruction evaluates to false, the forwarding of its result in suppressed, and subsequent instructions use the old value of the destination register of the guard instruction.

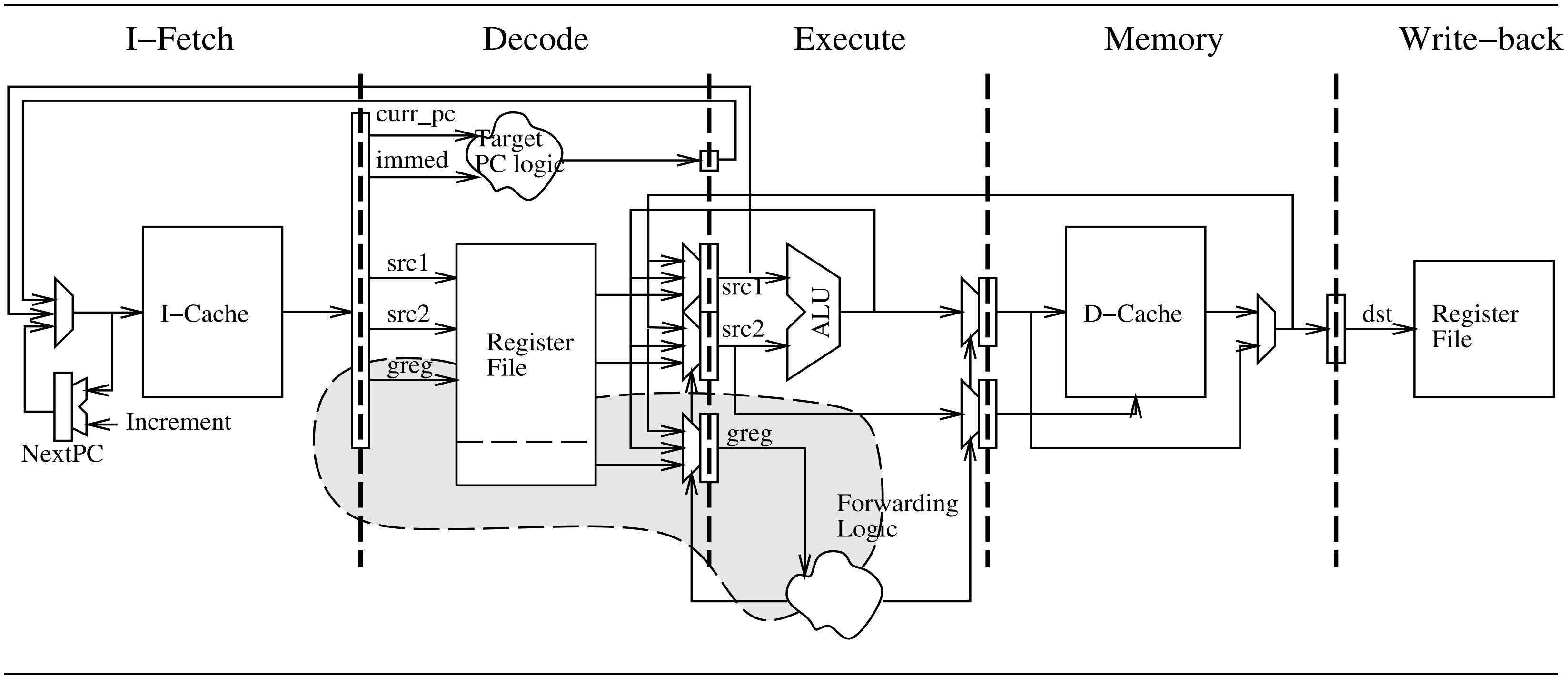


Figure 2.8: A 5-stage pipeline implementing guarding with changes in the register forwarding logic. The shaded portion of the pipeline is the portion that needs to be added or modified to support guarded execution, while the dashed line in the register file indicates the possibility of a dedicated guard register file.

to implement the on-the-fly squashing described above. Instead the processor must treat all guarded instructions as selects between the old and the new value. This solution is essentially the out-of-order execution equivalent of Figure 2.5. To implement this solution, the Reservation Station entries [Tom67] or Reorder Update Unit entries [SV87] must be augmented with storage that will hold the old value of the destination register, in case the guard condition evaluates to false. The logic that determines whether an instruction is ready to be executed and produce a result must also be modified. A guarded instruction is ready to produce a result only after the guard condition is available. If the guard condition evaluates to true, the instruction is ready when all the source operands are available; if the guard condition evaluates to false, the instruction is ready when the old value is available.

2.4 Summary

Guarded execution is a very powerful and promising concept, with the potential to reduce the unpredictability of the control flow caused by branches, smoothing the flow of instruction in the pipeline(s) and exposing more instruction level parallelism to the compiler and processor. The amount of hardware required to support guarded execution is also reasonable, making guarding an attractive feature for inclusion in an instruction set.

However, the integration of guarding in an instruction set is not easy, especially when the designer has to extend an existing instruction set, rather than design a new one. In addition, compiler techniques that make efficient use of guarding and introduce small amounts of overhead computation are needed. The next chapter describes in detail the compilation techniques that can be used to take advantage of guarded execution, while Chapter 4 describes GUARD instructions, a class of instructions that permit the introduction of full guarding in new as well as in existing instruction sets.

Chapter 3
Compiling for guarded execution

A compiler that effectively utilizes guarded execution involves a considerable number of decisions and tradeoffs. Generally, the front end of the compiler remains unchanged, implementing all the traditional code transformations and optimizations. Guarding introduces a set of new phases and possible optimizations in the back end of the compiler. Figure 3.1 shows the typical phases of a compiler that supports guarded execution. While this arrangement of compilation phases is not the only possible one, it is a natural one, since if-conversion involves decisions that are best taken when the program representation is as close to the target executable as possible. In addition, placing all the guarding specific phases before the instruction scheduling allows the same mechanism to be used for ordinary as well as guarded computation.

In Figure 3.1, the guard-specific compilation phases perform the following major functions:

- **Region selection:** identify the regions to be if-converted using guarding, possibly preconditioning the code generated by the compiler to increase the effectiveness of if-conversion,
- **If-conversion:** determine the conditions that guard each basic block in the selected region and insert the instructions required to evaluate the guard conditions,
- **Guard-specific optimizations:** perform optimization steps that improve the quality of the guarded code, and
- **Instruction scheduling:** schedule the if-converted regions for a particular execution model and resource configuration.

Each of these steps involves a number of tradeoffs that determine the quality and the performance of the generated code. The compiler aspects of guarded execution of general purpose programs have been only recently explored in detail and mostly by the Illinois IMPACT group [Lin92, MLC$^+$92]. This chapter presents an overview of the code transformations that can be used by a compiler that supports guarded execution.

3.1 Choosing what to guard

Starting with a control flow graph of a (piece of a) program, the compiler has to decide: (1) where if-conversion is applicable, and (2) where would it be beneficial. The first decision depends mainly on the CFG structure, since if-conversion cannot eliminate function calls and returns and in general cannot eliminate indirect branches; also, as described in Chapter 2, if-conversion has limited applicability on diverging CFG structures such as decision trees. The second decision involves an estimation of the execution time, parameterized by the execution model as well as the number and type of resources available in the target processor.

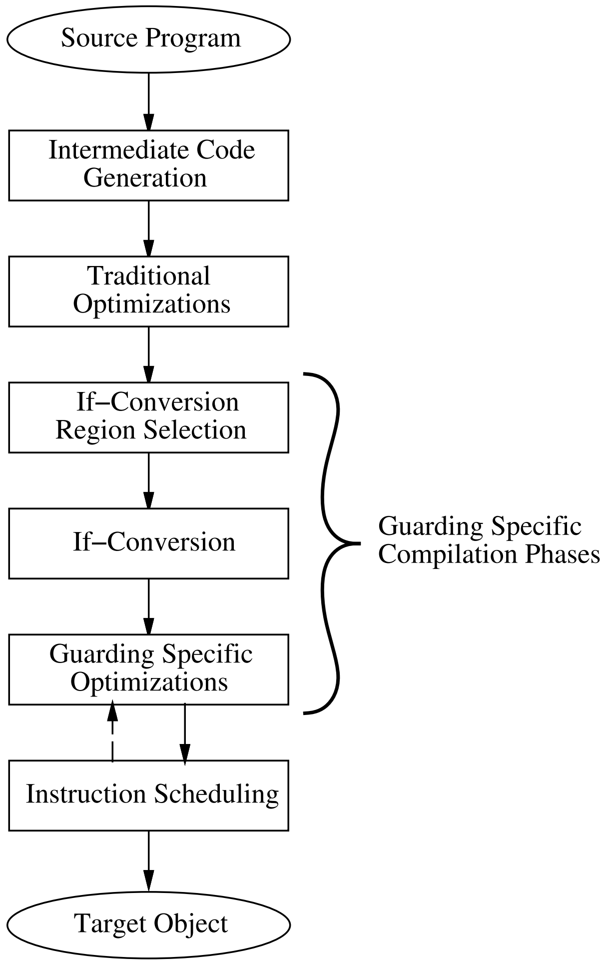


Figure 3.1: The structure of an if-converting compiler. The dashed line indicate a possible feedback from the instruction scheduler that may guide the if-conversion specific optimizations.

To decide which parts of the program should be guarded, the compiler makes an attempt to balance the execution time benefits of if-conversion, against the overhead of the computation that will be squashed at run time. This process can be helped by the use of profiling information. An if-converting compiler can take advantage of several types of profile information. The simplest and more widespread type of profiling information is basic block counts; tools such as *pixie* [MIP90] and *QPT* [Lar93, BL94] can provide this type of information. Other types of profiling include CFG arc frequencies [Lar93, BL94], and branch prediction accuracies [FF92].

Basic block or arc frequency profiling can be used to control the amount of squashed computation; using this information, the compiler can identify nodes and arcs that are so infrequent that including them in the if-converted code will only consume resources, without a corresponding improvement in execution time. The profile information can also be used to completely bypass the if-conversion process in the part of the programs that are executed infrequently, in order to reduce the compilation time for the program.

Branch prediction profile information can be used by the compiler when the target processor utilizes a dynamic branch prediction mechanism. In this context, branches that are correctly predicted most of the time may not be a very big performance impediment, and may not be worth eliminating. With branch prediction profile information, the compiler can determine the branches that dynamically cause most of the incorrect predictions and identify them as prime candidates for if-conversion. If profile information for the prediction accuracies of branches is not available, the compiler can resort to heuristic approaches to derive static branch prediction estimates [BL93].

When no profiling information is available, the compiler can follow a conservative approach and assume that all paths in the CFG are equally likely. This approach will only if-convert regions for which the guarded schedule will perform better than the original schedule regardless of the exact arc frequencies of the branch prediction accuracies. Obviously, this conservative approach will not be able to achieve the performance levels that profiled based approaches will.

Next, we describe two region selection schemes that have been used in earlier studies. The first, called Multiblock [PFS93], is based on the CFG structure, and the second, called Hyperblock [MLC$^+$92], is based on the CFG structure as well as on profiling information and heuristics to guide the Hyperblock formation process. Both schemes form a collection of nodes of the CFG, that have a single entry point. This requirement allows the compiler to freely rearrange the instructions inside the region. If multiple entry points are allowed, the compiler cannot move instructions across the multiple entry points.

3.1.1 The Multiblock region selection scheme

Pnevmatikatos *et al.* [PFS93] used the CFG structure to construct *Multiblocks*, which are loop free regions of contiguous basic blocks having at most two possible successors. The simple structure of these regions permits an easy, on-the-fly identification of multiblocks and allows dynamic branch prediction to be performed in a granularity of multiblocks rather than basic blocks. Multiblocks must satisfy three conditions:

- **Condition 1:** there should be a single entry point to each multiblock,
- **Condition 2:** the should be no nested loops inside a multiblock, and
- **Condition 3:** a multiblock should have at most two targets.

The Multiscalar processor [FS92, Fra93], used these multiblocks to dispatch multiple basic blocks to each of its multiple execution stages and to “avoid” predicting the branches that were inside multiblocks. Within each execution stage, the multiscalar processor employed speculative execution and suppressed the commitment of speculative results until all the control dependences were resolved, implementing in this way a dynamic form of guarding.

The guarding regions used by Pnevmatikatos and Sohi [PS94] are also similar to multiblocks, with the additional restriction that one of the two possible targets is restricted to be the fall through path. Since both these targets can be encoded in a single branch instruction, all the branches inside these regions can be eliminated using if-conversion, and the composite branch condition can be used in the branch that terminates the guarding region. Pnevmatikatos and Sohi used these regions to evaluate the effects of guarded execution on the static and dynamic branch behavior of programs.

3.1.2 The Hyperblock region selection scheme

The *Hyperblock* construct [MLC+92], proposed by the Illinois IMPACT group, is a hybrid of trace scheduling and if-conversion. Given a control flow graph, a hyperblock is formed by identifying a subset of the graph for inclusion in the hyperblock. The criterion for inclusion of a basic block into a hyperblock is a function of the execution frequency, the size and the instruction characteristics of each basic block. When the execution path can be predicted with high probability, the hyperblock selection process degenerates to trace selection process. To handle unpredictable branches hyperblock resorts to if-conversion.

The set of nodes selected for inclusion in a hyperblock must satisfy two conditions:

- **Condition 1:** There should be a single entry point from blocks outside the set, and
- **Condition 2:** The should be no nested loops inside the hyperblock.

These conditions ensure that all hyperblocks are entered only from the top and that each instruction in a hyperblock is executed at most once. The main difference from a multiblock is that a hyperblock is allowed to contain branches (and therefore have multiple exit targets). This difference gives more flexibility to the compiler to select the hyperblocks and permits the hyperblocks to be larger than multiblocks. Lin's Masters thesis [Lin92] and Mahlke *et al.* [MLC+92] provide a more detailed discussion of the hyperblock formation.

3.2 If-conversion basics

If-conversion as introduced by Allen *et al.* [AKPW83] operated on a per-statement level on the source of Fortran programs. Their if-conversion algorithm works by allocating a Boolean variable to hold the guard condition for each conditional branch in the code. As each branch under consideration is eliminated, the corresponding Boolean variable is added into the guard expression for all the statements that depended on that branches outcome. The process iterates until all branches in the loop body are processed, at which point, each statement is guarded with an arbitrary logical expression involving one or more Boolean variables. This expression is a flat representation of the conditions under which each statement would be executed. Since the algorithm was based on textual modifications of the source program, the guard expressions may be redundant. To reduce the amount computation required for the condition evaluation, Allen *et al.* suggest a Boolean minimization step which, using the Quine-McCluskey prime-implicant simplification method, would minimize these large expressions and eliminate any redundancies in them.

The textual-based if-conversion algorithm proposed by Allen *et al.* is very well suited to the purposes of a source to source translation program such as the PFC, but does not fit well with the graph-based internal representation used by compilers. Furthermore, the assignment of guard expressions and the insertion of the required instructions to evaluate them can be tricky, especially when the code is not structured, that is, when the code cannot be expressed as a hierarchy of nested blocks, where each block has a single entry and a single exit point. At least one compiler, the Cydrome compiler used by the Cydra 5 [RYYT89], is reported [PS91] to produced incorrect code for certain control flow graphs.

Park and Schlansker addressed this problem and formalized the if-conversion process in the RK algorithm [PS91]. The RK algorithm determines the guard condition that will be assigned to each basic block and the operations required to evaluate the guard conditions. The algorithm is provably correct for all types of control flow graphs, and generates a minimal set of condition evaluation instructions. Lin [Lin92] compared three if-conversion algorithms, one that associates a guard condition to each CFG arc (roughly corresponding to Allen *et al.*'s if-conversion algorithm after the Boolean minimization step and common subexpression elimination), one that associates one guard condition for every basic block, and the RK algorithm. Lin's results show that the optimality of the RK algorithm translates to a measurable improvement in execution time for four benchmark programs.

Given a directed acyclic control flow graph G with a single entry point, the first step of the RK algorithm is the computation the control dependences in G, as described by Ferrante *et al.* [FOW87]. A node Y in G is *control dependent* on node X when (i) X has two exit arcs, and (ii) when the path from one of the exit arcs from X guarantees that Y will be executed, while a path from the other arc may result in Y not being executed. The RK algorithm uses the control dependence information to determine which conditions determine the execution of each basic block. Next the algorithm determines the necessary set instructions that have to be inserted at every decision point (i.e., in the place of the original conditional branch instructions). A final step determines whether it is possible for a guard condition to be used without being previously set. The RK algorithm handles these cases by introducing initialization instructions in the entry point of the control flow graph. The result of the algorithm is two mappings, one called R and K (hence the name). R is a mapping from basic blocks to guard conditions and determines the assignment of guard conditions to each basic block in the control flow graph. K is a mapping between guard registers and sets of basic blocks and determines where (in which basic blocks) should a particular guard register be set. The number of guard registers that is produced by the RK algorithm depends solely on the structure of the input CFG. If the number of guard registers required to guard a CFG is larger than the number of available registers, the compiler has two options: either to partition the CFG into a set of smaller CFGs and re-run the RK algorithm, or to rely on register allocation to make the best use of the available registers.

Figures 3.2 shows the RK algorithm. The first step is the computation of control dependences in G, identifying in this way the *control equivalent* basic blocks, i.e., basic blocks that are executed under the same condition and should therefore be assigned the same guard register. The second step computes the R and K mappings. The third step, Augment(K), solves a data flow problem on G to determine the set of guard registers for which a use can be reached from the entry point of the graph before the register is defined. Augment(K) then records this set in the K mapping for the entry node of graph G.

The *if-convert* algorithm, shown in Figure 3.3, is straightforward. The first step is to call RK to determine the R and K mappings. Then steps (3) and (4) use the K mapping to determine which condition evaluation instructions must be added in each basic block. Step (5) removes the conditional branch from an if-converted basic block and steps (6) and (7) use the R mapping to introduce the appropriate guard register in each instruction of a guarded basic block of code. The result of this algorithm is a single, straight-line sequence of instructions.

Figure 3.4 illustrates the operation of the RK algorithm with a simple control flow graph of two nested if-then-else statements. The corresponding R and K mappings are shown in Tables 3.1 and 3.2. The two conditional branches in nodes 1 and 2 define the two corresponding conditions $t1$ and $t2$. These conditions are used to set the individual guard registers. The R mapping assigns one condition register to each node of the graph, with the exception of nodes 1 and 7 which are always executed, and node 6 which is control equivalent with node 2 and is assigned $p2$ as the guard register. The K mapping determines that the guard registers 4 and 5 can be used before they are set (this would happen when condition $t1$ is false, so node 2 will not be executed and the guard registers $p4$ and $p5$ will not be

```
Algorithm RK:
    Given a rooted graph G, compute the R and K mappings
    {
(1)    Compute Control Dependences(G)
(2)    Decompose Control Dependences(G) into R and K
(3)    Augment(K)
    }
```

Figure 3.2: The RK algorithm.

```
Algorithm if-convert(G):
    Given a rooted, acyclic graph G,
       produce the if-converted graph G'
    {
(1)    use RK to compute the R and K mappings
(2)    foreach basic block B {
(3)       foreach guard register P in K such that
                                the set K[ P ] contains B {
(4)          add an instruction in B that sets P
          }
(5)       remove the branch if any
(6)       foreach instruction I in basic block B {
(7)          assign guard register R[ B ] to instruction I
          }
       }
    }
```

Figure 3.3: The if-conversion algorithm.

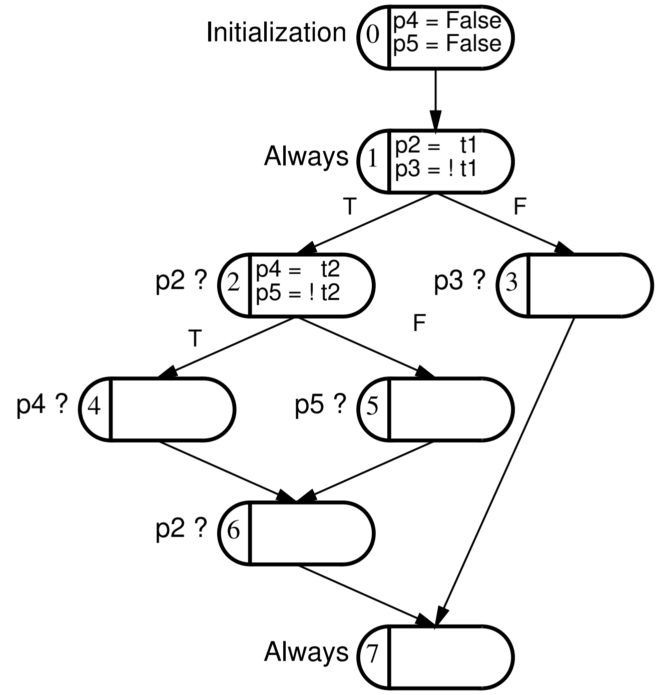
set), and must therefore be initialized in node 0.

3.3 Guarding-specific optimizations

The success of if-conversion depends on the structure of the control flow graph as well as on the dynamic behavior of the program. Several optimizations can improve the effectiveness of if-conversion. These optimizations can be categorized into two sets: optimizations that manipulate the control flow graph before if-conversion is performed so as to increase its ability to eliminate branches, and optimizations that reduce the execution time of the if-converted code in order to improve the overall execution time of the program. The optimizations can be further categorized according to whether their performance is dependent on the availability of accurate profiling information. A general purpose compiler cannot assume the availability of (accurate) profiling information, and is therefore limited to a set of optimizations that will improve the program's execution time regardless of profiling information. Ta-

```
< 1 >
if (t1) {
        < 2 >
        if (t2) {
                < 4 >
        } else {
                < 5 >
        }
        < 6 >
} else {
        < 3 >
}
< 7 >
```

(a)



(b)

Figure 3.4: Condition register assignment and definitions by the RK algorithm. Part (a) shows the skeleton of a C code fragment, and part (b) shows the corresponding CFG, annotated with the condition registers and the instructions to set the condition registers. In part (b), pn is the condition register and tn is the branch condition for node n.

Basic Block	0	1	2	3	4	5	6	7
Guard Register	N/A	True	2	3	4	5	2	True

Table 3.1: The R mapping assigns a guard register to each basic block. Note that the same guard register is assigned to the control equivalent basic blocks 2 and 6.

Guard Register	2	3	4	5
Basic block	{1}	{-1}	{2, -0}	{-2, -0}

Table 3.2: The K mapping determines the basic blocks in which each guard register should be set. A minus sign indicates that the required set instruction should invert the outcome of the comparison.

ble 3.3 categorizes a number of proposed optimizations according to these two criteria. In the following subsections we give a short overview of these optimizations and we discuss their requirements.

Optimization	Before or after If-conversion?	Profile information needed for performance?
Loop restructuring	Before	Useful
Tail duplication	Before	Yes
Loop peeling	Before	Yes
Control-tree height reduction	After	No
Condition evaluation optimizations	After	No
Exit coalescing	After	Yes
Predicate promotion	After	Useful

Table 3.3: Categorization of if-conversion specific optimizations.

3.3.1 Loop restructuring

Loop restructuring (also called loop branch coalescing by Chang *et al.* [CHPC95]) transforms the control flow graph of a loop into a canonical form, in an effort to increase the number of forward branches and make the if-conversion more effective.

The basic observation motivating this optimization is that the front end of the compiler can produce multiple loop-back arcs for a single logical loop of the source code. These loop-back arcs correspond to uses of the C `continue` and `goto` statements, or the results of the jump-optimization phase of the compiler. Since if-conversion cannot eliminate backward branches, loop restructuring merges all the loop-back arcs into a single one. All the original backward branches of the loop are redirected to this single backward branch using forward branches. These forward branches will later be eliminated by the if-conversion. For example, the control flow graph in Figure 3.5(a) has two loop-back arcs, one from node $n1$ and another from node $n2$. Loop restructuring, shown in Figure 3.5(b), will move the loop-back arc from node $n2$ into a new node $n3$ and redirect the loop-back arc from node $n1$ to this new node, transforming it into a forward branch.

While the loop restructuring transformation is very successful in increasing the effectiveness of if-conversion in eliminating branches [MHB$^+$94], it comes at a considerable cost. The amount of computation that is bypassed in the original code using the multiple backward arcs can be large, and including it in the if-conversion can hurt the performance instead of improving it. Hence, loop restructuring can be used successfully only when accurate profiling information is available to the compiler to evaluate the merit of this optimization, or when the amount of extra computation is limited. When accurate profiling information is not available, loop restructuring can confuse nested loop structures for a single loop with many loop-back arcs; for example, the control flow graph in Figure 3.5(a) could be the control flow graph of a doubly nested loop, or the control flow graph of a single loop that contains a `continue` statement.

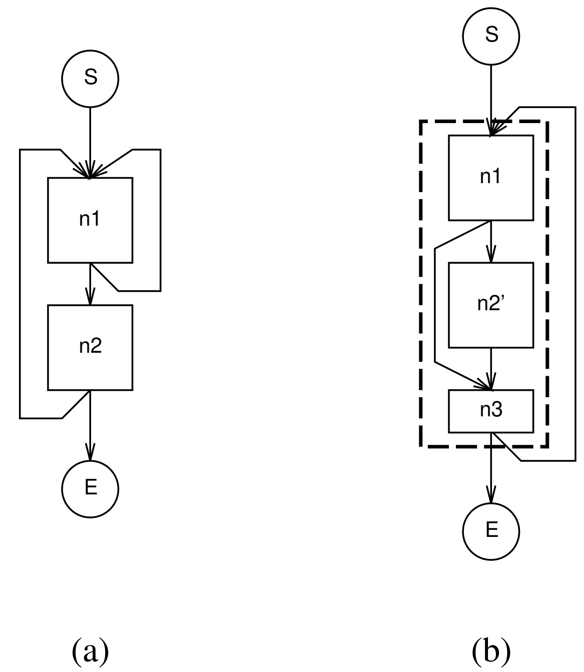


Figure 3.5: Loop restructuring example. Part (a) shows the original control flow graph. Part (b) shows the CFG after loop restructuring. Loop restructuring splits node $n2$ into $n2'$ and $n3$, and redirected the loop-back arc from node $n1$ to node $n3$, converting it into a forward arc.

3.3.2 Tail duplication

Tail duplication is a technique that eliminates multiple entries to a control flow region by replicating some of the nodes. This is a general optimization but is of particular importance in the hyperblock formation, because the trace selection process can exclude certain nodes and any reconverging arcs from these nodes will introduce additional entry points in the hyperblock. This situation is illustrated in Figure 3.6. The dashed line in Figure 3.6(a) outlines the desired hyperblock. Node $n3$ is excluded from the hyperblock formation of this loop body, under the assumption that the path to this node is infrequent. The "tail" path of node $n3$ (that is, all the paths starting from node $n3$) contains node $n6$ and introduces a second entry point to the desired hyperblock, violating in this way the first condition on hyperblock formation. Tail duplication, replicates all the nodes in the tail path(s) to eliminate all the entry points in the hyperblock. Figure 3.6(b) shows the body of the loop after the tail duplication. Node $n6$ is replicated as node $n6'$ allowing the creation of two hyperblocks. Tail duplication does not increase the dynamic instruction count, but it does increase the static code size of a program and can adversely affect the instruction cache performance. To limit the code increase, Mahlke *et al.* [MLC$^+$92] suggest limiting the duplication to make at most one copy of each basic block.

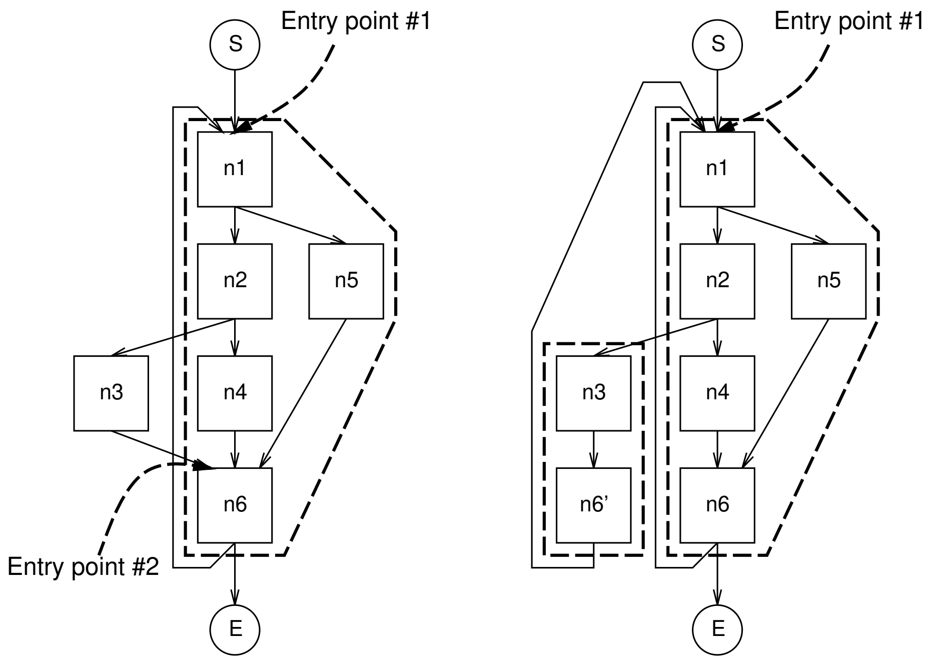


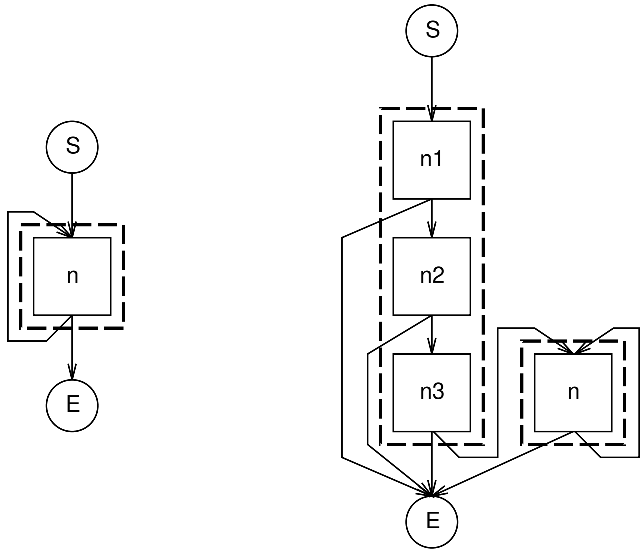
Figure 3.6: Tail duplication example. The selected hyperblock (outlined with a dashed line) in (a) has two entry points. Tail duplication (b) eliminates the second entry point by replicating node $n6$.

3.3.3 Loop peeling

Loops that are executed frequently but only iterate a few times can be main sources of mispredictions. For example, the SPEC benchmark *Gcc* contains many loops that generate code for the arguments in a function call of the compiled program; since the number of parameters is unbounded, a loop is used to handle the general case. In practice however, the average number of parameters to a function call is fairly small, and the loop only iterates a few times. A counter based branch prediction mechanism will always predict the loop-closing branch as taken, generating one incorrect prediction every time the loop terminates. Since the loop iterates just a few times, the prediction accuracy for the loop-closing branch will be very low. Loop peeling alleviates this situation using a limited form of loop unrolling.

In loop-peeling, a small number (N, called the peeling factor) of the initial iterations of the loop are unrolled. After the N peeled iterations, the original loop body is inserted. If-conversion can eliminate the forward branches in the first $N - 1$ iterations leaving only the last branch which redirects the execution to the real loop body. In cases where the loop iterates fewer than N times (which is expected to be the common case), all the computation is captured by the unrolled and if-converted part of the loop, and instructions are executed in a sequential fashion, eliminating any branch misprediction that would occur in the original code. In cases where the loop iterates more than N times, the original body of the loop executes the remaining loop iterations.

Figure 3.7 shows an example of loop peeling. Part (a) shows the control flow graph for a simple loop and part (b) shows the peeled version for a peel factor of 3. The dashed lines in the peeled version



(a) (b)

Figure 3.7: Loop peeling example. Part (a) shows the original unpeeled control flow graph and Part (b) shows the same loop peeled three times. The dashed lines outline possible if-conversion regions.

indicate possible if-conversion regions. The larger if-conversion regions in Figure 3.7(b) would capture all executions of the loop that iterate less than four times. Note that the simple loop structure in this example was used for the sake of simplicity. The loop peeling transformation can be used even when the loop body has an arbitrarily complicated control structure.

A good choice of the peeling factor N is very important for the effectiveness of this optimization. If the chosen value of N is less than the average number of iterations, I, peeling will be only partially successful. If N is larger than I, the code expansion will be larger, and the $N - I$ peeled iterations will be uselessly executed. The peeling factor for each loop can be accurately determined using profiling information. However, the usefulness of this optimization is limited when profiling information is not available, reducing the general purpose value of the transformation.

3.3.4 Control-tree height reduction

Control-tree height reduction [SKA94, SK95] is a technique that reduces the time to evaluate complex condition expressions. The C programming language employs short-circuit semantics for control expression evaluation. Compilers also employ short circuit evaluation for expressions in an attempt to reduce the instruction count of the generated program. For example, the condition:

$$(a < b) \ \&\& \ (c > d) \ \&\& \ (e <= d) \ \&\& \ (a < c) \tag{3.1}$$

would be evaluated using the four conditional branch instructions shown in Figure 3.8(a). After if conversion, the four branches would be converted into the four `set` shown in Figure 3.8(b). All these `set` instructions are dependent and must therefore be executed sequentially.

```
         bge a, b, Next                 slt      g1, a, b
         ble c, d, Next                 sgt g1 ? g2, c, d
         bgt e, d, Next                 sle g2 ? g3, e, d
         bge a, c, Next                 slt g3 ? g4, a, c
            .                             .
            .                             .
            .                             .
     Next:                          Next:

              (a)                           (b)
```

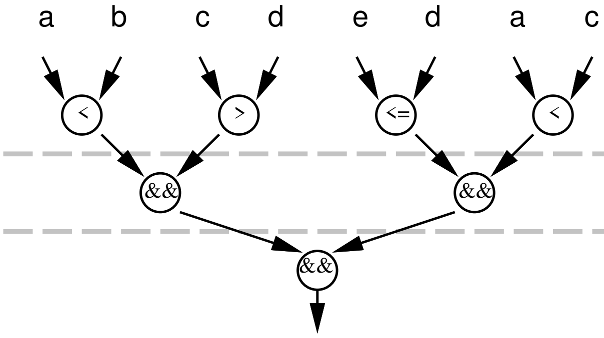
Figure 3.8: Short circuit condition evaluation. Part (a) shows the assembly for the short circuit condition evaluation of condition 3.1. Part (b) shows the if-converted set sequence for the same condition.

Expression 3.1 can be represented using the expression tree of height three, show in Figure 3.9. Control-tree height reduction is similar to the technique that was proposed by Kuck to reduce the time to evaluate expression trees [Kuc78]. This technique works by evaluating all the leaf nodes in parallel during the first step, and continues with the evaluation of the next level until the root of the tree is reached. In this way, the expression tree (control or computation alike) is evaluated in logarithmic time. Using the control-tree height reduction technique, Expression 3.1 can be represented with the tree in Figure 3.9(a), and it can be evaluated in just 3 cycles using the code sequence shown in Figure 3.9(b)

The control height reduction can be even more effective using the special set instructions and semantics provided by the PlayDoh architecture [KSR94]. As described in Chapter 2, the PlayDoh architecture defines an action specifier which, together with the result of the comparison, determine whether the destination register should be left unmodified, and if not, they determine the value that should be written to it. The possible actions include the functions OR, AND and their complements $\overline{OR}$ and $\overline{AND}$. These action specifiers can be used to directly map the control expression trees into set instructions. In addition, the PlayDoh architecture allows multiple writes to single register to be performed in a single cycle, as long as they all write the same value. Using these semantics, an arbitrary AND-tree can be evaluated by initializing a guard register to 1 and by performing all the `pred_set` instructions in a single cycle (if the processor has sufficient resources) using the AND action specifier. The semantics of the AND action specifier (defined in Table 2.2 in Chapter 2) are such that the destination register will only be written if the result is 0. That means that if more than one set instructions write the destination register, they all write the same value (zero), conforming with the PlayDoh semantics. The OR action specifier can be similarly used to speed-up the evaluation time for OR-trees. Figure 3.10 shows the `pred_set` sequence required for the evaluation of Expression 3.1. All `pred_set` instructions write register p1, using the AND action specifier. The dashes in the instructions indicate that the corresponding field is left unused.

3.3.5 Condition evaluation optimizations

The if-conversion process can introduce definitions and uses of similar conditions. For example, for each if-then-else statement two complementary conditions are generated and used, each of which must



(a)

```
set_lt   r1, a, b
set_gt   r2, c, d       1
set_le   r3, e, d
set_lt   r4, a, c
and      r1, r1, r2     2
and      r3, r3, r4
and      r1, r1, r2     3
```

(b)

Figure 3.9: Condition evaluation using control-tree height reduction.

```
pred_set.<   t1, AND, -, -, a, b
pred_set.>   t1, AND, -, -, c, d
pred_set.<=  t1, AND, -, -, e, d
pred_set.<   t1, AND, -, -, a, c
```

Figure 3.10: Condition evaluation using `pred_set` instructions.

be set separately. A peephole optimization step can identify similar conditions, and make the transformations necessary to eliminate all but one of them. For the case of an if-then-else statement, the only transformation needed is the inversion of all the uses of the guard condition in one of the *then* and *else* parts of the code. Figure 3.11 illustrates the condition evaluation optimization.

3.3.6 Exit coalescing

In loops with many exit points and arcs, a limited form of trace scheduling can be performed assuming that the exits are infrequent. Instead of leaving all the exit branches and executing them one after the

```
set_eq         r10, r1, r2
set_ne         r11, r1, r2
g_add   r11?   r3,  r4, r5
g_sub   r10?   r6,  r7, r8
```

(a)

```
set_eq         r10, r1, r2
g_add   !r10?  r3,  r4, r5
g_sub    r10?  r6,  r7, r8
```

(b)

Figure 3.11: Condition evaluation optimization for an if-then-else statement. In part (a), `r10` and `r11` are set to complementary conditions; in part (b) `r11` is eliminated and the guard condition of the `add` instruction is negated.

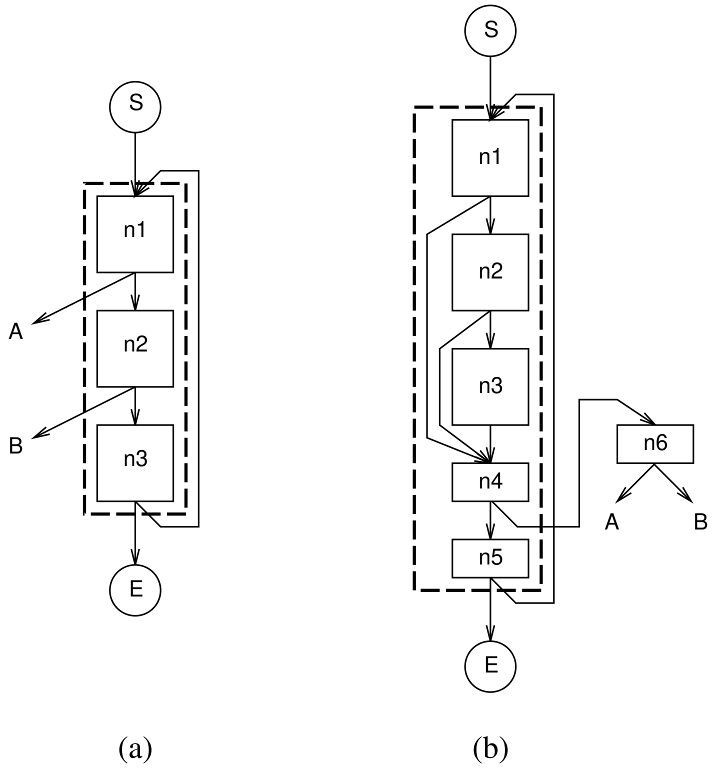


Figure 3.12: Exit coalescing example. Part (a) shows the original control flow graph of a loop with exits to two targets A and B. In part (b) the two exits are coalesced into one, node $n6$, which in turn will direct the control flow to the right target.

other to find out that in the common case none of them was taken, all the exit arcs are replaced with a single exit branch controlled by the OR of all the individual exit conditions. Once the loop is exited, the original exit sequence has to be executed to determine which one of the exit arcs should be followed. In processor implementations that are limited to execute a single branch instructions per cycle, exit coalescing can considerably improve the performance of loops with multiple, infrequent exit arcs, at the expense of a longer latency when the exits are indeed taken. Figure 3.12 shows an example of exit coalescing. Part (a) shows the original control flow graph for a loop with exits to targets A and B. The transformed control flow graph in part (b) shows the two exits coalesced into one, node $n6$. Node $n6$ is then responsible to demultiplex the control flow to the actual target.

3.3.7 Predicate promotion

If-converted code is restrictive in that it does not allow the execution of an instruction before its guard condition is evaluated. In the presence of long latency instructions such as multiplications or load instructions that may miss in the level-1 cache, this restriction can limit the performance. Code promotion, initially proposed by Tirumalai *et al.* [TLS90] and subsequently used by other research groups [MLC^{+}92, CHPC95, HMG^{+}95], bypasses this limitation by moving the guarded computation before the guard condition is evaluated and executing them unconditionally. Since the dependency on the guard register is removed, the instruction can be moved as far back as the remaining data dependences permit. This code transformation is essentially global code motion similar to Nicolau's Percolation Scheduling [Nic85]. (An excellent overview of global scheduling techniques can be found in Michael Smith's thesis [Smi92].)

Promoted instructions are always executed and produce results, even when the actual guard condition is false. As in the case of global scheduling, an instruction cannot be promoted unless the movement is *safe* (i.e., it will never generate an exception) and it is *legal* (i.e., the promoted instruction will not clobber the value of register that is not dead). Figure 3.13 illustrates the cases of unsafe and illegal promotions. In part (a), the promotion of the `load` instruction may cause an exception; in part (b), the promotion of the `sub` instructions will clobber the value of register `r2`, which is live along the path from $n1$ to $n2$. Unsafe instructions can be promoted only if there is sufficient architectural support, for example support for non-excepting instructions, or sentinel scheduling as described in Chapter 2. Instructions that would clobber live register values can be made legal by statically renaming the destination register and all its possible uses along the original guarded path; for example, the `sub` instruction in Figure 3.13(b) can be promoted as shown in Figure 3.13(c); `r2`, the destination register of the sub instruction is renamed to `r9` and the `sub` instruction is promoted to node $n1$. All the uses of `r2` along the False path from node $n1$ must also be renamed to `r9`, as shown for the `and` instruction.

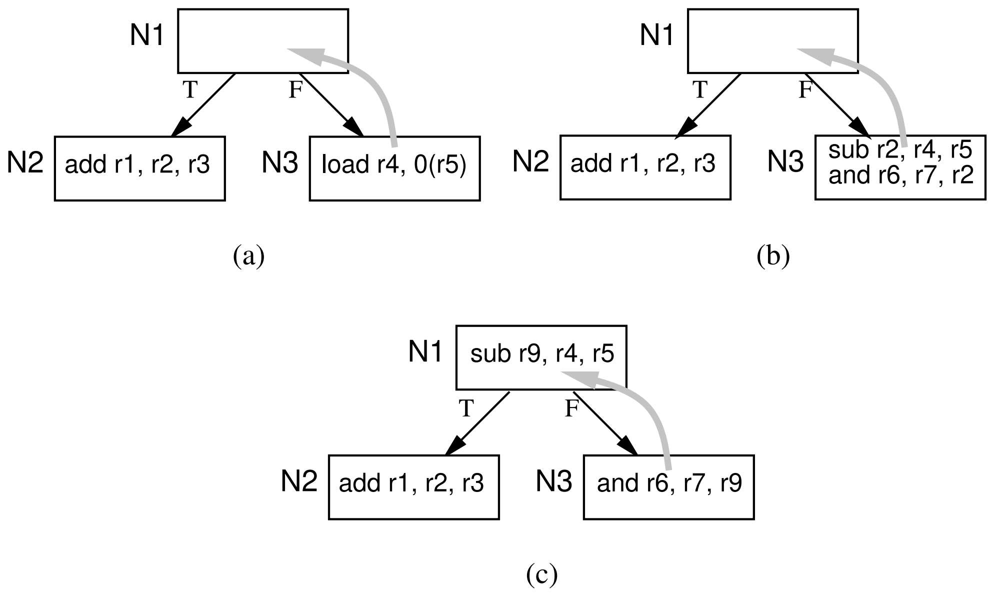


Figure 3.13: Instruction promotion example: case (a) is unsafe because the load may cause an exception, case (b) is illegal because r2 is in use along the other path. In case (c) the promotion of the `sub` instruction from case (b) is made legal by statically renaming register r2 to r9.

Predicate promotion can also be supported directly in hardware. Ando *et al.* [ANHN95] proposed "Predicate State Buffering" which allows a guarded instruction to execute before its guard operand(s) is available. In Predicate State Buffering, when a promoted guarded instruction is executed, the result, tagged with the guard condition, is stored into a "speculative" version of the destination register. When the guard operands become available, the tag in the speculative version of each register evaluates the guard condition, and determines whether the speculative state should be committed into the architectural state or should be squashed.

3.4 Scheduling guarded code

Once the code is if-converted, the control dependences within the if-converted regions are converted to data-dependences, allowing traditional scheduling algorithms, such as greedy prioritized scheduling, to be used to produce the final scheduled output. These algorithms can be used almost unchanged, with minor modifications that depend on the idiosyncrasies of the underlying execution hardware.

If the target instruction set does not support guarded instructions, the necessary branches can be introduced in the scheduled code using the *Reverse if-conversion* technique [WMHR93]. Although this transformation negates the branch eliminating benefits of if-conversion, it retains some of the scheduling benefits and can be used as an alternative to global scheduling for architectures that do not support guarded execution.

3.5 Summary

In this chapter we described the structure and operation of an if-converting compiler. To get effectively utilize guarded execution, the compiler uses optimizations that (i) restructure the control flow graph before if-conversion is performed, and (ii) reduce the amount of instructions required to evaluate the guard conditions after the if-conversion is performed. The compiler can also take advantage of special ISA support for guarded execution, such as the `pred_set` instructions of the PlayDoh instructions set, to generate more efficient guarded code. Profiling information may be used to evaluate the relative merit of possible actions and is critical to the effectiveness of many of the optimizations. Profiling information can also used to reduce the amount of wasted computation that is introduced by if-conversion. However, in the absence of profiling information the compiler is limited to conservative decisions, or can resort to the use static prediction and heuristics to determine the best course of action.

In this thesis we make the assumption that profiling information is not available, as we feel that in a casual, every-day use of the compiler profiling in not an option. This assumption restricts the range of optimizations that can be effectively used. The scheduler we use for our evaluation (which we will describe in Chapter 5 Section 5.3.1 implements a limited form of the control-tree height reduction, predicate promotion and some condition evaluation optimizations.

Chapter 4
GUARD instructions

The last two chapters described how guarded execution can increase the ability of the compiler to generate better instruction schedules and to smooth the flow of instructions in the processor. However, as we mentioned earlier, one of the biggest obstacle to the widespread use of guarding is that it cannot be easily incorporated in the successful existing instruction set architectures. Summarizing section 2.1.3, guarded execution specified using explicit guard condition operands with each (guarded) instruction has the following limitations: (i) it consumes a considerable amount of valuable instruction space, (ii) it increases the dynamic instruction count for a program (due to predicate set instructions and guarded computation), consuming instruction fetch, decode and possibly execute bandwidth, (iii) it requires an additional operand for each guarded instruction, requiring an additional read port in the register file for every instruction issued per cycle, and (iv) it uses registers to hold the value of guard conditions and increases the pressure on the architectural register file, unless a separate condition register file is defined in the architecture). In this chapter, we propose a new class of instructions called "GUARD" instructions which alleviates or completely overcomes these limitations, and allows a designer to incorporate guarding in either a new or an existing instruction set.

GUARD instructions are based on two observations (which we will verify in Chapter 5). The first observation is that guarded computation exhibits a form of spatial locality: instructions guarded by the same condition are likely to be in close proximity, both in the static and in the dynamic instruction stream. The second observation is that instructions in close proximity are likely guarded by the same guard condition or by a small number of guard conditions, in either true or complement form. These program properties arise from the common programming practices which favor small, modular code structures, and by the natural nesting of control structures that limit the scope of movement for the instructions (either guarded or not). The above observations indicate that a "forward" specification of guarding, in which the guard condition is specified in advance of the guarded computation, is possible. GUARD instructions implement such a forward specification, communicating to the processor both a guard condition and the list of instructions that are guarded by it. The benefits of this forward specification will become clear shortly. Next, we define the syntax and semantics of GUARD instructions, and show how they can be used to specify guarding.

4.1 Semantics of GUARD instructions

A generic GUARD instruction has two operands, a guard condition to be evaluated, and a guard list, which specifies which of the subsequent instructions (in the static, as well as in the dynamic instruction sequence) are guarded by the specified condition. The semantics of a GUARD instruction are: evaluate the condition, and if it is false, squash all the instructions in the guard list. For example, a GUARD instruction which evaluates whether a condition is true and if not squashes a set of instructions can be written as:

```
GUARDTRUE cond, i1, i2, i3, ...
```

where $i1$ $i2$, $i3$, ..., are the labels of the instructions that should be guarded by the condition $cond$.

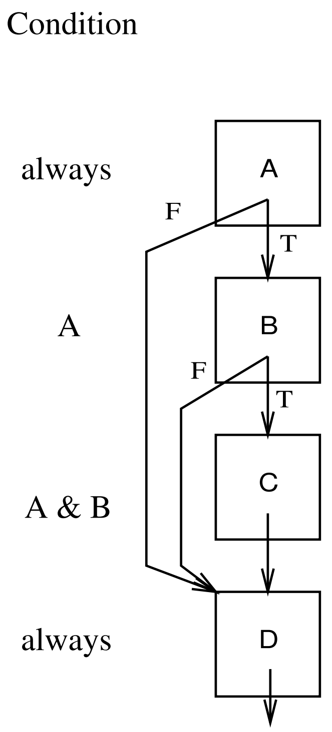
4.1.1 A small example

The following example illustrates the use of GUARD instructions. The simple control flow graph in Figure 4.1(a) consists of four basic blocks forming two nested if structures. The column labeled "Condition" indicates the guard condition for each of the basic blocks in the graph. To specify the guarded execution of basic blocks B and C, we need two GUARD instructions, one for each of the conditions A and $A\&B$. Figure 4.1(b) shows the assembly code for a MIPS-like instruction set without using guarding, while Figure 4.1(c) shows the if-converted assembly code with guarding specified using GUARD instructions. In Figure 4.1(c), we assume that register `r3` holds the condition A, and register `r5` holds the condition B. To specify the guarding of the instructions in basic block B, the first GUARD instruction lists the labels $i3$, $i4$ and $i5$ in its guard list. Similarly, the second GUARD instruction lists the labels $i6$ and $i7$, to specify the guarding of basic block C with the condition $A\&B$.

Comparing Figures 4.1(b) and (c) we notice that the non-branch instructions in both cases are identical in every respect; the only differences between the two assembly listings are the elimination of the branch instructions, the use of the AND instruction to evaluate the guard condition $A\&B$ in r3, and the use of GUARD instructions to specify guarding.

4.1.2 Features of GUARD instructions

The previous example, although simple, illustrated the power of GUARD instructions. Consider the limitations of guarding as described in Chapter 2, and re-iterated in the beginning of this chapter. All these limitations are artifacts of the way guarding is specified, and can be alleviated or completely eliminated by the use of GUARD instructions. The key property of the GUARD instruction is that it specifies the guard condition for many (subsequent) instructions. Several benefits stem from this property. First, the GUARD solution requires the addition of a very small number of instructions, and no modifications to any pre-existing instructions in the instruction set. This feature allows guarded execution, in its full form, to be easily integrated into existing (or new) instruction sets. Second, the processor is informed in advance that some of the instructions will be squashed, and can avoid even fetching them, proceeding with the fetching of instructions that will be useful. This *early-out* capability is very important, because it allows the compiler to use guarding more aggressively, relying on the hardware to ensure that extensive use of guarding does not result in too much dynamic overhead. (For example, in the guarded code of Figure 4.1, if the condition in r3 evaluates to false, the processor could jump to i8 after it is done with i5, bypassing the execution of instructions i6-i7 since these two instructions will be dynamically transformed into NOPs.) Third, since the computation is guarded in an indirect way, the guard register is read once by the GUARD instruction and is not read by each individual guarded instruction, obviating the need for the additional read port in the register file of the processor. Finally, since the guard register is read only once by the GUARD instruction, it can be immediately re-used for other computation, reducing the importance of a separate condition register file. In this thesis, we propose using the general purpose register file to hold both regular computation and guard conditions.



(a)

Assembly Code

```
i1:  ld    r6,  0(r2)
i2:  add   r1,  r2,  #2
     beq   r7,  zero, Label

i3:  ld    r3,  0(r1)
i4:  or    r17, r17, r3
i5:  sw    r17, 0(r1)
     beq   r5,  zero, Label

i6:  mov   r1,  r3
i7:  sub   r6,  r6,  #1

Label:

i8:  add   r7,  r7,  1
i9:  add   r5,  r5,  1
```

(b)

Assembly Code
Using GUARDs

```
     and     r3,  r7, r5
     GUARD   r7,  i3, i4, i5
     GUARD   r3,  i6, i7
i1:  ld      r6,  0(r2)
i2:  add     r1,  r2,  #2

i3:  ld      r3,  0(r1)
i4:  or      r17, r17, r3
i5:  sw      r17, 0(r1)

i6:  mov     r1,  r3
i7:  sub     r6,  r6,  #1

i8:  add     r7,  r7,  1
i9:  add     r5,  r5,  1
```

(c)

Figure 4.1: The use of GUARD instructions. Part (a) shows a simple control flow graph, part (b) lists the corresponding assembly for a MIPS-like instruction set, and part (c) shows the if-converted assembly using GUARD instructions.

4.2 ISA support for GUARD instructions

To introduce GUARD instructions in an instruction set, we must be able to encode the GUARD opcode, the guard condition and the guard list using or modifying the existing formats of the instruction set. Encoding GUARD opcodes can be easily achieved by allocating one (or more) of the unused opcodes in the instruction set, while encoding of a guard condition can be achieved using one (or more) of the register specifier fields in the instruction. However, a realistic instruction set cannot directly encode a list of multiple instruction labels in a single instruction. Instead, we can can take advantage of the spatial locality of the instructions guarded by the same condition, and encode the guard list using a bit-mask. This *guard mask* indicates which of the instructions following the GUARD instruction are guarded by the specified guard condition. Ideally, an infinite mask would be specified. However, given the likely proximity of the instructions guarded by the same condition, even a limited-size mask is likely to be an effective way to encode the list of guarded instructions. For example if the mask size is 10 bits, the syntax of a GUARD instruction would be:

```
GUARD cond, 0011010100
```

indicating which four of the instructions that follow are guarded with condition *cond*.

4.2.1 New instructions

The exact number of GUARD instructions that must be added to an instruction set to efficiently support guarding, and the nature of encoding of the mask field are tradeoffs between larger opcode space requirements and more compact and efficient guarding representation.

The simplest form of a GUARD instruction specifies a single condition and its guard mask. The simplest GUARD opcodes are GUARDTRUE and GUARDFALSE, specifying guarding on the condition being true and false respectively. (The GUARD instructions used in Figure 4.1 are actually GUARDTRUE instructions.) The guard mask encodes the two states for each instruction: not guarded or guarded. Therefore, we can use a unary encoding for the guard mask, in which the *i*th bit specifies whether the *i*th instruction following the GUARD instruction is guarded by the specified condition.

In many common cases such as if-then-else statements, we need to guard some instructions on the condition being true and some on the condition being false. Using only GUARDTRUE and GUARDFALSE instructions, we need two GUARD instructions for what intuitively is a single action. To achieve guarding both paths with a single instruction, we can introduce a GUARDBOTH instruction. The guard mask of a GUARDBOTH instruction must specify three states for each instruction: not guarded, guarded on true and guarded on false. Clearly, being able to specify guarding on both true and false conditions in a single instruction will reduce the number of GUARD instructions executed in a program. However, the encoding of three states for each instruction requires more bits in the guard mask, and for a fixed-size mask it would restrict the scope of the GUARDBOTH compared to the scope of the GUARDTRUE and GUARDFALSE instructions. The simplest way to encode the guarding state in the mask is to use 2 bits per guarded instruction. Alternatively, we can take advantage of the unused fourth state, and reduce the guard mask bit requirements. For example, the number of possible guarding states for three instructions are 3^3 (27); all these states can be encoded using 5 instead of 6 bits. This encoding increases the scope of the guard mask by at least 16% but requires a (relatively simple) decoding step before the guard mask can be used.

For a MIPS-like instruction format outlined in Figure 2.4, the opcode field is 6 bits, and one register specifier field is 5 bits, leaving up to 21 bits that can be used as the guard mask. With this mask size, a GUARDTRUE (or GUARDFALSE) instruction can guard up to 21 instructions. A single GUARDBOTH instruction using the compact encoding can guard up to 12 instructions (4 sets of 5 bits, each specifying the guarding state of 3 instructions) from both paths of a single branch. When the guard distance is larger than the mask, the compiler can insert additional GUARD instructions in the code. This can be easily done by extending the lifetime of the guard register and introducing additional GUARD instructions later in the code. An alternative to extending the lifetime of the guard register is to use a GUARDUPPER instruction, which specifies that the mask is first shifted by N bits, where N is the width of the GUARD mask in instructions. For example, if we only have a 5-bit mask, the instruction:

```
GUARDTRUE cond, 0011010100
```

can be expressed using the following instruction sequence (assuming that the GUARDUPPER mask will be shifted right):

```
GUARDTRUE        cond, 00110
GUARDUPPERTRUE   cond, 10100
```

For the assumed MIPS-like format, N would be 21 for GUARDUPPERTRUE and GUARDUPPERFALSE and 12 for GUARDUPPERBOTH.

We can also allow GUARD instructions to perform more powerful tests, for example compare if two operands are equal, etc. As described in Chapter 2, Section 2.2.2, allowing a more powerful condition specification will further reduce the number of set instructions required for guarding the code, but it will also reduce the size of the mask, reducing in this way the scope of the GUARD instructions. A more complicated condition might also affect the critical path of GUARD execution, as the condition evaluation will require more logic levels. For these reasons, in the remaining part of this thesis we only consider GUARD instructions that perform a simple true or false test on a single operand.

4.2.2 New ISA state

To support GUARD instructions, the processor has to keep track of the set of live and squashed future instructions. To achieve this, the processor can maintain a *scalar mask register* (akin to the vector mask of vector processors). Each bit in the scalar mask has the following meaning: if the i-th bit in the shift register is 1, the i-th instruction (counting from the current program counter) is to be executed; if the i-th bit is 0, the i-th instruction must be treated as a NOP. For the execution of a GUARD instruction with mask $mask$ evaluating condition $cond$, the bits in the scalar mask are updated as follows:

$$\begin{aligned} scalar\_mask_i &= scalar\_mask_i \ \& \ ((mask_i \ \& \ cond) \mid \overline{mask_i}) \\ &= scalar\_mask_i \ \& \ (cond \mid \overline{mask_i}) \\ &= scalar\_mask_i \ \& \ (\overline{\overline{cond} \ \& \ mask_i}) \end{aligned} \tag{4.1}$$

The intuition behind Equation 4.1 is that for every GUARD instruction, a set bit in the guard mask indicates that the instruction is to be executed only if the condition holds. A reset bit in the guard mask indicates that the state of the instruction is unaffected by this GUARD instruction.

After an instruction is completed, the scalar mask is shifted by one position, with a one being shifted in. The width (in bits) of the scalar mask register should be as wide as possible, to allow a larger guarding range for the GUARD instructions. However, as we will describe in Section 4.4.2, the scalar mask register must be saved and restored on interrupts and exceptions, so it is convenient to define that it is as wide as the general purpose registers of the architecture, so it can be easily manipulated with existing instructions. The scalar mask register may also be modified by branch instructions; we will discuss the interaction of GUARD and branch instructions in Section 4.4.1.

The scalar mask is key in permitting the processor to effectively skip around unnecessary computation. The processor can identify the unnecessary computation by performing a count of the leading zeros in the scalar mask, and can execute a short branch, changing the fetch address to $PC + count * 4$ (*count* gives the offset from the current PC to the next useful instruction). Therefore, instructions that will dynamically be transformed into NOPs are not even fetched into the pipeline.

Even when it is too late to stop the fetching of unnecessary computation, decoding the instructions as NOPs has significant advantages, because the decoding logic does not have to be conservative about possible dependences through guarded code. Consider for example this code sequence:

```
g_add  R10 ?   R3, R2, R1
sub            R5, R3, R1
```

where R10 is equal to zero. While the guarded `add` has no effect, it appears to write register R3, so the following `sub` instruction will have to wait for the possible result, and the code sequence executes in two cycles. However, at decode time, the scalar mask register will indicate that the guarded add should be decoded as a NOP, and the dependency checking logic can remove the artificial RAW dependence, allowing this code sequence to execute in a single cycle. Similarly, the scalar mask can be supplied to a *merging* or *collapsing* network [Joh90, CMMP95], which would completely remove from the pipeline any squashed instructions that have being fetched, before they are even decoded.

The combination of GUARD instructions and the scalar mask register allows an additional optimization. When multiple GUARD instructions list the same instruction in their respective guard masks, the conditions are implicitly AND-ed in the scalar mask. This feature can reduce the number of logic manipulation instructions and temporary registers required to evaluate the appropriate guard conditions. For example the sequence

```
GUARD  A, i2,i4
GUARD  B, i3,i4
```

guards instruction $i2$ on condition A, instruction $i3$ on condition B and instruction $i4$ on condition $A\&B$.

4.2.3 GUARD instruction use

To illustrate how GUARD instructions are used by the processor, we use the same example used in Figure 4.1. Figure 4.2(a) shows the same control flow graph and the guard conditions as Figure 4.1(a). The guard masks corresponding to conditions A and $A\&B$ are shown vertically in Figure 4.2(b). In these masks, a 1 indicates that the corresponding instruction is guarded by the condition, and a 0 indicates that the instruction is not dependent on the condition. Part (c) shows the MIPS-like assembly code using the binary masks to specify the guard lists of the GUARD instructions.

Figure 4.3 illustrates the execution of the code of Figure 4.2(c) annotated with the corresponding values of the scalar mask register. In Figure 4.3 we assume that initially condition A is true (i.e., register r7 is equal to 1) and condition B is false (i.e., register r5 is 0, and therefore register r3 will be set to 0). With these assumptions, the instructions in basic block C (i.e., the `mov` and the `sub`) must be squashed during the execution of this code sequence (these instructions are marked with a star in the assembly code). The initial value of the scalar mask register is assumed to be all ones. We also assume that the leftmost bit of the scalar mask register is the one determining whether the current instruction should be executed or squashed. The scalar mask register is not modified by the first GUARD instruction since its condition (r7) evaluates to true. However, the condition of the second GUARD instruction evaluates to false, and the scalar mask bits are updated to reflect that the two instructions in the guard mask of the GUARD instruction should not be executed. The execution continues with the scalar mask register being shifted one position to the left for each executed instruction. When the execution reaches the two instructions of basic block C, the leftmost bit of the scalar mask register is zero, indicating that the corresponding instructions must be squashed.

In the example above, the condition $A\&B$ that guards basic block C was explicitly computed by the first instruction in the code sequence. Taking advantage of the implicit AND property of GUARD instructions, the same effect can be achieved by using just two GUARD instructions:

```
GUARD r7, 0001111100  # Condition A
GUARD r5,  000001100  # Condition B
```

Note that the bits corresponding to instructions $i6$ and $i7$ are set in both masks, indicating that these instructions should be squashed when any of the conditions A or B is false.

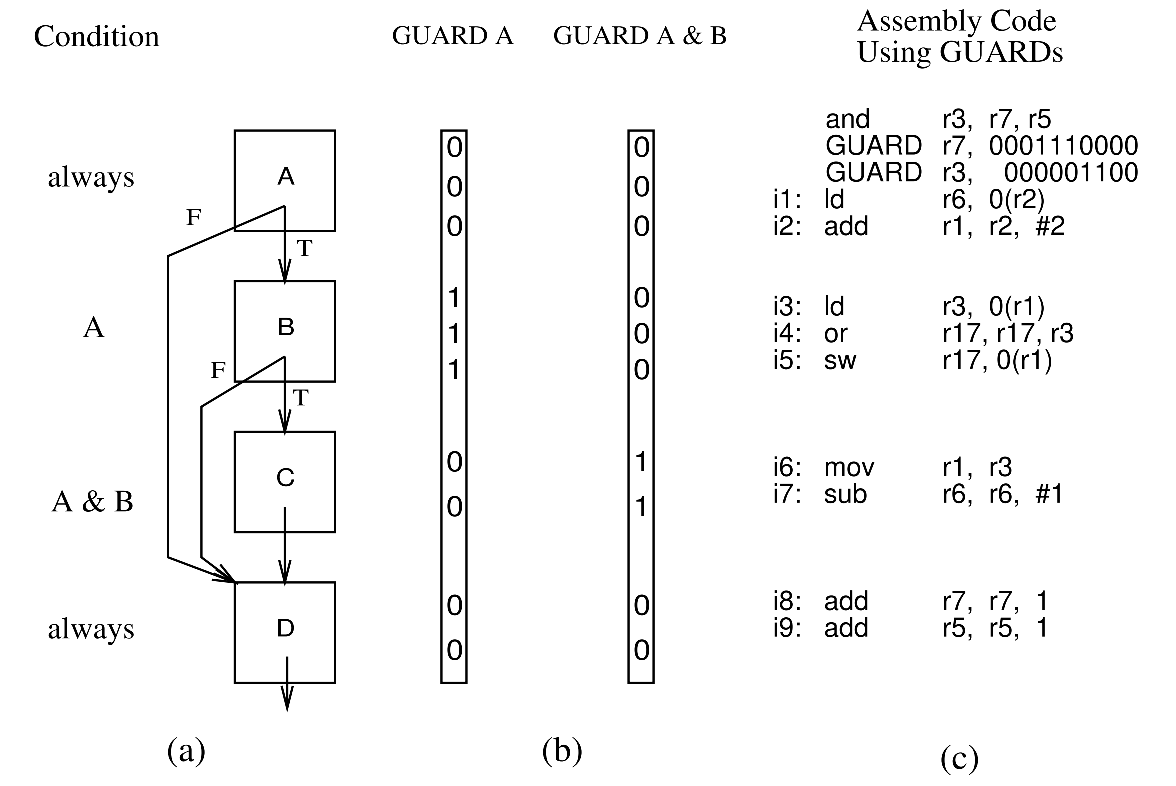


Figure 4.2: The use of GUARD masks. Part (a) shows the control flow graph and the guard conditions from Figure 4.1. Part (b) shows the guard mask corresponding to each condition and part (c) shows the MIPS-like assembly using GUARD instructions with binary masks.

4.3 GUARD instruction assignment

To effectively use GUARD instructions, the compiler has to (i) insert the necessary GUARD instructions in the appropriate places in the code, and (ii) determine the guard mask for each of them. (Internally the compiler uses guard lists in the GUARD instructions; these lists will be converted into masks by the assembler when the actual instruction arrangement is known. In this section we use the term mask to denote either the compiler guard lists, or the actual guard masks generated by the assembler, depending on the context.) One simple solution to both these problems is to run the RK algorithm (described in Chapter 3) and assign a GUARD instruction for each of the conditions generated by the RK algorithm. The GUARD instructions can be placed immediately after the definition of the corresponding guard registers, and the GUARD masks are easily determined, since they contain all the instructions that would be guarded by that guard register. While this approach would yield correct code, it does not fully exploit the potential of GUARD instructions. In particular, it would not exploit the implicit AND-ing in the scalar mask register. To determine a placement of GUARD instructions and a mask assignment that exploits this property, we developed the following algorithm.

Code with GUARDs	Scalar Mask Register
`and   r3,  r7, r5`	1111111111
`GUARD r7,  0001110000`	1111111111
`GUARD r3,   000001100`	1111111111
`ld    r6,  0(r2)`	1111100111
`add   r1,  r2, #2`	1111001111
`ld    r3,  0(r1)`	1110011111
`or    r17, r17, r3`	1100111111
`sw    r17, 0(r1)`	1001111111
`mov   r1,  r3       *`	0011111111
`sub   r6,  r6, #1   *`	0111111111
`add   r7,  r7, 1`	1111111111
`add   r5,  r5, 1`	1111111111

Figure 4.3: The execution of the code of Figure 4.2(c) assuming that the starting values of register r7 (condition A) and r5 (condition B) are 1 and 0 respectively. The left column lists the value of the scalar mask register when the instruction is considered for execution. A zero in the leftmost position of the scalar mask register indicates that the instruction should be squashed (marked with a star).

4.3.1 An algorithm for determining GUARD masks

The intuition behind GUARD instructions is that when an instruction is listed in the guard mask, it must be squashed if the condition does not evaluate to true. To map this behavior to the original control flow graph, we observe that it is very similar to the behavior of a conditional branch. In the CFG of a program before the if-conversion, every conditional branch guarantees that, depending on the branch condition, but independent of other conditional branches, some instructions will not be executed. Therefore, we can simulate the branch behavior if we add a GUARD instruction for every conditional branch in the original code, and we add in the guard mask all the instructions that are guaranteed not to execute when the branch is taken. We formalize this process for GUARD mask assignment in the algorithm shown in Figure 4.4.

The algorithm assumes that the input G is a directed acyclic graph with a single entry point. (Since this algorithm is applied to the same regions as if-conversion would, it is reasonable to assume that the graph will be loop-free.) The algorithm considers every node A in G with two outgoing arcs and determines the necessary GUARD instructions and masks. We prove that the GUARD assignment determination is correct, by showing how the algorithm takes the correct decisions for all possible cases. To determine whether a node X (where X is not A) should be included in the guard mask of the GUARD instruction in node A, we distinguish three possible cases.

- **Case 1:** node X is not reachable from node A, regardless of the A's outcome. Then the execution of node X is determined by other conditionals in G and X should not appear in the GUARD mask in A. Since X is not reachable from node A, it will not be added to the guard mask of the generated GUARD instruction.

```
Algorithm guard-assign(G):
    Given a rooted, acyclic graph G,
        determine the placement and masks of the GUARD instructions
{
(1)   for each conditional node N in G {
(2)       Treach(N) = reachable(cond(N) == true);
(3)       Freach(N) = reachable(cond(N) == false);
(4)       Tsquash(N) = Treach(N) - Freach(N);
(5)       Fsquash(N) = Freach(N) - Treach(N);
(6)       if (Tsquash(N) is not empty)
(7)           insert a GUARD  cond(N), Tsquash(N) in node N
(8)       if (Fsquash(N) is not empty)
(9)           insert a GUARD !cond(N), Fsquash(N) in node N
    }
}
```

Figure 4.4: An algorithm for GUARD instruction and mask assignment.

- **Case 2:** node X is reachable from node A both when A's branch condition evaluates to true and when it evaluates to false. Then node X is not control dependent on node A, and consequently X should not appear in the GUARD mask in A. Since X is reachable from A when the branch condition is either true or false, it will be present both in the $Treach$ and in the $Freach$ sets, but not in their difference, so it will not be added in the guard mask of the generated GUARD instruction.

- **Case 3:** node X is reachable only when A's branch condition evaluates to true (false). Then, if node A is executed, and its branch condition evaluates to false (true) node X will not execute, therefore node X will appear in the A's guard mask for the condition being false (true). The GUARD assignment algorithm will determine that node X is a member of the $Treach$ ($Freach$) set as well as a member of the $Tsquash$ ($Fsquash$) set, so node X will be added to the guard mask of the generated GUARD instruction for the condition being true (false).

The above algorithm can be easily modified to detect when a GUARDBOTH instruction can be used (by testing whether both $Tsquash$ and $Fsquash$ are not empty) and to generate the appropriate GUARDBOTH mask by merging the two masks generated in steps 7 and 9 into a single GUARDBOTH mask. Figure 4.5 shows the modified version of the GUARD mask assignment algorithm, supporting GUARDBOTH instructions.

4.3.2 An example of GUARD assignment

To illustrate the operation of guard assignment algorithm consider the control flow graph in Figure 4.6. (This CFG is similar to the examples used by Ferrante *et al.* [FOW87] and Park and Schlansker [PS91].) The CFG is annotated with the required GUARD instructions to achieve correct guarded execution as determined by the algorithm of Figure 4.4. Table 4.1 tabulates the conditions for each of the CFG nodes, the computed $Tsquash$ and $Fsquash$ sets and the generated GUARD instructions for the same CFG.

```
Algorithm guard-assign(G):
    Given a rooted, acyclic graph G,
        determine the placement and masks of the GUARD instructions
{
     for each conditional node N in G {
       Treach(N) = reachable(cond(N) == true);
       Freach(N) = reachable(cond(N) == false);
       Tsquash(N) = Treach(N) - Freach(N);
       Fsquash(N) = Freach(N) - Treach(N);
       if ((Tsquash(N) is not empty) && (Fsquash(N) is not empty) {
          TFsquash(N) = combineTFguardlist(Tsquash(N), Fsquash(N));
          insert a GUARDBOTH cond(N), TFsquash(N) in node N
       } else {
         if (Tsquash(N) is not empty)
            insert a GUARD  cond(N), Tsquash(N) in node N
         if (Fsquash(N) is not empty)
            insert a GUARD !cond(N), Fsquash(N) in node N
       }
     }
}
```

Figure 4.5: An algorithm for GUARD instruction and mask assignment supporting GUARDBOTH instructions.

Node	Condition	Tsquash Set	Fsquash Set	Corresponding GUARD Instructions
1	t1	{2, 4}	{3}	GUARD `t1, 2, 4` GUARD `!t1, 3`
2	t2	{4}	{5}	GUARD `t2, 4` GUARD `!t2, 5`
3	t3	{5, 6}	–	GUARD `t3, 5, 6`

Table 4.1: GUARD instruction assignment determination: for every branch condition the guard assignment algorithm computes the $Tsquash$ and $Fsquash$ sets and introduces the corresponding GUARD instruction(s).

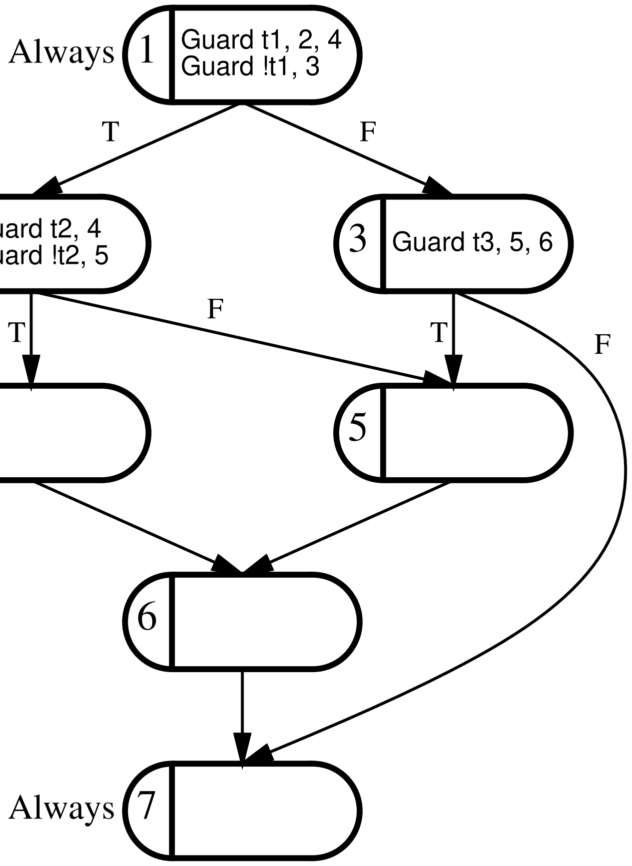


Figure 4.6: A control flow graph annotated with GUARD instructions. tn is the branch condition for node n. The guard lists in the GUARD instructions refer to node numbers.

For the three conditional nodes in the CFG, a total of five GUARD instructions are required. Furthermore, if the instruction set supports a GUARDBOTH instruction, the total is reduced to just three instructions. The results of the algorithm are straightforward. For example, node 2 is not executed when condition $t1$ is false, so it appears in the guard mask of the GUARD instruction with $t1$ as a condition. Node 4 is not executed when either of the conditions $t1$ or $t2$ are false, and so it appears in the guard masks of both the GUARD instruction with condition $t1$ in node 1 and of the GUARD instruction with condition $t2$ in node 2.

It is interesting to compare the results of traditional if-conversion as implemented by the RK-algorithm, with the result of our guard assignment algorithm. Figure 4.7 shows the same CFG as Figure 4.6, annotated with the results of the RK algorithm. In Figure 4.7, five distinct guard registers are used, requiring a total of 7 set instructions and one initialization instruction to correctly evaluate all the guard conditions. In contrast, only five GUARD instructions were required in Figure 4.6, indicating that the GUARD instructions not only can specify the guarding state of instructions, but can help lower the overhead of guarding as well.

To understand the differences between the two approaches, consider how the guarding of node

6 is handled by each one. Node 6 is executed either when $t1$ is true, or when $t1$ is false *and* $t3$ is true. To achieve this effect, the RK algorithm generates two sets for $p6$. One is in node 1 with condition $t1$ which will take care of the first condition. The other is in node 3 using the condition $t3$; the set instruction is itself guarded by $p3$, which will be true if $t1$ was false. With GUARD instructions, node 6 simply appears in the guard mask of the GUARD instruction in node 3.

4.4 Hardware considerations for GUARD instructions

The control logic required to execute GUARD instructions is fairly cheap and straightforward. The processor has to: (i) implement the scalar mask register described in Section 4.2.2, and (ii) include the circuit required to update the scalar mask register when a GUARD instruction is executed.

The scalar mask register can be implemented using a shift register which will accumulate the results of the GUARD instructions and determine the execution of future instructions. As described earlier, a set bit in the scalar mask register indicates that the instruction is to be executed while a reset bit indicates that the instruction should be squashed in the pipeline. After an instruction is completed, the scalar mask is shifted by one position, with a "one" being shifted in to indicate that future instructions are to be executed unless they are explicitly squashed by a GUARD instruction.

Figure 4.8 shows the high level diagram for a 5-stage pipeline of a processor that supports GUARD instructions. Compared to a ordinary 5-stage pipeline, the only additions are the scalar mask register and the "cond eval" circuit, which is the hardware realization of Equation 4.1 consisting of just two levels of logic. In Figure 4.8, the rightmost bit of the scalar mask register corresponds to the current instruction. This bit is sent to the forwarding logic which is responsible for the squashing of the current instruction when the bit is zero. When an instruction needs to be squashed, the pipeline control has to take two actions: (i) inhibit the write-back of the squashed instruction, and (ii) inhibit the squashed instruction from forwarding its result to a subsequent instruction. In Figure 4.8 we assume that both these actions will be performed by the modified forwarding logic.

An aggressive ILP processor must be able to execute multiple GUARD instructions per cycle. Although the scalar mask is a centralized resource, the operations performed on it are simple logic functions. Equation 4.2 extends Equation 4.1 for the concurrent execution of two GUARD instructions (where Mn and Cn are the guard mask and the guard condition of the nth GUARD instruction).

$$\begin{aligned} scalar\_mask_i &= scalar\_mask_i \ \& \ ((M1_i \ \& \ C1) \mid \overline{M1_i}) \ \& \ ((M2_i \ \& \ C2) \mid \overline{M2_i}) \\ &= scalar\_mask_i \ \& \ (C1 \mid \overline{M1_i}) \ \& \ (C2 \mid \overline{M2_i}) \\ &= scalar\_mask_i \ \& \ (\overline{\overline{C1} \ \& \ M1_i}) \ \& \ (\overline{\overline{C2} \ \& \ M2_i}) \end{aligned} \tag{4.2}$$

Notice also that the parenthesized parts of the equation do not involve the scalar mask register, and are independent for each GUARD instruction. Taking advantage of this inherent parallelism and of the associativity of the AND operation, the processor can multiple (ready) GUARD instructions using an AND-tree to update the scalar mask. As a result, the generalization of the scalar mask update equation for 4 simultaneous GUARD instructions uses just four, 2-input gate levels. Using gates with higher fan-in can further reduce the required number of levels.

Figure 4.9 shows the high level diagram for a dual issue pipeline. The complexity of executing more than one GUARD instructions per cycle is concentrated in the "cond eval" circuit, which, as described earlier, uses an AND-tree to combine the results of both GUARD instructions. Figure 4.9 also

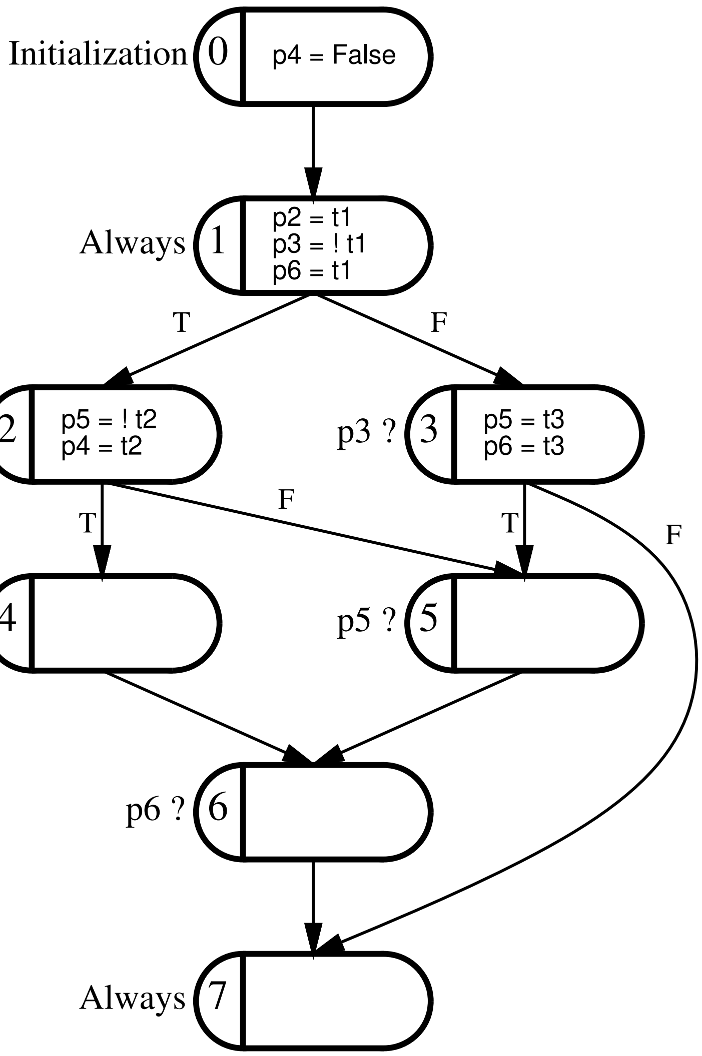


Figure 4.7: Condition register assignment and definitions by the RK algorithm. pn is the predicate register, and tn is the branch condition for node n.

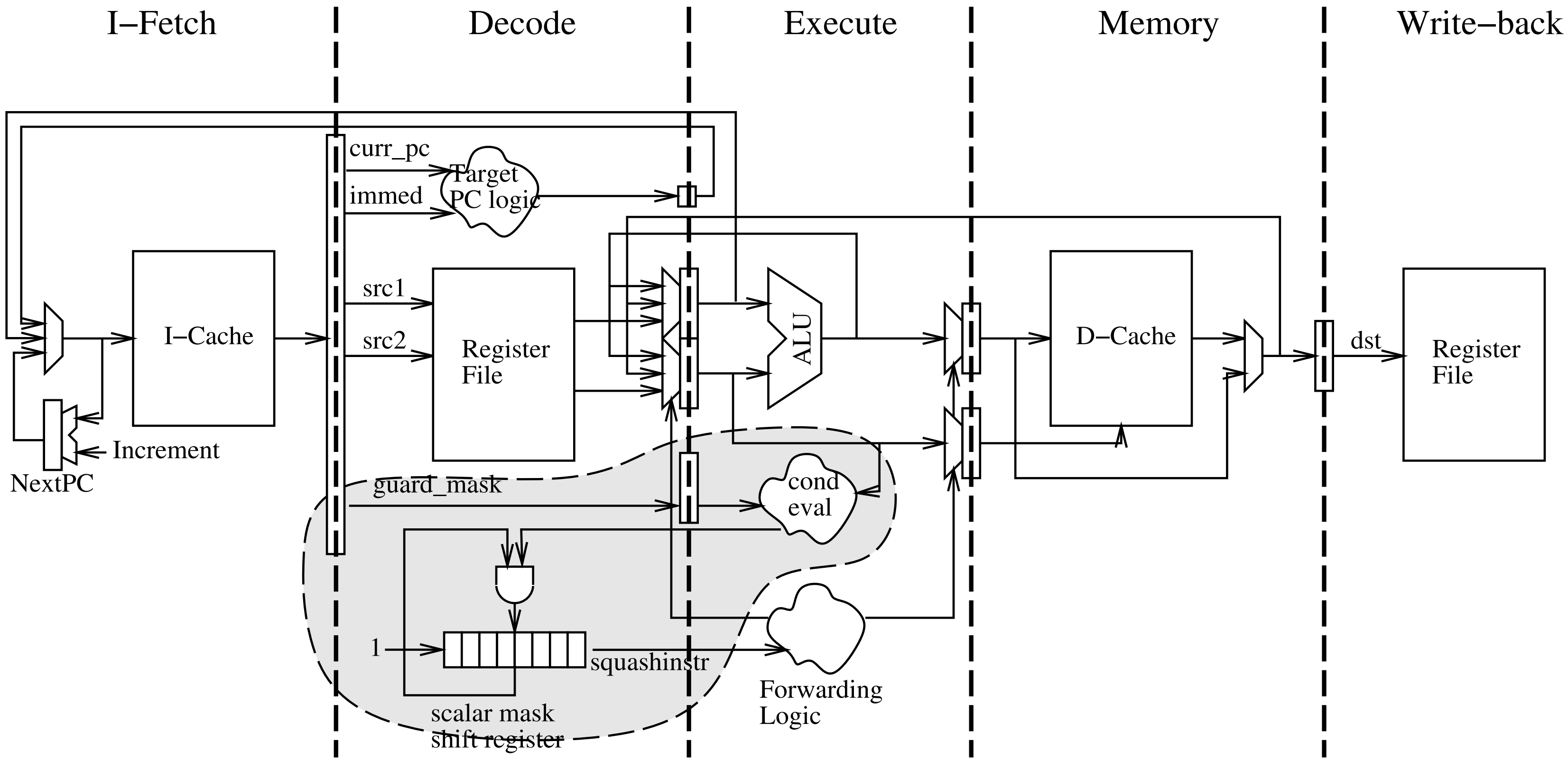


Figure 4.8: A pipeline supporting GUARD instructions. The shaded portion of the pipeline is the portion that needs to be added to support the execution of GUARD instructions.

shows how the scalar mask register can be used to skip around squashed computation. The "skip logic" circuit performs a leading zero count on the scalar mask register and adds it to the current fetch PC. In this way, the skip logic injects a short forward branch in the pipeline which bypasses the instructions that are already squashed by previous GUARD instructions.

The ability to execute multiple GUARD instructions per cycle and the implicit AND property of the scalar mask register, achieve a result that is very similar to the control-tree height reduction technique described in Chapter 3, Section 3.3.4. To evaluate an AND-tree, the control-tree height reduction technique uses multiple `pred_set` instructions that evaluate the parts of the condition and use the AND action specifier to combine all these parts in the destination register in a single cycle. GUARD instructions use a hardware AND-tree to achieve the same effect.

4.4.1 Interaction between GUARD and Branch instructions

GUARD instructions are based on the assumption that instructions in a guarded region are contiguous in memory and are fetched sequentially. Taken branches violate this assumption since they change the control flow. Therefore, any reset bits in the scalar mask bits after a taken branch would be meaningless for the instructions in the target of the branch. However, an architecture that supports GUARD instructions must define the expected behavior of overlapping branch and GUARD instructions.

To ensure simple semantics and correct execution of programs we have two options:

(a) define a convention which requires that no branches will appear within a single guarded region (and consequently they will not appear within the range of a GUARD instruction), or

(b) define that the run-time effects of a GUARD instruction will never span past the next taken branch.

The convention of the first option is sufficient to guarantee the correct execution for all correct programs, but has two disadvantages. First, it imposes strict limitations to the code that the compiler is allowed to generate. For example, it does not allow Hyperblock-type scheduling where some branches (which presumably are mostly not-taken) are retained in a guarded region. Second, the behavior of incorrect programs is not defined, which may cause unexpected behavior for incorrect programs. In environments were self-modifying code is not allowed, this convention can be enforced at compile time, by the compiler or by the assembler. However, when self-modifying code is allowed, the user can easily create (on purpose, or accidentally) incorrect programs, which in turn will exhibit unexpected behavior. To avoid these problems, the processor must detect any violations of the convention and report an exception to the user.

The second option, based on the observation that branches that are not taken do not terminate the guarded region and are therefore safe to execute, provides both simple and flexible semantics to the compiler. To ensure that the dynamic scope of the GUARD instruction does not extend beyond the next taken branch, it is sufficient to define that when a branch is taken, it sets all the bits in the scalar mask register. Assuming for a moment that the processor does not employ branch prediction, when a branch executed and is determined to be taken, the control logic has ample time to set the scalar mask register bits for the future instructions during the fetching of the target instructions. Setting the scalar mask register bits will ensure that when the target instructions are fetched and enter the pipeline, they will be executed correctly, even if GUARD instructions that preceded the branch may have reset some of the scalar mask register bits. This definition allows the compiler the flexibility to use any arbitrary region selection scheme for if-conversion.

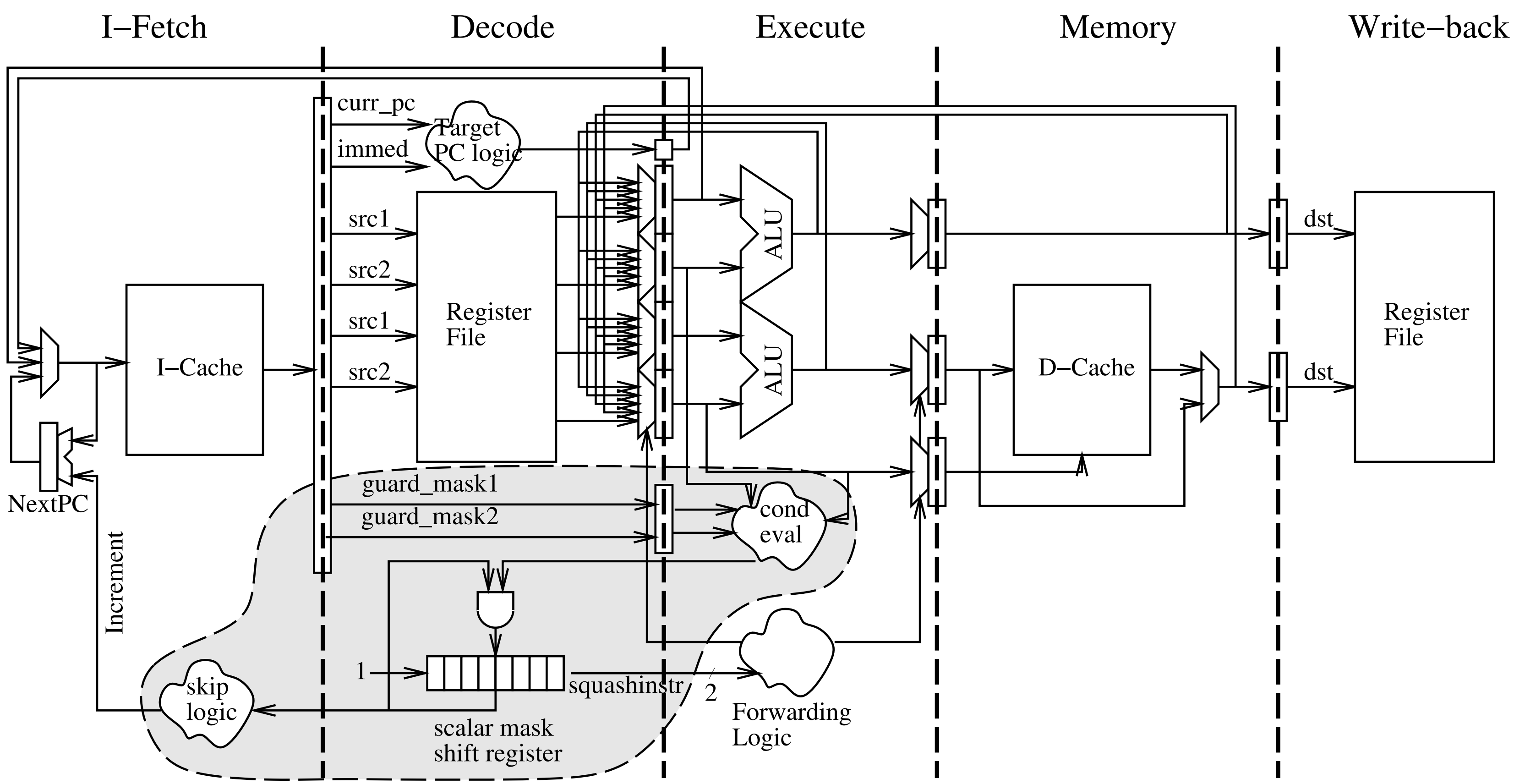


Figure 4.9: A dual issue pipeline supporting GUARD instructions and a skip capability. The shaded portion of the pipeline is the portion that needs to be added to support the execution of GUARD instructions.

If dynamic branch prediction is employed, a processor that allows branches inside the guarding regions must handle correctly the interactions between the execution of GUARD instructions and branch prediction. If a branch within the range of a GUARD instruction is predicted as taken, the scalar mask bits corresponding to instructions in the taken path should not be affected by the GUARD instruction. If the branch predicting mechanism is decoupled from the execution of GUARD instructions, the branch prediction may occur before the (logically) preceeding GUARD instructions are executed. To ensure correct execution, the taken prediction must be communicated to the GUARD execution logic, which should restrict the effects of the GUARD instructions to the instructions before the (predicted) taken branch. Since the prediction may turn out to be incorrect, the current value of the scalar mask register at the time of the execution of the branch must be kept along with the program counter of the branch instruction, so that the execution can be correctly restarted from the fall-through path of the branch.

A simpler alternative is to disallow the dynamic prediction for branches that fall within the range of GUARD instructions. By doing so, the execution unit of the processor is assured that the stream of instructions will be sequential, until either the region is terminated, or until a branch is executed and is determined to be taken. The former case poses no complications to the execution of GUARD instructions, while the later case can be handled as described earlier for the case of no branch prediction. To ensure that branches inside the guarding regions are not predicted, the processor must detect them somehow. One way to detect these branches is to use a different opcode for them. Alternatively, the processor can detect them at run-time, and store an additional bit in a pre-decoded instruction cache, which will later indicate to the branch prediction mechanism whether a branch should be predicted or not. Since the pre-decoding information may be lost due to replacement of cache lines, and to correctly handle the very first time a branch is encountered, the processor must still track the bounds of the guarding regions and the predictions made by the branch prediction unit. When the processor detects a taken branch prediction inside the bounds of a guarded region, it can wait to resolve the branch (thus ensuring the correct execution without any additional hardware support), and update the pre-decode information for the branch, marking it so it will not be predicted. Table 4.2 summarizes the interactions, advantages and disadvantages of the branch handling options.

4.4.2 Interrupt handling and GUARD instructions

Since the scalar mask is part of the processor state, it must be saved and restored on interrupts and context switches. The saved scalar mask, together with the saved program counter value and the architectural register file(s), provide sufficient information to restart an interrupted process correctly.

A fairly straightforward solution is to expose the scalar mask register to the user and system software as part of the processor state, and introduce user-level instructions to save and restore the scalar mask. To ensure that the value of the scalar mask at the time of the interrupt is saved by the save instruction, the processor must stop shifting the scalar mask as soon as the interrupt is accepted. As a consequence, the interrupt handler code cannot use GUARD instructions until the scalar mask is saved. The shifting of the scalar mask register can be enabled by the instruction that restores a value in it.

Saving and restoring the scalar mask value can be easily achieved if the architecture defines an *Exception Scalar Mask* (ESM) register, similar to the *Exception Program Counter* of the MIPS architecture [Kan87]. When an interrupt or an exception is taken by the processor, the value of the scalar mask register is copied into the ESM register, and all the scalar mask bits are set to 1. The exception handler can proceed using GUARD instructions since the old scalar mask value is safe in the ESM reg-

	Branches inside guarding regions	
	Do not allow	Allow
Advantages	Ordinary treatment of branches	Flexibly region selection
Disadvantages	Restricts region selection	Special handling for branches
Not-taken branch handling	Ordinary	Ordinary
Taken branch handling	Ordinary	Set the remaining bit in scalar mask register
Predict branches inside guarding regions	N/A	Must save guard mask register value for recovery
Do not predict branches inside guarding regions	N/A	Must detect which branches are inside guarding regions (use different opcodes, or detect at run-time)

Table 4.2: Interactions between guarding regions and branches.

ister. To allow nested exception and interrupts, the exception handler must ensure that the ESM register is stored in memory before the next interrupt is accepted. The ESM register will be automatically restored into the scalar mask register when a *Return From Exception* instruction is executed. In this way, only the ESM register needs to be directly exposed to the user and system software, and no restrictions are placed on the use of GUARD instructions in the exception handlers.

An alternative to saving and restoring the scalar mask register is to require that interrupts will be accepted only on PC values that correspond to a "clean" state (i.e., to a scalar mask with all the bits set). In this case, the PC value and the register file(s) contents are sufficient to fully describe the state of the processor and to restart the process. In this approach, the handling of traps (which cannot be deferred until the state of the processor becomes clean) requires that the processor must be able to revert to the last PC for which the state was clean, in a manner similar to the checkpoint repair of [HP87]. Since the control of GUARD instructions is in the hands of the compiler or the user, it will be hard or even impossible to predict how soon the processor state will become clean. For systems that want to ensure timely service of interrupts, deferring interrupt handling until the processor is in a clean state will be unacceptable. Also, since the guarding regions can be arbitrarily large, the state that the processor is required to buffer so that it can revert to a clean state can also be arbitrarily large.

Table 4.3 summarizes the interrupt handling options. Among the options, using an Exception Scalar Mask register that snapshots the value of the scalar mask on an interrupt or an exception is both simpler and more flexible, and should be the option of choice.

4.5 Implications of out-of-order execution

When GUARD instructions are used in an out-of-order processor, the guard condition may not be available at the time the GUARD instruction is issued. The GUARD instructions should be buffered in the Reorder Update Unit (RUU) [SV87, Soh90] or the equivalent structure, so that they will be executed

Interrupt Handling and GUARD Instructions		
Option	Mechanism	Comments
Expose scalar mask register to the user	Need a mechanism to freeze and resume the shifting of the scalar mask register and instructions to save/restore its value	Flexible and guarantees forward progress
Introduce an Exception Scalar Mask (ESM) register	Scalar mask automatically saved into ESM on exceptions and restored by RFE instruction	Simple semantics, flexible and guarantees forward progress
Hide scalar mask register	On interrupts, H/W must roll-back to clean state, or postpone the interrupt (if possible) until the state becomes clean	Forward progress is hard to guarantee with roll-back, and fairness is hard to guarantee if the interrupts can be postponed

Table 4.3: Interrupt handling options with GUARD instructions.

when its operand becomes available. This out-of-order execution of GUARD instructions makes the implementation of GUARD instructions in an out-of-order processor more complicated than that in an in-order processor. In this section we address two problems: (i) how to determine when a particular bit of the scalar mask is ready and (ii) how can we handle taken branches and branch prediction.

4.5.1 Determining when a scalar mask bit is ready

In order to execute an instruction, an out-of-order processor supporting GUARD instructions has to consult the corresponding scalar mask register bit. However, this bit will not be available if some GUARD instruction that has been issued in the RUU but not been yet executed lists this bit in its guard mask. In fact, multiple GUARD instructions may list the same scalar mask bit in their guard masks, and the processor needs a mechanism to determine when all these instructions have been issued, to determine whether a scalar mask bit is available or busy.

To determine the set of busy scalar mask bits, the processor can OR all the guard masks of the outstanding GUARD instructions (appropriately shifted) and subtract any scalar mask bits that are already zero (a zero in a scalar mask bit means that the instruction is guaranteed to be squashed regardless of any outstanding GUARD instruction). When a scalar mask bit is busy, the corresponding instruction in the RUU the instruction should not be allowed to execute; the instruction must be held until all the outstanding GUARD instructions which have this bit set in their guard masks are executed, or until the scalar mask bit becomes 0 (in which case the instruction can be immediately squashed).

Guard Mask Buffer

The process just described can be implemented directly in hardware. The processor can keep a copy of the guard masks of all the outstanding GUARD instructions, in a small buffer, called the *Guard Mask Buffer*. An entry in this buffer is allocated for every GUARD instruction that is issued before its operand is available. Entries in this buffer are freed (and zeroed) as soon as a GUARD instruction is executed.

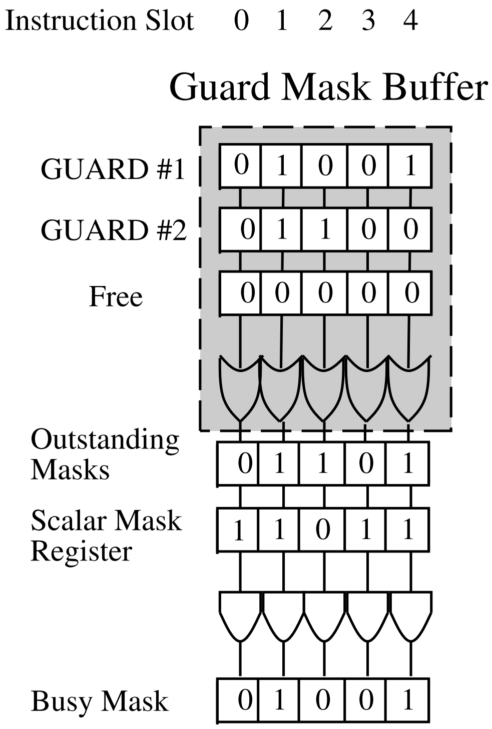


Figure 4.10: Support for out-of-order execution of GUARD instructions: Guard Mask Buffer.

To determine the set of busy scalar mask bits. all the entries of the Guard Mask Buffer are OR-ed and the result is AND-ed with the scalar mask. Figure 4.10 illustrates the structure and operation of a Guard Mask Buffer with 3 entries. Two of the entries contain the mask of outstanding GUARD instructions while the third one is free. The masks are OR-ed to compute the "Outstanding Masks", which is in turn AND-ed with the scalar mask register to get the final "Busy Mask". Notice, that despite the fact that there is an outstanding GUARD instruction guarding instruction slot #2, the corresponding bit in the scalar mask register is zero, and it is *not* marked as busy.

Figure 4.11 shows how the Guard Mask Buffer would be connected to the RUU of the processor. The Guard Mask Buffer is shown to be as wide as the number of RUU entries. In this layout, the mapping between scalar mask bits and RUU entries is fixed. Instead of shifting the scalar mask register, we have to rotate the guard masks of all the GUARD instructions to refer to the appropriate RUU entry. As described earlier, the Guard Mask Buffer produces a busy bit for each of the scalar mask register bits. The scalar mask bit is stored in the *SMi* field while the busy bit is inverted and stored into the *Vsm* field (which stands for Valid Scalar Mask). The instruction is considered ready when the scalar mask bit is valid and either it is set and all the source operands are available, or it is false and the old value of the destination register is available.

A limitation of this method is that it requires that the RUU is managed as a strict circular queue, so that there is a trivial one-to-one mapping between bits in the scalar mask register and RUU entries. While the RUU was designed with this management in mind, other alternatives (for example splitting the RUU into a possibly centralizes set of Reservation Stations coupled with a reorder buffer) can reuse the entries as soon as they are freed, resulting in a more efficient utilization of the reservation station

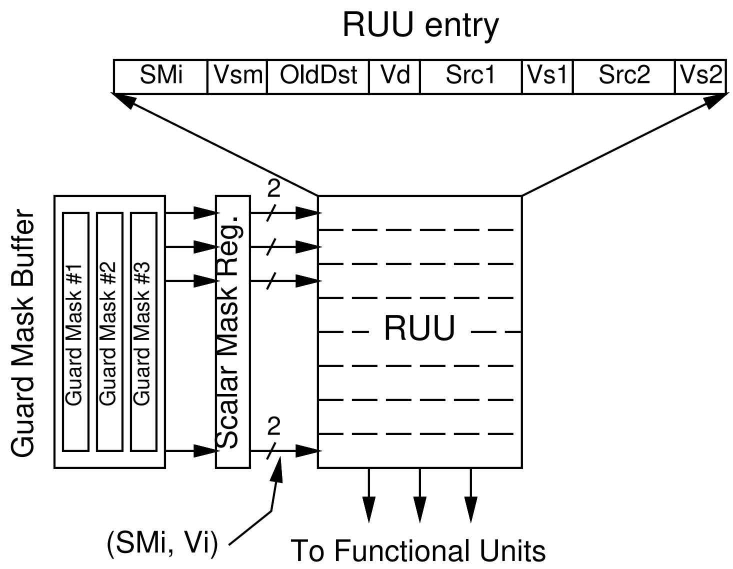


Figure 4.11: Guard Mask Buffer and RUU. The scalar mask register is distributed among all the RUU entries. For each of the RUU entries, the Guard Mask Buffer provides two bits, the scalar mask bit *SMi* and its corresponding valid bit (the inverted busy bit) *Vsm*. This diagram does not show the logic inside the Scalar Mask Buffer, nor the logic to generate the busy bits from the outstanding mask and the scalar mask register.

entries.

Reference Counts

An alternative way to keep track of the outstanding scalar mask bits, is to associate a *Reference Count* with each of the bits of the scalar mask. When a GUARD instruction is issued before its operand is available, it will increment the reference counts for each of the bits in its guard mask. When a GUARD instruction is executed, it will decrement the reference counts for each of the bits in its guard mask. Therefore, for each scalar mask bit, the reference count determines how many GUARD instructions have this bit set in their guard masks and are outstanding in the RUU. If the count is zero, the corresponding scalar mask bit is available. Figure 4.12 repeats the previous example using reference counts.

The implementation of either a Guard Mask Buffer or of a Reference Count scheme must set a limit on the maximum number of outstanding GUARD instructions in the processor. This number will either be the number of entries in the Guard Mask Buffer, or the maximum value of the reference counters. When this maximum number is exceeded, the processor must cease issuing instructions, either until one of the outstanding GUARD instructions complete, or until the operand of the current GUARD instruction becomes available.

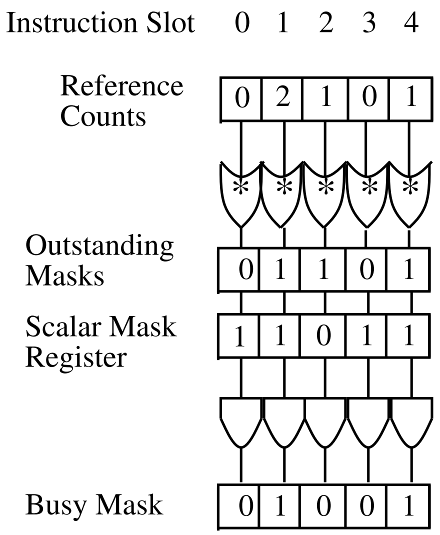


Figure 4.12: Support for out-of-order execution of GUARD instructions: Reference Counts.

4.5.2 Interaction of GUARD instructions and Branch prediction

The handling of branches inside guarding regions is also made more complicated by the asynchronous nature of an out-of-order processor. As we described earlier, when a branch is taken, it should set all the bits of the scalar mask register that correspond to the target instructions, to ensure that these instructions are not affected by a GUARD that precedes the taken branch. In an in-order processor, all preceeding GUARD instructions would have already been executed by that time, and this action is enough to guarantee that preceeding GUARD instructions cannot affect the instructions from the taken path. However, in an out-of-order processor there is no guarantee that the preceeding GUARD instructions will have been already executed. When these GUARD instructions are executed, they may incorrectly squash the wrong instructions (as described in Section 4.4.1). Therefore, the processor needs a mechanism that will allow the out-of-order execution of GUARD and branch instructions, while at the same time ensure that the "reach" of a GUARD instruction (that is the range of instructions that the GUARD instruction can affect) will never exceed the next taken branch.

Our solution to this problem works by renaming the scalar mask register whenever a branch is taken and by combining the correct parts of the renamed scalar mask registers to obtain the actual scalar mask register value To achieve both renaming and combining the renamed scalar masks we use a *Scalar Mask Rename Buffer* (SMRB). Each entry in the SMRB is a pair of a *Renamed Scalar Mask* and a *Reach Mask*. As the name implies, the Renamed Scalar Mask is a renamed version of the architectural scalar mask register. The Reach Mask is used to restrict the effects of the Renamed Scalar Mask to the appropriate instructions (i.e., up to the next taken branch). Our convention is that a reset bit in the Reach Mask means that the execution of the corresponding instruction is determined by the Renamed Scalar Mask, and a set bit means that the instruction in outside the reach of this Renamed Scalar Mask. The Reach Masks are set so that exactly one Renamed Scalar Mask refers refer to a particular RUU entry at any time (the exact procedure that we use to manage the Reach Masks will be described shortly). To determine the actual value of the scalar mask, we use a two step process. First, we OR each Reach Mask with its Renamed Scalar Mask. This operation computes a list of partial squash masks. Since all

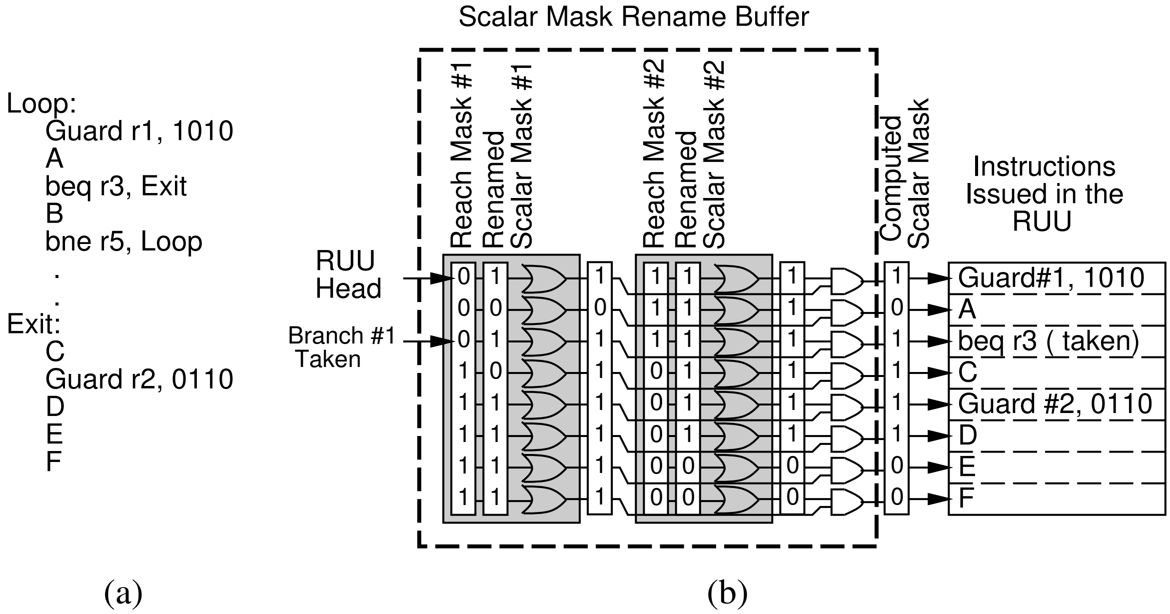


Figure 4.13: The Scalar Mask Rename Buffer. Part (a) shows a simple code sequence of a guarded loop containing an exit branch. Part (b) shows the contents of a two-entry Scalar Mask Rename Buffer assuming that the conditions of both GUARD instructions evaluate to false and that the exit branch is taken.

the Reach Masks are mutually exclusive, the second step combines the squash masks by simply AND-ing the partial squash masks to obtain the overall squash mask. This squash mask corresponds to the value that the architectural scalar mask register would have at this point.

Figure 4.13(a) shows a code sequence that illustrates the operation of the SMRB. It consists of a guarded loop body that contains an conditional exit in the middle. The code after the Exit label is itself guarded. In Figure 4.13(a) we assume that both guard conditions evaluate to false and that the first branch is taken. Under these assumptions, the extent of the first GUARD instruction must be restricted so it does not affect instructions which come after the taken branch. Figure 4.13(b) shows the state of a two entry SMRB after all the instructions have been issued in the RUU and the two GUARD instructions are executed. Renamed Scalar Mask #1 contains two zero entries, and the second one corresponds to instruction B which is statically but not dynamically after the branch. This zero is masked by Reach Mask #1. None of the zero bits in the Renamed Scalar Mask #2 are masked since they do not span across a taken branch instruction.

At this point we should note that the SMRB contains all the information required to allow the use of dynamic branch prediction for branches within guarded regions. The Renamed Scalar Masks can accumulate the entire contents of all the GUARD instructions that precede the branch that caused the renaming. Therefore, if we rename the scalar mask register whenever such a branch is *predicted* to be taken, we can allow the prediction of branches inside guarded regions. The renaming and the Reach Mask will ensure that if the prediction is correct the execution of the instructions from the predicted path will proceed correctly, while the older Renamed Scalar Masks can be used to recover the correct scalar mask value is the prediction was incorrect. Note, that the scalar mask register need not be renamed when a branch outside a guarding region is predicted as taken (for example, a taken prediction for the loop-closing branch in Figure 4.13(a) need not rename the scalar mask register). The processor can detect the branches are within a guarded region by keeping track of the most distant bit that was set in any of the GUARD instructions that have been already issued and by checking whether the predicted taken branch precedes or succeeds that instruction. If the processor is able to track this information, it will rename the scalar mask register more infrequently, reducing the number of the required SMRB entries.

Management of the Scalar Mask Rename Buffer

As the processor issues and commits instructions to and from the RUU, it has to perform some book-keeping to maintain the Reach Masks. At any point, one entry of the SMRB is designated as the "current" entry. When a GUARD instruction is issued, we assign the current Renamed Scalar Mask as its destination. Note, that since we rename only on (predicted) taken branches, several GUARD instructions may write to the same Renamed Scalar Mask register; these instructions are not serialized since the AND operation that they perform on the scalar mask is associative.

When an instruction is removed from the RUU (because it was committed or because it was transfered to a Reorder Buffer), the corresponding bit of the Reach Mask of all the older Renamed Scalar Masks is set to 1, indicating that these older Renamed Scalar Masks do not refer to the future instruction that will be assigned to this newly freed entry of the RUU. The corresponding bit of the current Reach Mask is set to 0, to indicate that the current scalar mask will determine the execution of the instruction that will be assigned there.

When a branch is (predicted) taken, two actions must be taken: (i) the reach of the current scalar register should be set so that it does not refer to instructions after the taken branch (i.e., the bits corresponding to instructions after the branch should be set to 1), and (ii) the scalar mask register should be renamed by allocating a new Reach Mask/Scalar Mask pair from the SMRB. If a free entry is not available, the issuing of instructions must stop, until an entry is freed. The Reach Mask of the new entry contains zeros for all the entries that are already allocated in the RUU and ones for all the remaining bits. An SMRB entry can be freed as soon as all the instructions within the reach of the Renamed Scalar Mask have been removed from the RUU (that is, either they have been committed, or they have been transfered to a Reorder Buffer or a similar structure). This condition can be detected by comparing the RUU head pointer with the location of the branch that caused the renaming. It can also be detected by checking whether all the Reach Mask bits are set to one, indicating that all the instructions within its reach have been removed from the RUU.

The number of bits in the Reach Mask and Renamed Scalar Masks is the maximum of the scalar mask register width so that the renamed scalar mask can accumulate the result of all GUARD instructions, and the number of RUU entries, so that each of the entries has storage for its corresponding scalar mask bit. A nice feature of the SMRB is that even though a GUARD instruction may refer to instructions not yet placed in the RUU, the effects of the GUARD instruction can be stored in the renamed scalar mask. These results will be masked by the Reach Mask until the corresponding (future) instructions are issued and placed in the appropriate entry of the RUU.

4.6 Summary

GUARD instructions offer an easy and powerful way to accommodate guarded execution in an instruction set. With the modest requirements of just a few opcodes, GUARD instructions are sufficient to provide efficient support for full guarding. Furthermore, with the addition of a scalar mask register in the architecture, GUARD instructions provide the processor with advance information about the instructions that will be squashed later in the execution of a program. The processor can take advantage of this early warning and skip over the squashed computation and allocate the resources to instructions that contribute to the programs computation. GUARD instructions can also reduce the number of instructions required to evaluate the guard conditions for an if-converted region of code. Therefore, GUARD instructions are ideal candidates for inclusion in an instruction set that supports guarding, especially for existing instruction sets, for which a complete rehaul of the architecture to support guarding in the traditional way is very hard or impossible. In the next two chapters we quantify the potential of guarded execution and we compare the performance of guarded execution using a guard operand per instruction with guarded execution using GUARD instructions.

Chapter 5
Effects of guarding on the dynamic program characteristics

The previous chapters discussed the use and benefits of guarding, the compilation techniques and one way to add guarding in instruction sets. In this and in the next chapter we investigate the performance potential of guarding. This chapter describes our methodology and concentrates on the impact of guarding on the instruction mix and dynamic program characteristics. The next chapter will address the ultimate performance metric: the execution time of a program.

5.1 Motivation and metrics

Guarding affects the dynamic program characteristics in many ways. The goal of this chapter is to get a qualitative, as well as a quantitative feel for each of the ways a program is affected, and be able to compare and draw conclusions about the effectiveness of guarding for each program. This effectiveness is largely determined by the extent of instruction set support for guarding. In this chapter we consider three cases: full guarding using the ordinary encoding of guard operands, restricted guarding using only conditional moves, and full guarding using GUARD instructions. The metrics we use to measure the effects of guarding are: overhead in the number of instructions fetched and executed, branch counts, dynamic branch behavior and guarding region characteristics. Next we define these metrics and explain how they are affected by guarding.

5.1.1 Guarding overhead

The first metric we use is the dynamic *guarding overhead*, that is, the amount of additional instructions that the processor has to fetch and (possibly) execute after a program has been if-converted. As described in Chapter 2, guarding reduces the number of executed branches, but introduces instructions to initialize and set the guard registers, and executes the guarded instructions even when they will be squashed by their guard conditions. The net result is that the guarded version of a program may execute more instructions than the original version. The guarding overhead measures the increase in the processor resources that are consumed during the execution of the programs. Note that in this chapter we do not distinguish between instructions fetched, decoded and executed. However, as described in Chapter 4, Section 4.1.2, using GUARD instructions the processor may be able to reduce the number of guarded instructions that must be fetched and decoded; we will address this issue in the next chapter.

The guarding overhead can be further subdivided into squashed computations, and instructions required to evaluate guard conditions. The number of squashed instructions indicates how successful the region selection algorithm is in picking the important nodes in the CFG, and including them in the guarding regions. If the region selection process is overly aggressive, it will include too much useless computation, which will be squashed dynamically. The number of condition evaluation instructions is also important to understand how the overhead of guarding is distributed between actual computation that has been guarded and support instructions that are introduced by the if-conversion process.

5.1.2 Branch counts

The effectiveness of guarding can also be measured by how much it reduces the number of branches in a program. The elimination of branches may improve the performance of a program since the processor experience fewer changes in the control flow. Also, the number of eliminated branches will determine the size of the if-converted regions. In general, we want the guarding regions to be as large as possible, to expose more opportunities for optimizations to the compiler, and consequently to the processor, but without introducing too much overhead computation. Although if-conversion primarily targets conditional branches, it also eliminates some unconditional branches (for example, an if-then-else structure contains one conditional and one unconditional branch, while the if-converted code would not contain any branches). However, if-conversion, by itself, does not affect the number of function calls and returns, or the number of indirect and loop-closing branches. Therefore, the counts of these types of branches would remain unaffected by the use of guarding.

5.1.3 Dynamic branch behavior

The dynamic behavior of the branches in a program is also very important. When a processor employs a dynamic branch predictor, the effects of guarding on the prediction accuracy is important. Since guarding eliminates some of the conditional and unconditional branches, it will affect the performance of any dynamic branch prediction scheme such as a BTB or a dynamic branch predictor. For a BTB, a smaller number of branches means a lower probability of conflict or capacity misses in the buffer. For a dynamic branch predictor, guarding may eliminate branches that are not very predictable, the prediction accuracy will increase. If however, the eliminated branches were well behaved (i.e., were almost always predicted correctly), the prediction accuracy might actually decrease. In this chapter we report the combined *prediction accuracy* of the branch predictor and the BTB, since a correct prediction would be meaningless without a predicted target to fetch from.

A better metric to measure the impact of guarding on a dynamic predictor is the total *number of mispredictions* experienced by the original and the guarded versions of the program. The number of mispredictions determines the number of cycles that are wasted while the processor fetches the instructions from the correct target path. An alternative way of expressing the impact of mispredictions is to use the dynamic *misprediction distance*, that is the number of instructions between mispredictions. The misprediction distance indicates how often the processor has to interrupt the instruction supply due to an incorrect prediction.

One way to measure the effectiveness of if-conversion in eliminating small basic blocks and reducing the frequency of taken branches, is to measure the *run-length*, that is, the number of instructions between taken branches. This metric is important because the instruction fetch mechanism of a processor can utilize the high-bandwidth on-chip instruction cache to fetch, decode and issue (if the dependencies permit) the sequential instructions.

To summarize, our metrics for the dynamic branch behavior are: (i) branch prediction, (ii) number of mispredictions, (iii) misprediction distance and (iv) run-length.

5.1.4 Guarding region characteristics

To determine the characteristics of our guarding regions we used three metrics: (i) the number of guard conditions per region, (ii) the guard condition lifetimes, i.e., the distance between the first definition and the last use of the condition, and (iii) the number of instructions guarded per conditions. The number of the guard conditions and the guard condition lifetimes are an indication of the register pressure created by the if-conversion. Each of the guard conditions has to live in a guard register throughout its lifetime (spilling the condition into memory is always possible, but introduces additional overhead). Therefore, a large number of conditions with long lifetimes will consume more registers; if the register pressure is very high, a separate guard register file may be a necessary addition to the architecture. The number of instructions guarded per condition measures how often the condition is used during its lifetime and indicates how many instructions share the cost of evaluating the guard condition.

The three aforementioned metrics are also key in determining the effectiveness of GUARD instructions. The number of guard conditions in a guarded region together with the guarding distance determine the number of GUARD instructions that are required to specify the guard state of the computation. Each distinct guard condition requires a separate GUARD instruction, which the guarding distance determines whether the guard mask of a single GUARD instruction is enough to specify the guarding state of all the instructions that are guarded by the specified condition. The number of instructions that are guarded by the same guard condition measures the compactness of GUARD instructions. If each GUARD instruction guards just a few instructions, the overhead will be large; if each GUARD specifies the guard condition for several instructions, the specification overhead will be relatively small.

5.2 Related work

Several studies have addressed the effects of guarding on the instruction mix and the dynamic branch behavior of programs [PS94, MHB$^+$94, MHM$^+$95, CHPC95, Tys94b]. The first study to consider these effects was done by Pnevmatikatos and Sohi [PS94]. They considered full and restricted guarding (where only ALU instructions were available in a guarded form) and, using binaries compiled for a MIPS R2000 processor, found that full guarding could eliminate over 30% of the branches and about the same percentage of mispredictions for the cost of about 33% overhead computation. They also proposed and evaluated the concept of GUARD instructions, reporting an preliminary overhead of about 8%.

Mahlke *et al.* [MHB$^+$94] evaluated the effect of guarding on the branch characteristics of a program using the IMPACT compiler. The IMPACT compiler uses sophisticated code transformations guided by profiling information and heuristics to form the Hyperblocks described in Chapter 3. They found that the number of branches was reduced by 27%, with a corresponding 30% reduction in the number of mispredictions. In a later paper [MHM$^+$95], Mahlke *et al.* compared the performance of full guarding to the performance of guarding using conditional move instructions, and found that conditional moves achieved about 50% of the performance of full guarding. However, Mahlke *et al.* use a processor model with an infinite number of registers. An infinite number of registers allows the compiler to aggressively use optimizations that consume registers, such as unrolling, predicate promotions etc, which help achieve greater levels of performance for the fully predicated code, and can hide some of the overhead of synthesizing guarded instructions using conditional moves.

Chang *et al.* [CHPC95] used branch prediction profiling and hand-coding to if-convert branches that cause many mispredictions. They found that this profile-based if-conversion can eliminate more than 60% of the mispredictions for the programs Compress, Eqntott and Sc and estimated that it could eliminate 40% of the misprediction for Gcc.

Tyson [Tys94b] considered a form of predication where instructions covered by a short forward branch are transformed into guarded instructions. This transformation can be performed either statically by the compiler/assembler, or dynamically by the processor. Tyson studied the effectiveness of guarding in removing the short forward branches; he also studied the effects of guarding on the prediction accuracies of several predictors. Tyson found that guarding could eliminate up to 30% of the branches of a program and up to the same amount of the mispredictions for some of the branch predictors.

5.3 Methodology

For the purposes of our evaluation, we use a MIPS-like instruction set architecture. Our ISA, called *SimpleScalar*, is similar to the MIPS R2000 ISA with the following modifications: (i) we removed all the architectural delay slots, (ii) we augmented the instruction set by adding guarded versions for all instructions and (iii) we added the set of required GUARD instructions, namely GUARDTRUE, GUARDFALSE, GUARDBOTH and the GUARDUPPER versions of these instructions.

5.3.1 Compilation techniques

To compile our programs we use the GNU *Gcc* compiler (version 2.6.0) configured to generate SimpleScalar assembly output. The generated assembly is then scheduled by *Titan*, our instruction scheduler, which performs if-conversion and code scheduling. The scheduled assembly files are assembled into an object using the GNU assembler *gas* (version 2.2), modified to accept the modified SimpleScalar assembly. The object files are linked into an executable using GNU *gld* (version 2.3). The standard C libraries were recompiled from the sources that accompany the MIPS compiler version 3.01.

Region Selection

Titan uses no profiling information to select the range of guarding, as we feel that profiling is not an option in many environments, especially in casual, every-day use of a compiler. Instead, we use the structural properties of the program to determine the guarding regions of the control flow graph. Our region selection criteria are similar to the ones used by the Hyperblock [MLC$^+$92]. The regions are required to have a single entry point, and must not contain nested loops. In addition we require that a function call must terminate the guarding region. This restriction simplifies the register allocation in guarded regions since the guard registers can be allocated with no cost in any unused callee-save registers; if the regions spanned across function calls, the guard registers would have to be saved and restored on each function call. Our region selection algorithm allows multiple exit branches inside an if-converted region. This flexibility is very important, since common programming structures and practices generate code in which certain segments are infrequently executed. For example, such segments handle special input cases, error conditions etc. Other common structures that can generate regions with multiple exit arcs are loops that contain the C `break`, `continue` or `return` statements. If we disallow branches inside guarding regions, the bodies of such loops cannot be if-converted. Titan uses a limited form of the control-tree height reduction technique (described in Chapter 3). When all the operands of a complex control equation are already available in registers (i.e., when the operand evaluation does not require any computation other than comparisons), the control structure is transformed into a tree.

If-conversion for full guarding

For full guarding and conditional moves, once the if-conversion regions are identified, we use the RK algorithm (described in Section 3.2) for if-conversion. The output of the RK algorithm is passed through a peep-hole optimization step that detects and eliminates multiple complementary conditions (these conditions are generated for the mutually exclusive paths of if-then-else statements). This modification reduces the total number of guard registers required for the if-conversion of a region, reducing at the same time the number of set instructions required to evaluate the guard conditions. The if-converted code is passed thought a register re-allocation step; this step first identifies the set of unused and dead registers, and then statically re-assigns temporary registers into these free registers, in order to eliminate false dependencies and increase the parallelism in the guarded regions.

When we schedule the code for full guarding, we consider both an architecture with a single, unified guard and general purpose register file and one with a separate guard register file. To schedule the code for an architecture with a unified register file for both guard registers and general purpose registers, we perform a register allocation step to allocate the guard registers in unused general purpose register. This allocation competes with the registers that hold computation results and may introduce register spills and reloads to free enough general purpose registers. These register spills and reloads are scheduled along with the original program computation; the overhead of these spills and reloads will be presented in Table 5.9. When a separate guard register file is available, guard conditions are simply allocated in the guard register file, greatly simplifying the determination of regions to be guarded.

If-conversion for conditional moves

When we use conditional moves, the region selection process gives preference to basic blocks containing only safe instructions (instructions that cannot cause exceptions such as integer and logic computations, and loads from safe locations such as the stack frame or the global variable region of memory). When a basic block contains unsafe instructions, the scheduler checks whether the inclusion of this basic block will actually increase the schedule length. If so, the basic block is not included in the guarding region. Because of these stricter restrictions, the selected guarding regions will be smaller for conditional moves, than for full guarding.

After the guarding regions are identified, we use the RK algorithm for if-conversion and the peep-hole optimizations to assign the appropriate conditions. Titan expands the fully guarded output of the if-conversion process using conditional moves according to the instruction sequences in Table 2.3. Scheduling for conditional moves assumes a single, unified register file. All program variables, guard conditions, and temporaries for the synthesis of guarded computation are allocated in this unified general purpose register file.

If-conversion for GUARD instructions

Since GUARD instructions can specify full guarding, we use the same guarding regions as with full guarding. The only difference between the two codes is that instead of the usual initialization and set instructions, we use the GUARD instructions, as generated by the GUARD assignment algorithm described in Section 4.3.1. The if-converted code is then passed through the register re-allocation step to increase the parallelism of the guarded regions.

5.3.2 Simulation techniques

To measure the effects of guarding on our metrics we use an execution driven functional simulator. The compilation process generates four executables for each program. The first one is the original, scheduled but otherwise unmodified executable to be used as the base for all comparisons. The three remaining executables are one for each type of instruction set support for guarding: full, conditional moves and GUARD instructions.

Our simulated processor architecture includes a separate guard register file with 32 registers. To simulate the effects of a unified register file, the executable for full guarding is tagged in every location where a spill or a reload would have been scheduled; when simulating for a unified register file, the simulator injects on-the-fly the appropriate load and store instructions.

To evaluate the effects of guarding on the behavior of dynamic branch predictors we simulate a branch target buffer (BTB) along with a dynamic branch predictor to record the history of branches. The BTB is direct mapped and contains 1024 entries that record the last taken target of each branch, including the latest target of indirect branches. For the dynamic branch predictor we use two configurations. The first is a straightforward counter based predictor, using 4096 (4K) 2-bit counters to record the history of the branches. The second is a 2-level adaptive predictor that uses an 8-bit pattern history register in the first level and 4K 2-bit counters in the second level; this configuration corresponds to the *Correlation Branch Predictor* of Pan *et al.* [PSR92] or GAp(8) according to the Yeh's and Patt's classification scheme [YP93].

5.3.3 Benchmark programs

For evaluation purposes we use 8 benchmark programs. These are the 6 integer SPEC92 benchmark programs [SPE91], namely *Compress*, *Eqntott*, *Espresso*, *Gcc*, *Sc* and *Xlisp*, along with *Elvis* and *Wc*. All these programs are integer intensive; we did not use any floating point intensive programs as we feel that the effectiveness of guarded execution for scientific programs has long being established for vector and VLIW architectures [RYYT89, DHB89, HT72, Wat72, Rus78].

For the SPEC92 benchmark programs we used the SPEC reference inputs, with the exception of *Xlisp*, for which we used a smaller input (we solved the seven queens problem instead of the nine queens problem specified by the SPEC92 reference input file). For *Wc* the input is file *cccp.c* supplied by the Illinois IMPACT group. *Elvis* is a vi-compatible full-screen editor performing two regular expression search-and-substitute operations (in a batch-mode) for the input file *unix.c*. The same benchmark and input was also used by Austin *et al.* [APS95]. Table 5.1 lists all our benchmark programs with their command line arguments.

5.4 Effects of full guarding

In this section we examine the effects of full guarding on the instruction mix and dynamic behavior of our benchmark programs. The metrics of interest are the amount of overhead computation (either computation required to evaluate conditions or squashed computation), and the reduction of the number of branches in the programs. We also evaluate the guard condition characteristics of our guarded regions and the impact of guarding on the dynamic branch behavior of the program.

Program	Command Line	Comments
Compress	in-ref	reference SPEC92 input
Elvis	-b CMDS unix.c	input used in [APS95]
Eqntott	-s -.ioplte int_pri_3.eqn	reference SPEC92 input
Espresso	-t cps.in	one of the SPEC92 reference inputs
Gcc (cc1)	-O integrate.i	one of the SPEC92 reference inputs
Sc	test.start < loada1	one of the SPEC92 reference inputs
Xlisp	queens7.lsp	SPEC92 reference solves 9 queens
Wc	cccp.c	input file used by the IMPACT Group

Table 5.1: Benchmark programs and their command lines and inputs.

5.4.1 Guarding overhead

We begin the evaluation by addressing the impact of guarding on the instruction mix of our benchmark programs. Table 5.2 shows the instruction counts for the original and the guarded versions of our benchmark programs and the guarding overhead as a percentage of the original instruction count. In Table 5.2 we see that there is a consistent overhead involved in guarded execution. The smallest overhead is 15% for *Elvis* while the largest is 37% for *Gcc*. The average guarding overhead (all averages in this chapter are calculated assuming that all programs will run for the same number of instructions) is about 19%.

Program	Total Instructions		Guarding
	Original	Guarded	Overhead (%)
Compress	81,972	94,678	15.50
Elvis	18,482	21,257	15.02
Eqntott	1,164,352	1,346,355	15.63
Espresso	522,085	651,770	24.84
Gcc	71,803	98,371	37.00
Sc	402,133	481,388	19.71
Wc	2,176	2,633	20.99
Xlisp	228,708	276,200	20.77
Total/Average	2,491,711	2,972,652	19.23

Table 5.2: Instruction counts for the original and guarded benchmarks (in thousands) and overhead (percentage) of guarding.

The guarding overhead can be attributed to: (i) initialization and sets of guard registers and (ii) guarded computation that is dynamically squashed and therefore does not contribute to the useful program computation. Table 5.3 quantifies the overhead according to these categories. In Table 5.3, *Inits* and *Sets* are the counts of instructions needed to initialize (clear) and set the guard registers as generated by the RK algorithm for if-conversion, and *Rest* include all the remaining computation (including both useful and squashed computation). The *Squashed* column lists the number of guarded instructions that are dynamically squashed, and the *Squashed %* column shows the same number as a percentage of the total instruction count of the program. In Table 5.3 we can see that the number of initializations and sets of guard registers represent a significant fraction of the overall instruction count. The amount of squashed computation, however, is not very large, with a maximum of 18.6% for *Wc* and an average of about 8.5%, indicating that our region selection algorithm is not overly aggressive, and that non-useful computation will not overwhelm the processor resources.

Program	Inits	Sets	Rest	Squashed	Squashed (%)
Compress	4,003	8,052	82,621	5,734	6.06
Elvis	991	2,002	18,263	1,454	6.84
Eqntott	15,484	259,673	1,071,196	81,010	6.02
Espresso	28,969	45,575	577,224	84,306	12.93
Gcc	6,503	10,062	81,805	16,176	16.44
Sc	19,404	35,201	426,782	51,327	10.66
Wc	105	210	2,317	489	18.60
Xlisp	12,468	23,815	239,916	22,168	8.03
Total/Average	87,927	384,590	2,500,124	262,664	8.56

Table 5.3: Instruction mix for the guarded benchmark programs.

5.4.2 Branch counts

Next, we turn our attention to the effects of guarding on the branch statistics of our benchmark programs. Table 5.4 presents the number of calls, returns, indirect and loop-closing branches for all the benchmarks. It also lists the average number of instructions between branches. Figure 5.1 presents the same numbers, as well as the number of remaining conditional and unconditional branches as a percentage of all the branches in each program. Observing Table 5.4 and Figure 5.1 it is easy to get a qualitative feel of the nature of these benchmarks. Most benchmarks are loop dominated, with *Eqntott* being the most prominent in this category. However, *Sc* and *Xlisp* perform a large fraction of their computation using function calls, suggesting that if-conversion may not be as effective for these two as for the rest of the programs.

Table 5.5 presents the number of conditional and unconditional branches for both the original and the guarded version of the programs; it also shows the percentage of the reduction in the branch count for conditional and unconditional branches. In Table 5.5 we see that the reduction in the number of conditional branches ranges widely, from an impressive high of 99.9%, to a low of about 13%. with a corresponding range of 3% up to 99.9% for unconditional branches. The best case occurs for *Wc*, which has a simple control structure: a single loop with several conditional and unconditional branches dominates the entire program. If-conversion was able to eliminate all the branches inside the loop body, eliminating almost all the conditional and unconditional branches in the program (excluding, of course, the loop closing branches). In contrast, if-conversion had a much smaller effect on the branch counts of *Sc*. The reason is that *Sc* has a more complicated and larger control flow graph: it uses numerous nested statements in its computations but it also uses frequent function calls and returns which limit the

Program	Calls	Returns	Indirect	Loop	Instructions per Branch
Compress	251,961	251,958	0	3,109,665	5.6
Elvis	56,707	56,704	14,358	2,795,743	3.7
Eqntott	4,012,205	4,012,202	318,927	89,438,385	3.3
Espresso	1,903,273	1,903,270	1,710	32,380,849	5.7
Gcc	694,040	694,037	285,775	4,849,735	5.1
Sc	5,286,492	5,265,772	851,636	17,966,589	4.2
Wc	81	78	10	105,326	3.1
Xlisp	6,769,493	6,694,400	918,293	9,292,560	4.6

Table 5.4: Dynamic call, return, indirect, loop-back branch counts and average number of instructions between branches for our benchmark programs.

effectiveness of if-conversion.

Program	Original		Guarded		Reduction (%)	
	Cond.	Uncond.	Cond.	Uncond.	Cond.	Uncond.
Compress	9,608,893	1,470,477	5,993,460	630	37.63	99.96
Elvis	2,049,003	38,358	624,471	27,345	69.52	28.71
Eqntott	250,605,253	5,323,008	94,032,938	2,569,882	62.48	51.72
Espresso	52,486,176	2,162,454	32,919,494	771,701	37.28	64.31
Gcc	6,986,908	673,838	4,508,646	297,886	35.47	55.79
Sc	62,136,131	3,190,986	54,068,197	3,105,803	12.98	2.67
Wc	543,254	43,796	351	39	99.94	99.91
Xlisp	23,424,087	2,791,589	18,184,506	1,969,861	22.37	29.44
Total/Average	407,839,705	15,694,506	210,332,063	8,743,147	46.58	67.65

Table 5.5: Dynamic conditional and unconditional branch counts for the original and guarded benchmark programs, and the corresponding percent reduction. These counts exclude call, return, indirect and loop branches.

If-conversion is also effective in eliminating the unconditional branches in the programs. For more than half of our benchmarks, more (sometimes significantly more) than half of all the unconditional branches were eliminated. For all programs except *Sc*, if-conversion eliminated at least 28% of the unconditional branches. For *Sc* however, guarding eliminated less than 3% of the unconditional branches, for the same reasons as in the case of conditional branches. In absolute numbers, the reduction in the number of unconditional branches is always less than the reduction in conditional branches. This behavior is expected, since the unconditional branches are only eliminated by if-conversion if they are part of a reconverging nested structure, in which case there would be at least on conditional branch eliminated for each unconditional branch.

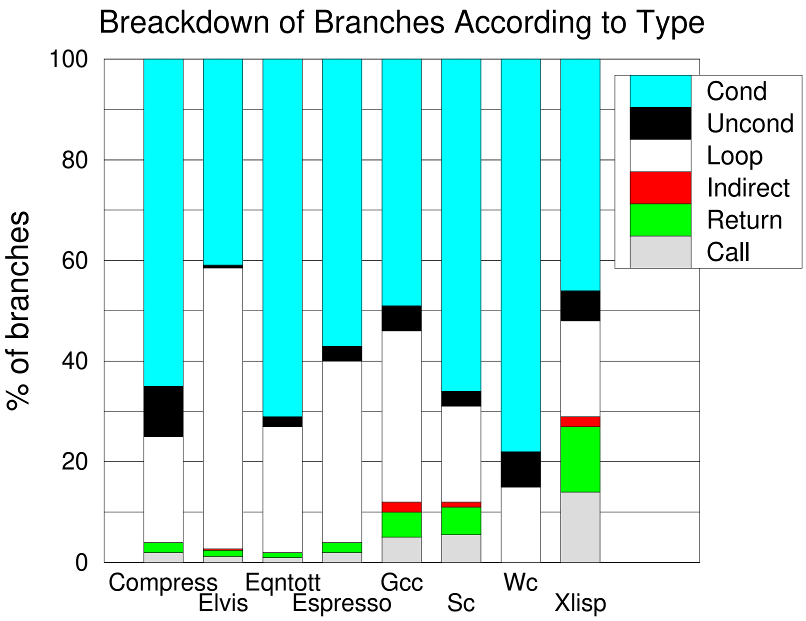


Figure 5.1: Breakdown of branches according to branch type.

5.4.3 Effects on dynamic branch behavior

Next, we turn our attention to the effect of guarding on the dynamic branch behavior of programs. To quantify the effects of guarding on the prediction accuracies of dynamic branch predictors we simulated a counter and a correlation based predictors, as described in Section 5.3.2. The prediction accuracies are obtained assuming that the actual outcome of a conditional branch is available immediately after the branch prediction is made. Therefore, our predictors always record the correct branch history, even when the prediction was incorrect. A real superscalar processor may perform multiple predictions before the branch outcomes are known; these predictions will be based on either old, or speculatively updated (and therefore possibly incorrect) branch history. However, using a timing simulator (which may predict based on the speculative branch history), we found that the actual prediction accuracies are very close (within 1%) to the accuracies obtained using perfect branch history. Other researchers have also reported that predicting based on speculative branch history usually has a small impact on the prediction accuracy [TYS+94a, HCP94].

Table 5.6 lists the dynamic prediction accuracies and the absolute number of mispredictions for the original and guarded version of the benchmark programs for a counter based predictor with 1024 2-bit counters. Guarding increases the prediction accuracies for all the programs except *Elvis*. The magnitude of the increase however ranges from less that 0.2 percentage points for *Sc* up to almost 16 percentage points for *Eqntott*. *Eqntott* is dominated by a single loop that contains two unpredictable branches; guarding eliminates both these branches and all the mispredictions that they cause, achieving a prediction accuracy of 97.76%. The small effect of guarding on the prediction accuracy of *Sc* are expected, given that if-conversion had a limited effect on the branch mix of this benchmark. On the average, the prediction accuracies increases from 88 to almost 93%.

Program	Prediction Accuracy		Number of Mispredictions		
	Original	Guarded	Original	Guarded	% Difference
Compress	89.77	90.01	1,503,363	1,008,594	32.91
Elvis	97.19	96.70	140,646	117,962	16.13
Eqntott	82.23	97.98	62,862,530	3,929,601	93.75
Espresso	88.82	89.30	10,156,622	7,609,463	25.08
Gcc	82.55	83.76	2,475,821	1,906,174	23.01
Sc	92.65	92.81	6,964,077	6,400,942	8.09
Wc	89.97	99.72	69,435	300	99.57
Xlisp	81.77	83.37	9,097,109	7,807,901	14.17
Total/Average	88.25	92.73	93,269,603	28,780,937	44.45

Table 5.6: Effects of guarding on the misprediction statistics for a counter-based predictor.

Guarding has a larger effect on the absolute number of mispredictions than on the prediction accuracy. From the last column of Table 5.6 we see that, with the exception of *Sc*, all the programs show a significant reduction in the number of mispredictions. *Wc* and *Eqntott* show an impressive reduction of more that 93%. *Sc* shows a small improvement of about 8% which is expected given the small effect of guarding in the number of conditional branches shown in Table 5.5. On the average, guarding reduces the number of mispredictions by 44%.

Table 5.7 lists the dynamic prediction accuracies and the absolute number of mispredictions for the original and guarded version of the benchmark programs for a correlation based predictor. Qualitatively, the results follow the same trends as in the case of a counter based predictor. However, the magnitude of the improvement is smaller than in the case of the counter based predictor, because the 2-level adaptive predictor achieves higher prediction accuracies for the base case. Guarding can actually reduce the prediction accuracy, as shown in the case of *Elvis*. Comparing prediction accuracies we see that guarding reduces the prediction accuracy from 97.86% to 97.72%; however, comparing the corresponding number of mispredictions for the original and guarded versions of *Elvis*, we see that the actual number of mispredicted branches is reduced by almost 24%. In essence, for *Elvis* guarding eliminates slightly more well-behaved branches than badly-behaved branches, resulting in a lower prediction accuracy but fewer mispredictions. As in the case of the counter based predictor, guarding has little effect on *Sc*. The prediction accuracy is increased by less than 0.3 percentage points and the number of mispredictions is reduced by about 9%. On the average, guarding increases the prediction accuracy from 92.14% to 94.17% and reduces the number of mispredictions by 46%.

An interesting anomaly occurs on *Wc*. The number of mispredictions for the correlation-based predictor is larger than for a counter-based predictor (302 versus 300). The reason for this (small) difference is that a 2-level adaptive predictor uses both the branch address and the last N past branch outcomes to determine which counter to use to predict a branch; since more than one counters may be used for a single static branch, the cold-start effects are larger for an adaptive predictor than for a counter based. Since *Wc* has a very simple control structure, the steady state of the counter and the correlation-based predictors are the same (i.e., they both predict taken the loop-back branch), and the longer cold-start period of the correlation based predictor results in a larger number of mispredictions.

The prediction accuracies in Table 5.7 at first glance may appear low for a 2-level adaptive

Program	Prediction Accuracy		Number of Mispredictions		
	Original	Guarded	Original	Guarded	% Difference
Compress	90.92	91.09	1,333,424	899,982	32.51
Elvis	97.86	97.72	107,071	81,590	23.80
Eqntott	93.52	98.58	22,909,099	2,760,576	87.95
Espresso	93.71	94.30	5,714,557	4,055,780	29.03
Gcc	82.90	84.80	2,425,284	1,784,309	26.43
Sc	93.05	93.30	6,579,131	5,960,121	9.41
Wc	95.74	99.71	29,529	302	98.98
Xlisp	85.03	87.37	7,467,367	5,931,761	20.56
Total/Average	92.14	94.17	46,565,462	21,474,421	46.09

Table 5.7: Effects of guarding on the misprediction statistics for a correlation-based predictor.

predictor. The reason is that these accuracies represent the combined prediction accuracy of the branch predictor and the BTB hit rate. When the branch prediction mechanism predicts a branch as taken but the branch misses in the BTB, the branch target is unknown and the prediction defaults to a not-taken prediction.

Figure 5.2 plots the cumulative number of run-lengths for our benchmark programs. In the plots of Figure 5.2, the top curve corresponds to the original benchmark and the lower curve corresponds to the guarded version of the benchmark. Both curves are scaled to the number of run-lengths of the original program. (Note that the curve for the guarded program will not reach 1, since the absolute number of taken branches, and therefore run-lengths, is smaller than in the original program.) The difference between the two curves shows the effectiveness of guarding to eliminated taken branches and to coalesce many small basic blocks into larger sequences of straight-line code. The most impressive change occurs in *Wc*: before guarding more than 70% of the run-lengths were less than 5 instructions, while after guarding almost all run-lengths are 25 instructions long. The behavior of *Eqntott* is very similar to the behavior of *Wc*, but with a smaller magnitude: the majority of the run-lengths are about 13 instructions long. Guarding has a uniform effect on *Compress*, *Espresso*, *Elvis* and *Xlisp* reducing the number of all run-length sizes by about the same amount; however, guarding has a very small effect *Sc*: the number of run-lengths smaller that 5 instructions is almost unaffected by guarding. The effect of guarding on *Gcc* is small for short runs, but more noticeable for larger ones.

Table 5.8 summarizes the dynamic branch behavior of our benchmarks, listing the average basic block size, the average run-length and the average misprediction distance. For the guarded programs we list the effective sizes, that is, we subtract the guarding overhead from the actual value of the corresponding metric. The effective sizes permit a direct comparison between the original and guarded sizes. In Table 5.8 we see that guarding increases the average basic block size from 4.4 to 7.9 instructions, while at the same time increasing the average run-lengths from 7.1 to 10.8 instructions. The effects of guarding are more pronounced on the average misprediction distance. For the counter-based predictor, the distance increased from 49 to 1314 instructions, while for the correlation-based, it increased from 70 to 1344 instructions. For the guarded version of *Wc*, the misprediction distance actually decreased by the use of the correlation based prediction, due to the cold-start reasons described earlier.

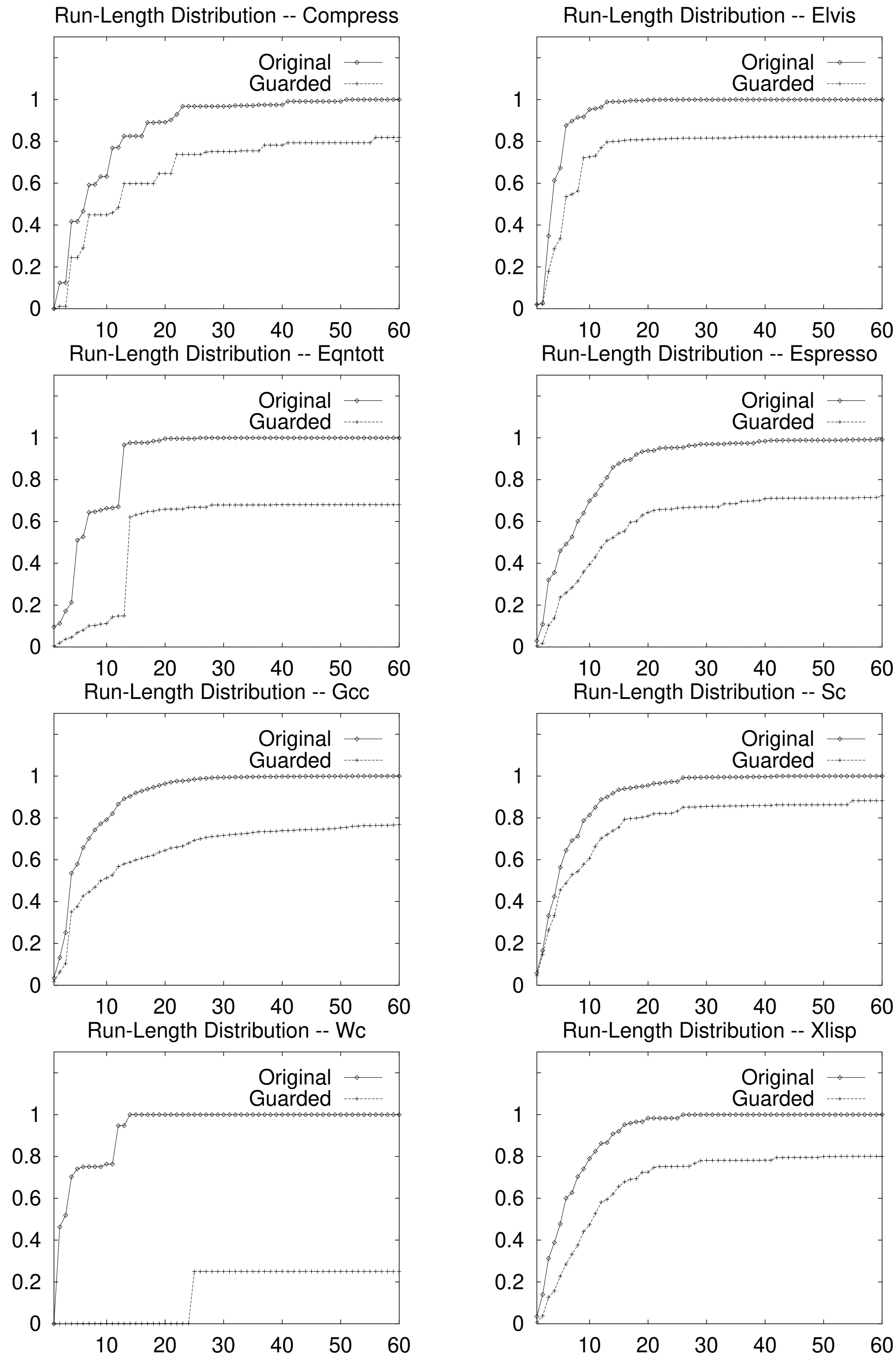


Figure 5.2: Run-length (number of instructions between taken branches) distributions for the original and guarded versions of our benchmark programs.

Program	Basic Block		Run-Length		Misprediction Distance			
					Original		Guarded	
	Original	Guarded	Original	Guarded	Counter	2-Level	Counter	2-Level
Compress	5.6	8.5	9.4	11.4	54	61	80	91
Elvis	3.7	5.1	4.9	5.9	131	172	156	226
Eqntott	3.3	5.9	7.6	11.2	18	50	295	421
Espresso	5.7	7.4	9.2	12.3	51	91	68	128
Gcc	5.1	6.2	6.8	8.8	29	29	37	40
Sc	4.2	4.7	6.8	7.7	57	61	62	66
Wc	3.1	20.6	5.1	20.6	31	73	7,253	7,205
Xlisp	4.6	5.1	7.0	8.7	25	30	28	38
Average	4.4	7.9	7.1	10.8	49	70	1,314	1,345

Table 5.8: Average basic block, run-length and misprediction distance for a counter and a 2-level adaptive predictor. The sizes for the guarded versions are the effective sizes, computed by removing the guard overhead computation.

5.4.4 Guarding region characteristics

Table 5.9 lists the average number of guard registers, the average number of instructions guarded by the same condition and the average guard register lifetimes for our benchmark programs. In Table 5.9 we distinguish between static and dynamic averages, where static is computed giving the same weight to each guarded region, and dynamic assigned the execution frequency for weight.

In Table 5.9 we see that the static average number of guard registers needed is quite small, below 3 for all programs. However, the static average may be skewed by the infrequently executed guarded region statistics. The dynamic averages reveal this skew, increasing slightly the averages to more that 2 for most programs and showing a maximum of about 3 for *Gcc*. Table 5.9 also lists the average number of instructions guarded by the same condition. This number is relatively small, ranging from about 3 to about 5.3. Across all benchmarks, the static number of instructions guarded by the same condition is about 4. Dynamically however, the number of instructions guarded by the same condition ranges more widely, from a low of 1.5 to a high of 6.54 instructions per condition.

Statically, the guard register lifetimes are between 9 and 19 instructions. Dynamically, this number increases for most programs; the exceptions are *Espresso*, *Eqntott* and *Wc* for which the guard register lifetimes decrease significantly to a dynamic average of 8.5, 4.22 and 12 instructions respectively. The largest guard register lifetimes occur for *Espresso* with a value of 22 instructions between definition and use.

The results in Table 5.9 suggest that a separate register file may not be required to support guarded execution, as the required number of guard registers is relatively small. Indeed, the "spill overhead" column in Table 5.9 shows that, for our benchmarks, the allocation of guard registers in a unified register file has little or no impact no the instruction count; the scheduler is able to allocate the guard conditions into unused registers with no extra cost. The highest spill overhead occurred for *Espresso* for which the instruction count was increased by about 2% due to the extra loads and stores. However, the relatively long guard register lifetimes indicate that if aggressive loop unrolling is used, the

Program	Guard Conditions per Region		Guarded Instructions per Condition		Guard Register Lifetimes		Spill Overhead
	Static	Dynamic	Static	Dynamic	Static	Dynamic	(%)
Compress	1.98	1.85	5.28	6.54	16.75	18.17	0.0
Elvis	2.08	2.08	3.86	3.47	13.09	8.51	0.1
Eqntott	1.77	2.84	3.99	2.32	12.06	4.22	0.0
Espresso	1.43	2.61	3.92	4.88	9.16	22.35	2.1
Gcc	2.51	3.03	4.03	4.42	17.58	20.72	0.2
Sc	1.87	2.84	3.81	3.91	12.54	19.74	1.2
Wc	2.75	2.00	4.21	1.50	19.39	12.00	0.0
Xlisp	1.94	1.93	2.93	3.13	10.22	11.04	0.3

Table 5.9: Static and dynamic guarding statistics: average number of guard conditions per guarded region, number of instructions guarded with the same condition, guard register lifetimes and spill overhead (as a percentage of the total instructions) due to register spill and reload for a unified register file.

requirements for guard registers will be increased dramatically. These transformations create multiple, overlapping copies of the code; the resulting code will contain more instances of these registers and the lifetimes will be lengthened, increasing the register requirements.

The results in Table 5.9 also verify the observations we made in Chapter 4 about the spatial locality of guarded computation. The small number of guard conditions per if-converted region indicates that the number of GUARD instructions required to guard an if-converted region would be small, while the reasonable guard register lifetimes indicate that instructions guarded with the same condition are in close proximity, within the reach of a guard mask of reasonable size.

5.5 Effects of Conditional Moves

Next, we examine the potential of using only conditional moves to support guarding and if-conversion. As described in Chapter 3, conditional move instructions can be used to synthesize other guarded instructions using a sequence of ordinary instructions that computes the result in a temporary register, and a conditional move to commit (according to the guard condition) the result in the actual destination register. As in the case of full guarding, the metrics of interest are the amount of overhead computation (the temporary computation instructions, the conditional moves, either squashed or committed), the reduction in the number of branches and the improvement in the dynamic branch behavior of the program.

5.5.1 Conditional Move guarding overhead

Table 5.10 shows the instruction counts for the original and the guarded versions of our benchmark programs; it also shows the guarding overhead as a percentage of the original instruction count, the number of conditional moves executed in the guarded program, and the percentage of the squashed

Program	Total Instructions		Overhead	Total	Squashed
	Original	Guarded	(%)	Cmoves	Cmoves (%)
Compress	81,972	87,292	6.49	2,817	26.62
Elvis	18,482	20,241	9.52	833	68.24
Eqntott	1,164,352	1,228,358	5.50	173,068	29.88
Espresso	522,085	573,330	9.82	22,393	16.90
Gcc	71,803	82,460	14.84	3,890	45.40
Sc	402,133	486,781	21.05	31,794	71.36
Wc	2,176	3,053	40.32	631	66.49
Xlisp	228,708	245,700	7.43	4,740	46.57
Total/Average	2,491,716	2,727,220	14.37	240,170	52.46

Table 5.10: Instruction counts for the original and guarded benchmarks (in thousands) and overhead (percentage) of conditional moves.

conditional moves (that is the percentage of conditional moves that did not commit their results due to a false condition).

The overhead of guarding is relatively small, less than 10% for most benchmarks, compared to an average overhead of about 20% for full guarding. The reason for the smaller overhead is that the guarding regions are smaller in the conditional move case, compared to the guarding regions for full guarding. Our region selection process has to ensure that unsafe instructions are either avoided or properly handled, and that all the synthesized sequences can be scheduled so that they do not increase the critical path of the computation. These restrictions force the selection of smaller guarding regions, resulting in smaller overhead,

The major exception is *Eqntott* which uses conditional moves for the majority of the guarded instructions. Even if the instruction set supports just conditional moves, it can still achieve almost the same effects as with full guarding with less overhead. The *Wc* case is also interesting: the main loop of *Wc* evaluates a number of conditions and depending on their values increments the word and line counters. This type of computation is safe, and can be easily if-converted using just conditional moves. The overhead of these conditional moves however is a substantial 40%, which is twice as much as the overhead required for if-conversion when the instruction set supports all the instructions in a guarded form.

The absolute number of conditional moves is not very high, in most cases less than 10% of the instruction count. However, the percentage of conditional moves that are squashed dynamically is fairly high at about 14.3% compared to the 8.5% of squashed computation for full guarding. This difference indicates that the guarding regions that are possible using conditional moves are less effective in capturing the useful computation than the guarding regions for full guarding.

5.5.2 Conditional Move branch counts

The effects of conditional moves on the branch counts is shown in Table 5.11. In general, the results are positive but not impressive in magnitude, with the exception of *Eqntott* and *Wc* and an average of 40%. This average is quite close to the average reduction for full guarding which is 48%. However, for unconditional branches conditional moves are much less effective. Across the benchmarks, conditional moves eliminated 27% of the unconditional branches compared to the 67% for full guarding.

Program	Original		Guarded		Reduction (%)	
	Cond.	Uncond.	Cond.	Uncond.	Cond.	Uncond.
Compress	9,608,893	1,470,477	7,216,292	1,470,476	24.90	0.00
Elvis	2,049,003	38,358	1,436,936	28,266	29.87	26.31
Eqntott	250,605,253	5,323,008	98,267,877	4,280,255	60.79	19.59
Espresso	52,486,176	2,162,454	47,635,506	1,314,368	9.24	39.22
Gcc	6,986,908	673,838	6,438,863	541,956	7.84	19.57
Sc	62,136,131	3,190,986	60,741,752	3,145,459	2.24	1.43
Wc	543,254	43,796	351	39	99.94	99.91
Xlisp	23,424,087	2,791,589	21,499,892	2,703,806	8.21	3.14
Total/Average	407,839,705	15,694,506	243,237,469	13,484,625	33.9	27.13

Table 5.11: Dynamic conditional and unconditional branch counts for the original and guarded benchmark programs, and the corresponding percent reduction. These counts exclude call, return, indirect and loop branches.

5.5.3 Effects of Conditional Moves on dynamic branch behavior

The impact of guarding using conditional moves on the dynamic branch behavior is shown in the next two tables. Table 5.12 lists the prediction accuracies for the original and guarded version of the program, the number of mispredictions for the original and guarded versions of each program and the percentage reduction in the number of mispredictions, for the counter-based dynamic predictor. Table 5.13 shows the same statistics but for the correlation-based dynamic branch predictor.

For the counter-based predictor, the effects of conditional moves is a general improvement on both the prediction accuracy and the number of mispredictions. The magnitude of the improvement depends on the program. As expected, *Eqntott* and *Wc* show the same dramatic improvement as they did with full guarding. The remaining of the programs however how a much smaller improvement. Overall, the use of conditional moves improves the average prediction accuracy from 88 to 92% and reduces the number of mispredictions by 36%.

For a correlation-based predictor, the effects of guarding using conditional moves are similar. The branch prediction accuracy improves for all programs except *Espresso*, and the average prediction accuracy increases from 92 to almost 93.7%. The number of mispredictions is reduced for all programs, but the magnitude of the reduction is less than 10% for half of the benchmarks; the average reduction is about 34% compared to a corresponding average of about 46% for full guarding.

Table 5.14 summarizes the effects of conditional moves on the dynamic branch behavior of our benchmarks, listing the average basic block size, the average run-length and the average misprediction distance. As in the case of the Full guarding, we list the effective number of instructions for the programs that are guarded using conditional moves. In this table we see that conditional moves increase the average basic block size from 4.4 to 7.1 instructions, while at the same time increasing the average run-lengths from 7.1 to 9.8 instructions. Guarding using conditional moves increased the average mis-

Program	Prediction Accuracy		Number of Mispredictions		
	Original	Guarded	Original	Guarded	% Difference
Compress	89.77	91.37	1,503,363	1,061,563	29.39
Elvis	97.19	96.92	140,646	135,382	3.74
Eqntott	82.23	96.16	62,862,530	7,690,198	87.77
Espresso	88.82	88.57	10,156,622	9,858,350	2.94
Gcc	82.55	83.09	2,475,821	2,314,224	6.53
Sc	92.65	92.64	6,964,077	6,777,381	2.68
Wc	89.97	99.72	69,435	300	99.57
Xlisp	81.77	82.78	9,097,109	8,525,128	6.29
Total/Average	88.25	92.38	93,269,603	36,362,526	36.07

Table 5.12: Effects of conditional moves on the misprediction statistics of a counter-based predictor.

Program	Prediction Accuracy		Number of Mispredictions		
	Original	Guarded	Original	Guarded	% Difference
Compress	90.92	92.01	1,333,424	982,285	26.33
Elvis	97.86	97.65	107,071	103,194	3.62
Eqntott	93.52	96.91	22,909,099	6,198,466	72.94
Espresso	93.71	93.63	5,714,557	5,495,650	3.83
Gcc	82.90	83.70	2,425,284	2,231,038	8.01
Sc	93.05	93.19	6,579,131	6,498,139	1.23
Wc	95.74	99.71	29,529	302	98.98
Xlisp	85.03	86.99	7,467,367	6,438,858	13.77
Total/Average	92.14	93.74	46,565,462	27,947,932	34.20

Table 5.13: Effects of conditional moves on the misprediction statistics of a correlation-based predictor.

Program	Basic Block		Run-Length		Misprediction Distance			
					Original		Guarded	
	Original	Guarded	Original	Guarded	Counter	2-Level	Counter	2-Level
Compress	5.6	6.6	9.4	10.4	54	61	77	82
Elvis	3.7	4.2	4.9	5.8	131	172	129	178
Eqntott	3.3	5.7	7.6	10.5	18	50	150	187
Espresso	5.7	6.0	9.2	9.9	51	91	52	94
Gcc	5.1	5.2	6.8	7.4	29	29	30	31
Sc	4.2	4.2	6.8	7.1	57	61	58	61
Wc	3.1	20.5	5.1	20.5	31	73	7,254	7,206
Xlisp	4.6	4.6	7.0	7.0	25	30	26	35
Average	4.4	7.1	7.1	9.8	49	70	1,284	1,295

Table 5.14: Average basic block, run-length and misprediction distance for a counter and a 2-level adaptive predictor using conditional moves. The sizes for the guarded versions are the effective sizes, computed by removing the guard overhead computation.

prediction distance from 49 to 1,284 instructions for the counter-based predictor, and from 70 to 1,295 instructions for the correlation-based predictor.

Conditional moves are easy to add to an instruction set, and appear to achieve effects very close to these of full guarding. However, the effects of conditional moves vary greatly from program to program. For programs with simple computations and regular control structure such as *Eqntott* and *Wc*, conditional moves are sufficient to achieve most if not all the benefits of guarding. For the rest of the programs however, the impact of guarding was severely reduced when limited to conditional moves, suggesting that support for full guarding is needed to exploit the potential of if-conversion.

5.6 Evaluation of GUARD support

In this section we evaluate the effects of GUARD instructions on the instruction mix of the programs. For the evaluation we use the same guarding regions the we used for the full guarding, with the only difference that we use GUARD instead of set instructions to evaluate the guard conditions. Since the guarding regions are the same as with full guarding, the branch and computation instruction counts and behavior will also the same. The branch prediction accuracy and number of mispredictions would also remain unaffected for infinite size prediction tables and BTBs; for finite size structures, the difference in the location of branches may change (either increase or decrease) the number of conflict misses in the BTB and prediction structures. This difference is random in nature and usually is quite small, so we do not address it in this thesis. The two main parameters that determine the effectiveness of GUARD instructions are: (i) the extent of ISA support for different types of GUARD instructions, and (ii) the width of the guard mask.

To evaluate the impact of the instruction set support for GUARD instructions we use two configurations. The first, uses only GUARDTRUE and GUARDFALSE instructions, which guard the instructions in their guard mask on the condition being true and false respectively. The second uses adds the GUARDBOTH instruction which can be used to to guard both paths of an if-then-else structure. Both configurations include a GUARDUPPER version of all GUARD instructions which specifies that the mask is first shifted by N bits, where N is the width of the guard mask in instructions. The total opcode requirements are 4 distinct opcodes for the first configuration, and 6 distinct opcodes for the second one. All the GUARD instructions use simple true-or-false comparisons of a single register as their guard condition.

Another important parameter that affects the efficiency of GUARD instructions is the guard mask size. A larger mask gives a larger scope to a GUARD instruction, allowing it to guard instructions that are farther away in the static program representation. In this section we evaluate two mask sizes: infinite, which means that the scope of a GUARD instruction will always reach up to the end of the guarded regions, and 21-bits, which is a realistic assumption for the MIPS-like instruction set that we use. While an infinite mask size is clearly impractical, it will gives us a lower bound to the total number of GUARD instructions required to guard a program.

When the mask size is limited to 21 bits, the scope of a GUARDTRUE or a GUARDFALSE instructions are 21 instructions. For the GUARDBOTH instructions we use the compact guard mask encoding described in Section 4.2.1, which results in a scope of 10 instructions. When a guard distance is larger that the actual mask size, a single GUARD instruction is not enough, and the simulator induces a sequence of GUARDUPPER instructions to guard the instructions outside the scope of the first GUARD instruction.

5.6.1 GUARD region characteristics

First we examine the guarded region characteristics that determine how effective are GUARD instructions in capturing the guarding state of the computation. The metric we use here are: (i) the number of instructions that are guarded by a single condition, (ii) the average guarding distance, i.e., how far from the GUARD instruction is the last guarded instruction, and (iii) the number of guard instructions required to guard the region.

Table 5.15 presents the average value for these metrics for our benchmark programs; the left half of the table lists the results when the instruction sets supports only GUARDTRUE and GUARDFALSE instructions while the right half lists the results when the instruction set supports the GUARDTRUE, GUARDFALSE and the GUARDBOTH instructions. For each of these instruction set configurations, Table 5.15 also lists the increase in the number of GUARD instructions caused by the use of a limited size mask.

Examining Table 5.15 we can identify some basic trends. First, we can see that the average guarding distance for our benchmarks varies from a low of 2.67 to a high of about 12.5 instructions, with an average of a little over 7 instructions, regardless of the combination of GUARD instruction available in the instruction set. The relatively small average distances indicate that a guard mask will not require an enormous number of bits to cover the entire guarding region. Another observation we can make from Table 5.15 is that the average number of instructions guarded per GUARD instruction is greater that 3 for most benchmarks, and it is more than 7 for half of the benchmarks. The average value is about 5.6 when the ISA supports only GUARDTRUE and GUARDFALSE instructions and about 6.2 instructions per GUARD when the ISA that supports GUARDBOTH instructions as well.

When the instruction set supports a GUARDBOTH in addition to the GUARDTRUE and GUARDFALSE instructions, the total number of GUARD instructions is expected to be reduced, since

Program	GUARD TRUE/FALSE				GUARD TRUE/FALSE/BOTH			
	# Instrs per GUARD	Distance	# GUARDS Infinite mask	# GUARDS 21-bit mask	# Instrs per GUARD	Distance	# GUARDS Infinite mask	# GUARDS 21-bit mask
Compress	7.30	9.43	2.20	+0.35	8.46	9.19	1.85	+0.46
Elvis	4.81	6.06	2.13	+0.07	5.01	5.78	2.02	+0.11
Eqntott	2.52	3.85	2.86	+0.01	2.61	3.64	2.77	+0.02
Espresso	7.62	9.10	2.32	+0.40	8.03	8.84	2.17	+0.48
Gcc	9.28	12.47	3.20	+0.78	10.35	12.26	2.79	+0.85
Sc	8.01	8.97	2.99	+0.16	8.45	9.04	2.81	+0.29
Wc	1.50	2.67	6.00	+0.00	2.25	2.75	4.00	+0.00
Xlisp	4.09	5.40	1.49	+0.02	4.20	5.34	1.44	+0.03
Average	5.64	7.24	2.90	+0.22	6.17	7.11	2.48	+0.28

Table 5.15: Dynamic GUARD Statistics: average number of instructions guarded per GUARD, guarding distance, number of GUARD instructions per guarded region for an infinite size mask, and the increment in this number when the mask is restricted to 21-bit. In the left half of the table, the instruction set supports only the GUARDTRUE and the GUARDFALSE instructions, while in the right half it supports the GUARDBOTH instruction as well.

the scheduler can replace a GUARDTRUE and a GUARDFALSE instructions that use the same guard register with a single GUARDBOTH instruction (if the guard distances are small enough to fit in the smaller scope of the GUARDBOTH instruction). This expectation is verified by comparing the two columns labeled “infinite mask” for the two instruction set configurations. When only GUARDTRUE and GUARDFALSE instructions are available, the required number of GUARD instructions ranges from 1.5 to 3.2, with an average of about 2.9. If we add the GUARDBOTH instruction, the range is between 1.44 and 2.81 instructions, with an average of about 2.5 GUARD instructions per guarded region.

The effects of a finite mask with 21 bits on the number of GUARD instructions are shown in the two columns labeled “21-bit mask” to be relatively small. When the instruction set supports GUARDTRUE and GUARDFALSE instructions, the largest increase occurs for *Gcc*: 0.78 additional GUARD instructions are needed per guarded region, corresponding to an increase of about 24% in the number of GUARD instructions. However, the change is much smaller for the rest of the benchmarks, with a magnitude less that 0.02 instructions for three of the benchmarks, while the average increase is 0.22 instructions. When we add the GUARDBOTH instruction, the largest increase occurs again for *Gcc* with a magnitude of 0.85 instructions or a 30% increase, and the average increase is 0.28 instructions.

5.6.2 Overhead of GUARD instructions

While the guarding region characteristics give insight about the efficiency of the GUARD instructions, an important metric is the GUARD *Overhead*, i.e., the number of GUARD instructions that the processor has to execute. This overhead depends on the guard mask size and on the type of the GUARD instructions that are supported in the ISA.

Program	Original Instructions	Computation Instructions	GUARD Overhead (% Computation)				Inits&Sets Avoided (% Comp.)
			Unlimited Mask		21-bit Mask		
			All three	GT/GF	All three	GT/GF	
Compress	81,972	82,562	9.52	11.30	11.91	13.12	14.60
Elvis	18,482	18,257	10.33	10.91	10.90	11.26	16.37
Eqntott	1,164,352	1,299,869	19.61	20.27	19.74	20.37	3.50
Espresso	522,085	577,177	8.32	8.90	10.18	10.45	13.94
Gcc	71,803	81,737	10.12	11.61	13.22	14.46	20.28
Sc	402,133	426,757	6.39	6.80	7.06	7.17	12.79
Wc	2,176	2,317	18.16	27.24	18.16	27.24	13.62
Xlisp	228,708	237,880	8.37	8.69	8.53	8.82	13.69
Total	2,491,716	2,726,560	11.35	13.21	12.46	14.11	13.60

Table 5.16: Overhead of GUARD instructions. The first two columns list the reference instruction count of the unmodified program, and the count of computation instructions in the if-converted program. The four overhead columns show the GUARD overhead as the percentage increase on the computation instruction counts for infinite and 21-bit masks and for instructions set that supports either the GUARDTRUE and GUARDFALSE instructions only, or the GUARDBOTH instruction as well. The last columns lists the amount of set and initialization instructions that is avoided by the use of GUARD instructions.

Table 5.16 lists the instruction count of the original, non-if-converted program, and the number of computation (non-GUARD) instructions in the guarded program. It also lists the GUARD overhead as a percentage of the computation instructions, for both an unlimited and a 21-bit guard mask sizes and for two ISAs: one that support the GUARDTRUE and GUARDFALSE instructions only (labeled "GT/GF"), and one that supports the GUARDBOTH instruction as well (labeled "All three"). To compute the actual instruction count for a program and for a given instruction set support and mask size, one can take the number of computation instructions, and add to it the GUARD overhead for that ISA support and mask size configuration. For example, the instruction count for *Compress* using all three GUARD instructions and a 21-bit guard mask would be: 82,562 * (1 + 11.91%) which comes out to 92,395 instructions.

In Table 5.16 we see that for most benchmarks, and for all combinations of guard mask size and ISA support, the overhead of GUARD instructions is less than 12%. Adding the GUARDBOTH instruction in the instruction set reduces the overhead by less than 1%. Similarly, the impact of going from an infinite size to a 21-bit guard mask increases the overhead by less than 1%. Averaging across the benchmarks, the smallest overhead is 11.35% for an unlimited size mask and for an ISA supporting all three flavors of GUARD instructions, while the largest overhead is 14.11% for a 21-bit guard mask and for an ISA supporting only the GUARDTRUE and GUARDFALSE instructions.

The two notable exceptions from these trends are *Eqntott* and *Wc* which require an overhead of almost 20%; furthermore, removing the support for a GUARDBOTH instruction increases the GUARD overhead from 18% to 27% for *Wc*.

Table 5.16 also list the number of set and initialization instructions that would have been needed had we used full guarding, again as a percentage of the computation instructions. These initializations and sets are are obviated by the use of GUARD instructions. Therefore, for a fair comparison of GUARD instructions and ordinary guarding, one should compare the GUARD overhead to the amount of set and initialization that are avoided. For example, comparing the last two columns of the table, we see that for most programs, the guard overhead is actually less than the amount of initializations and sets that are eliminated; this means that the instruction count will actually be smaller when we use GUARD instructions instead of ordinary guarding.

Eqntott and *Wc* are again exceptions to that observation: their GUARD overhead is substantially more than the overhead of condition manipulation instructions in full guarding. The explanation for this phenomenon is that GUARD instructions are able to eliminate sets when the guard conditions are nested; both *Eqntott* and *Wc* have relatively simple control structures with very few nested control structures, eliminating that potential benefit of GUARD instructions. The net GUARD overhead is less than 1% of the program computation over the instruction count of the program using ordinary guarding with explicit guard operands per instruction.

5.7 Summary

In this chapter we performed an evaluation of the impact of guarding on the instruction characteristics and the dynamic branch behavior of our benchmarks. For full guarding and for our evaluation assumptions, we found that: (i) the amount of the overhead computation in a guarded program is not very high at about 20%, and the amount of squashed computation is about 8.5%. (ii) the effectiveness of guarding in eliminating branches varies heavily with each program, from an unimpressive 13% to a stunning 99%, (iii) the number of conditions per guarded region is small, but the lifetimes of the guard registers are in most cases more than 10 instructions. (v) the effect of guarding on the dynamic branch prediction is correlated with the effectiveness of branch elimination; overall guarding eliminates close to half of the mispredictions for both counter- and correlation-based predictors.

It is hard to draw conclusions about the effectiveness of synthesizing guarding using conditional moves. Conditional moves perform extremely well for two benchmarks, *Eqntott* and *Wc*; for the remaining of benchmarks the impact of conditional moves on our metrics was limited. Overall, the average values of our metrics suggest that conditional moves are successful in capturing more than half of the potential of full guarding.

We found the overhead of GUARD instructions, under realistic assumptions, to be small: the instruction count with GUARD instructions is less than 1% larger than the instruction count using ordinary guarding. Furthermore, for many programs, GUARD instructions actually reduced the instruction count compared to ordinary guarding. The overhead of GUARD instructions is not very sensitive to the mask length: when we limiting the mask size from infinite to 21 bits introduces less than 1% overhead in the total instruction count of the program. Finally, we found that the support for GUARDBOTH instruction is not crucial to the performance of GUARD instructions. However, adding the GUARDBOTH instruction reduces the GUARD overhead for *Wc* considerably.

These results show that guarding has indeed the potential to improve performance, even under our conservative assumptions, and without any profiling information. However, these results are just hints of the expected performance of guarding. Without supporting results using timing simulations, these results are of little value. In the next chapter we will address this limitation, and evaluate the real performance benefits of guarding using timing simulations of realistic processor configurations.

Chapter 6
Effects of guarding on execution time

In the previous chapter, we evaluated the impact of guarding on the program characteristics, such as the number of branches in the program, the branch prediction accuracy, etc. However, these metrics are just indicative of the performance potential; an improvement in any of these metrics may, or may not, translate into faster program execution. The ultimate performance metric for any evaluation is the execution time of a program. In this chapter we evaluate the performance of guarding using the execution time of the programs as our metric. To do so, we use detailed cycle-by-cycle simulations to determine the execution time (in cycles) of each of our benchmarks.

The performance of the guarded programs depends on many parameters. In this chapter, we concentrate on four major parameters: (i) the processor's issue model, which can be either in-order or out-of-order, (ii) the issue width, i.e., how many instructions the processor can issue per cycle, (iii) the structure of the dynamic predictor, and (iv) the branch misprediction penalty. We measure the effects of these parameters for each of the three levels of ISA support for guarding we used in Chapter 5: full guarding, conditional moves, and GUARD instructions.

This chapter is organized as follows. Section 6.1 describes in detail our simulation assumptions and methodology. Section 6.2 investigates the performance potential of guarding in the context of an in-order issue processor. Section 6.3 measures the impact of the misprediction penalty on the performance of guarding. Section 6.4 evaluates the performance of guarding on an out-of-order issue processor and Section 6.5 summarizes this chapter.

6.1 Simulation Methodology

For our evaluation we use the SimpleScalar timing simulation tools. The benchmark programs were compiled into binaries using the compilation methodology described in Chapter 5, and then fed into the SimpleScalar in-order and out-of-order timing simulators. These simulators are execution driven, and model all aspects of the pipeline, functional units, caches, etc. Both the in-order and the out-of-order processors support dynamic branch prediction and speculative execution of instructions. We use the same predictors we used in Chapter 5. The processors can perform a single prediction per cycle, but can execute multiple branch branches per cycle, since the only necessary action is the verification of the corresponding predictions. Next, we describe the parameters of our in-order and out-of-order processor models, and the latencies we assume for each of the processor components.

6.1.1 In-order execution model

Our in-order issue execution model is a symmetric superscalar of degree N. It supports speculative execution and allows the out-of-order completion of instructions via a score-boarding mechanism. The processor pipeline is assumed to be a simple, 5-stage pipeline, with a base branch misprediction cost of 1 cycle.

The memory systems is assumed to be multi-ported by replication, and it can support up to N loads or a single store per cycle. Loads and stores are executed in program order. For all the simulations, we use a 16Kbyte, non-blocking, direct-mapped cache. The cache uses a write-allocate policy and the block size of 16 bytes. To decouple the execution of loads and stores, the processor implements a 16-entry, non-merging write buffer. for our simulations we assume a perfect instruction cache with 100% hit ratio.

The integer multiplier, the floating point adder and floating point multiplier functional units are fully pipelined and can support a completion rate of one result per cycle. The integer as well as the floating point dividers are not pipelined.

When the processor decodes a guarded instruction, it evaluates the guard condition in the decode stage of the pipeline, and if the condition is false, the instruction is transformed into a NOP. However, the squashed instruction consumes the instruction fetch bandwidth, inhibiting in this way the execution of other instructions.

The processor supports GUARD instructions, as described in Chapter 4; the GUARD masks have either an infinite size, or they are 21 bits wide. When an instruction enters the decode stage of the pipeline, the corresponding bit of the scalar mask register is checked; if it is zero, the instruction is converted into a NOP. The instruction decoder detects sequences of GUARD instructions immediately followed by their guarded computation and short-circuits the guard condition in the computation instruction, allowing both the GUARD instruction, and the instructions it guards to be issued in the same cycle. The guarded computation will be squashed (if needed) by the guard condition by the end of the decode stage.

6.1.2 Out-of-order execution model

Our out-of-order processor uses a Reorder Update Unit (RUU) [SV87, Soh90] to achieve instruction buffering and register renaming. Guarded instructions are squashed during the decode stage if the guard condition is available. When the guard condition is not available, the guarded instructions are treated as selects between the old value and the (potential) new value of the destination register (as described in Section 2.3.3) and are issued in the RUU. These select statements are considered ready when, either the guard condition is false and the old value of the destination register is valid, or when the guard condition is true and all the operands of the instruction are valid. To allow the out-of-order execution of GUARD instructions, the processor uses the Guard Mask Buffer described in Section 4.5. Our out-of-order processor model uses the same memory system configuration as our in-order processor model.

6.1.3 Functional unit latencies

For our evaluation we use a realistic processor configuration. Table 6.1 shows the functional unit latencies. For all our simulations we use a cache miss latency is six cycles, assuming a perfect hit ratio in the second level cache. We use a branch misprediction penalty of one cycle from the time the branch in resolved. This misprediction penalty is quite low, corresponding to a very aggressive and very short pipeline. A larger misprediction penalty would boost the results of guarding and its effects. To evaluate this impact, in Section 6.3 we will present the simulation results for misprediction penalties of two, three, and eight cycles.

Functional Unit	Latency (Cycles)	Pipelined?
Integer ALU	1	N/A
Integer Multiplication	4	Yes
Integer Division	12	No
Floating Point Addition	2	Yes
Floating Point Multiplication	4	Yes
Floating Point Division	12	No
Floating Point Square Root	24	No

Table 6.1: Functional Unit Latencies.

6.1.4 Metrics

In this Chapter we use three metrics: (i) Execution Time, (ii) Instruction Per Cycle (IPC) completion rate and (iii) Speedup. The execution time is self-explanatory and includes all cycles spend for execution of useful instructions, data cache miss latencies etc. The IPC completion rates are just the number of instructions divided by the execution time. However, for guarded programs we distinguish two types of IPC rates: the *Effective IPC*, which is computed by dividing the number of instruction in the *original* program by the execution time of the guarded program, and the *Actual IPC* which is computed by dividing the number of instructions (including all the squashed computation and guarding overhead) by the execution time. (The actual IPC can be computed from the Effective IPC of a program by adding the guarding overhead shown in Table 5.2 in Chapter 5.) The speedup metric is straightforward, and is computed by dividing the Effective IPC of the guarded program by the IPC of the original program for the same processor configuration and subtracting one. The resulting speedup is reported as a percentage. To compute the average IPC, we just compute the *harmonic* mean of the IPCs of all the programs. To compute the average speedup, we first compute the average IPCs and then we use the speedup calculation on the average IPC values.

6.2 Performance of guarding for an in-order issue processor

First, we investigate the performance potential of guarding in the context of an in-order issue processor. We consider the three levels of ISA support for guarding, full guarding, conditional moves, and GUARD instructions, in this order. For each of these levels of ISA support for guarding, we simulate the benchmarks for two processor configurations, one that can issue 4 and another that can issue 8 instructions per cycle. Similarly, to determine the effects of the dynamic branch predictor on the performance of guarding we simulated the counter- and correlation-based dynamic predictors described in Chapter 5, Section 5.3.2.

6.2.1 Performance of full guarding

We begin our evaluation by examining the performance of full guarding. Table 6.2 lists the execution times for the original and guarded version of the benchmarks for a 4-issue processor using a counter-based predictor. It also shows the instruction completion rate (IPC) and the speedup that guarding

Program	Execution Time		IPC			Speedup
	Original	Guarded	Original	Effective	Actual	(%)
Compress	83,067	80,257	0.99	1.02	1.18	3.5
Elvis	12,429	11,540	1.49	1.60	1.84	7.7
Eqntott	855,862	695,257	1.36	1.67	1.94	23.1
Espresso	448,670	424,474	1.16	1.23	1.54	5.7
Gcc	56,991	54,033	1.26	1.33	1.82	5.5
Sc	347,252	333,255	1.16	1.21	1.44	4.2
Wc	1,188	881	1.83	2.47	2.99	34.9
Xlisp	197,275	187,167	1.16	1.22	1.48	5.4
Total/Average	2,002,738	1,786,867	1.26	1.37	1.66	9.1

Table 6.2: Execution time (in thousand cycles) and IPC for the original and the fully guarded programs on a 4-issue processor using a counter-based predictor.

achieves as a percentage. Table 6.2 lists all three IPC values: Original, which is the IPC of the original, unmodified program, Effective and Actual.

In Table 6.2 we find two striking results. First, two of the programs (namely *Eqntott* and *Wc*) perform very well with guarding, showing a speedup of about 23% and 35% in their execution time on a 4-issue processor. The second result is that for the rest of the benchmarks, guarding has a modest effect on the execution time, ranging from a 3.5 to a 7.7% speedup. Modest results may have been expected for *Sc*, given the small effect of guarding and if-conversion on the control structure. The poor performance of *Compress* is also explained by its sequential nature: the main computation of *Compress* is a recurrence on the input file, leaving few opportunities for gains through scheduling; furthermore, the working set size of compress is several times larger than the data cache, causing a large number cache misses that dominate the execution time. The instruction completion rate is modest for all programs except *Wc*: the IPC values range from 0.99 to 1.49 for the unmodified program; *Wc* achieves an effective IPC of 1.83. Table 6.2 also lists the average IPC and speedup. For the original program the average IPC is 1.26; full guarding increases this average to 1.37, or by 9%. To achieve this effective IPC rate, the processor executes 1.66 instructions per cycle.

Table 6.3 shows the same statistics for an 8-issue processor. For the unmodified programs, a wider issue improves the execution time by a marginal amount, indicating that a 4-issue processor has sufficient resources to exploit the instruction level parallelism of the original program (as scheduled by the compiler). On the average, the 8-issue processor increases the IPC of the original programs from 1.26 to 1.28.

The plentiful processor resources have a larger impact of the performance of the guarded program. One of the factors that limit the performance of the guarded version of the programs is the burstyness of the guard condition evaluation (as described in Section 2.2.1). In many cases, multiple guard conditions need to be set before we execute the guarded computation, and these instructions can (temporarily) deplete the issuing capacity of a 4-issue processor. Similarly, squashed computation, even if it is not executed, depletes the issue bandwidth of the processor, excluding other instructions from executing. A wider-issue processor is better able to smooth out this bursty behavior, and provide better speedups. A good example of this situation is *Wc* which has more that 20% overhead computation (see

Program	Execution Time		IPC			Speedup
	Original	Guarded	Original	Effective	Actual	(%)
Compress	82,593	78,810	0.99	1.04	1.20	4.8
Elvis	12,294	11,362	1.50	1.63	1.87	8.2
Eqntott	847,490	690,984	1.37	1.69	1.95	22.6
Espresso	447,064	404,949	1.17	1.29	1.61	10.4
Gcc	55,283	51,582	1.30	1.39	1.91	7.2
Sc	341,731	320,872	1.18	1.25	1.50	6.5
Wc	1,107	776	1.97	2.80	3.39	42.7
Xlisp	194,157	180,275	1.18	1.27	1.53	7.7
Total/Average	1,981,722	1,739,612	1.28	1.43	1.72	11.2

Table 6.3: Execution time (in thousand cycles) and IPC for the original and the fully guarded programs on a 8-issue processor using a counter-based predictor.

Table 5.2 in Chapter 5); the 8-issue processor increased the effective IPC by 13%, from 2.47 to 2.80. However, the execution time speedups for the rest of the benchmarks are low, less than 10% for more than half of our benchmarks, and the average speedup is 11%. The corresponding effective IPC is 1.43, and the actual IPC is 1.72 instructions per cycle.

Next we consider the effects of a correlation-based predictor. In general, a better branch prediction mechanism reduces the processor stalls due to incorrect predictions, and allows the original program to sustain better baseline performance, thus reducing the performance potential of guarding. Table 6.4 lists the execution times and IPCs for the original and guarded version of the benchmarks for a 4-issue processors using a correlation-based predictor.

When the misprediction rate is small, the only benefits of guarding are due to (a) better static instruction schedules and (b) more efficient sequential fetch of the instructions in the guarded regions (that is, fewer taken branches to disrupt instruction fetching). Considering the prediction accuracies in Tables 5.6 and 5.7 of Chapter 5, we see that the correlation-based predictor achieves much better prediction accuracies than the counter-based predictor for the original (non-if-converted) programs. As expected, the better prediction accuracy reduces the effects of guarding: for the 4-issue processor, *Eqntott* and *Wc* perform the best with a speedup of 10 and 22% in their execution times.The remaining programs, however, show only small speedups in the range of 3 to 8.4%. The overall average speedup is 6.5%, while the average effective IPC achieved by guarding is 1.39.

As in the case of a counter-based predictor, increasing the issue width of the processor from 4 to 8 helps the guarded version of the program more than the original, resulting in slightly better speedups. Table 6.5 shows the execution time statistics for an 8-issue processor using correlation-based prediction. All programs except *Wc* show speedups of less that 9%. Overall, guarding achieves a average speedup of 7.8% over the original program execution times.

Comparing the tables for counter- and correlation based prediction, can draw some interesting conclusions. First, the combination of full guarding and a counter based predictor gives shorter execution times than that of the original programs running on a processor with correlation-based prediction. For example, with guarding, the effective IPC for a 4-issue processor using a counter-based predictor is 1.37, compared to the 1.31 IPC for a processor using correlation-based prediction but no guarding; for

Program	Execution Time		IPC			Speedup
	Original	Guarded	Original	Effective	Actual	(%)
Compress	82,594	80,158	0.99	1.02	1.18	3.0
Elvis	12,374	11,418	1.49	1.62	1.86	8.4
Eqntott	739,473	670,419	1.57	1.74	2.01	10.3
Espresso	436,110	414,978	1.20	1.26	1.57	5.1
Gcc	56,198	53,427	1.28	1.34	1.84	5.2
Sc	340,629	327,851	1.18	1.23	1.47	3.9
Wc	1,077	881	2.02	2.47	2.99	22.2
Xlisp	193,146	185,554	1.18	1.23	1.49	4.1
Total/Average	1,861,604	1,744,690	1.31	1.39	1.68	6.5

Table 6.4: Execution time (in thousand cycles) and IPCs for the original and the fully guarded programs on a 4-issue processor using a correlation-based predictor.

an 8-issue processor the corresponding IPCs are 1.43 and 1.33. Second, a correlation-based predictor cuts the guarding speedups by about 30% (from 9 to 6.5%, and from 11.2 to 7.8%).

6.2.2 Performance of guarding using conditional moves

Next we turn out attention to the performance of conditional move instructions as a way to implement guarding. As described in Chapter 2, arbitrary guarded instructions can be synthesized using ordinary computation into temporary registers, followed by a conditional move to commit the result in the actual destination register when the condition evaluates to true. However, these instruction sequences lengthen the dependence paths thought the code compared with full guarding. Therefore, conditional moves are expected to show smaller speedups than full guarding.

Table 6.6 shows the execution time, IPCs and speedup using conditional moves on a 4-issue processor using a counter-based branch predictor. The table also includes the effective and the actual IPC and the corresponding percent speedup. For comparison purposes the table also includes the speedups using full guarding on the same processor configuration. In Table 6.6 we see that the speedups that conditional moves are able to show substantial speedup for the benchmarks *Eqntott* and *Wc*: 21 and 20% respectively, compared to the 23.7 and 35% of full guarding. The speedups for the rest of the programs are smaller than 4.2%. On the average, for a 4-issue processor, conditional moves give a 6.3% speedup, or about 70% of the speedup of full guarding (9.1%). That speedup increases the average IPC of the programs from 1.26 to 1.34, while the actual processor IPC (including all overheads) is 1.51.

Table 6.7 shows the execution time statistics for an 8-issue processor using a counter-based predictor. The longer computation paths due to the extra conditional moves limit the performance of conditional moves for wider issue processors. Comparing Tables 6.6 and 6.7 we see that the impact of wide issue processor is smaller on the IPCs for conditional moves than it is for full guarding. For most programs the effective IPCs change only slightly. The only exceptions are *Eqntott* and *Wc* which exhibit an increase of more than 20% in their IPCs. Overall, for an 8-issue processor, conditional moves increase the average IPC from 1.28 to 1.37. The average speedup is 7.1%, compared to 11.2% for full

Program	Execution Time		IPC			Speedup
	Original	Guarded	Original	Effective	Actual	(%)
Compress	82,052	78,721	1.00	1.04	1.20	4.2
Elvis	12,235	11,326	1.51	1.63	1.88	8.0
Eqntott	726,353	666,990	1.60	1.75	2.02	8.9
Espresso	434,388	398,876	1.20	1.31	1.63	8.9
Gcc	54,391	51,441	1.32	1.40	1.91	5.7
Sc	338,027	318,544	1.19	1.26	1.51	6.1
Wc	973	776	2.24	2.80	3.39	25.4
Xlisp	189,988	179,596	1.20	1.27	1.54	5.8
Total/Average	1,838,410	1,706,274	1.33	1.44	1.74	7.8

Table 6.5: Execution time (in thousand cycles) and IPCs for the original and the fully guarded programs on a 8-issue processor using a correlation-based predictor.

Program	Conditional Moves					Full
	Execution	IPC			Speedup	Guarding
	Time	Original	Effective	Actual	(%)	Speedup (%)
Compress	81,367	0.99	1.01	1.07	2.1	3.5
Elvis	11,929	1.49	1.55	1.70	4.2	7.7
Eqntott	705,862	1.36	1.65	1.74	21.3	23.1
Espresso	432,670	1.16	1.21	1.33	3.7	5.7
Gcc	55,254	1.26	1.30	1.49	3.1	5.5
Sc	337,952	1.16	1.19	1.44	2.8	4.2
Wc	986	1.83	2.21	3.10	20.5	34.9
Xlisp	190,275	1.16	1.20	1.29	3.7	5.4
Total/Average	1,816,298	1.26	1.34	1.51	6.3	9.1

Table 6.6: IPCs and speedup achieved by conditional move instructions for an 4-issue in-order processor using a counter-based predictor.

Program	Conditional Moves					Full Guarding
	Execution Time	IPC			Speedup (%)	Speedup (%)
		Original	Effective	Actual		
Compress	80,967	0.99	1.01	1.08	2.0	4.8
Elvis	11,635	1.50	1.59	1.74	5.7	8.2
Eqntott	700,862	1.37	1.66	1.75	20.9	22.6
Espresso	421,760	1.17	1.24	1.36	6.0	10.4
Gcc	54,019	1.30	1.33	1.53	2.3	7.2
Sc	331,552	1.18	1.21	1.47	3.1	6.5
Wc	881	1.97	2.47	3.46	25.6	42.7
Xlisp	185,775	1.18	1.23	1.32	4.5	7.7
Total/Average	1,787,454	1.28	1.37	1.54	7.1	11.2

Table 6.7: IPCs and speedup achieved by conditional move instructions for an 8-issue in-order processor using a counter-based predictor.

guarding.

Qualitatively, the relative performance of conditional moves and full guarding remains unaffected by the choice of dynamic branch predictor. Tables 6.8 and 6.9 present the execution time statistics for conditional moves using a correlation-based predictor for a 4- and an 8-issue processor respectively. As expected, the more accurate predictor reduces the speedups achieved by conditional moves. For more than half of the programs, the speedup over the original program is less than 3%, regardless of the issue width of the processor. The only program that shows a speedup greater than 10% is *Wc*, and requires an 8-issue issue to achieve it. On the average, conditional moves achieve a speedup of 3.2 and 3.5% for a 4- and an 8-issue issue processor respectively, compared to the 6.5 and 7.8% speedups with full guarding.

6.2.3 Performance of GUARD instructions

Next, we evaluate the performance of GUARD instructions. As described in Chapter 5, when we compile the programs with GUARD instructions, we use the same guarding regions as in the case of full guarding; therefore, the performance of GUARD instructions is expected to be comparable with – if not better than – the performance of full guarding. In fact, we expect it to be better, since, as we showed in Chapter 5, Section 5.6.2, GUARD instructions are able to reduce, in most cases, the instruction count of the guarded program, and, as described in Chapter 4, Section 4.4, GUARD instructions allow the early squashing of guarded computation.

To examine the impact of limited size guard masks on the execution time of the programs, we simulated all four of the configurations that we used in Chapter 5, namely, two guard mask sizes: infinite and 21 bits, and two instruction set support levels for GUARD instructions: GUARDTRUE and GUARDFALSE only, or GUARDTRUE GUARDFALSE and GUARDBOTH. Lastly, we evaluate the effectiveness of "early squashing", that is the ability of a processor that supports GUARD instructions to squash instructions before they are even fetched (using the skip capability described in Chapter 4, Section 4.2.2.

Program	Conditional Moves					Full Guarding Speedup (%)
	Execution Time	IPC			Speedup (%)	
		Original	Effective	Actual		
Compress	81,567	0.99	1.00	1.07	1.3	3.0
Elvis	11,859	1.49	1.56	1.71	4.3	8.4
Eqntott	706,312	1.57	1.65	1.74	4.7	10.3
Espresso	426,670	1.20	1.22	1.34	2.2	5.1
Gcc	54,491	1.28	1.32	1.51	3.1	5.2
Sc	334,252	1.18	1.20	1.46	1.9	3.9
Wc	986	2.02	2.21	3.10	9.2	22.2
Xlisp	188,475	1.18	1.21	1.30	2.5	4.1
Total/Average	1,804,617	1.31	1.35	1.52	3.2	6.5

Table 6.8: IPCs and speedup achieved by conditional move instructions for an 4-issue in-order processor using a correlation-based predictor.

Program	Conditional Moves					Full Guarding Speedup (%)
	Execution Time	IPC			Speedup (%)	
		Original	Effective	Actual		
Compress	81,067	1.00	1.01	1.08	1.2	4.2
Elvis	11,672	1.51	1.58	1.73	4.8	8.0
Eqntott	694,741	1.60	1.68	1.77	4.6	8.9
Espresso	421,481	1.20	1.24	1.36	3.1	8.9
Gcc	52,991	1.32	1.36	1.56	2.6	5.7
Sc	327,252	1.19	1.23	1.49	3.3	6.1
Wc	881	2.24	2.47	3.46	10.4	25.4
Xlisp	185,575	1.20	1.23	1.32	2.4	5.8
Total/Average	1,775,664	1.33	1.38	1.55	3.5	7.8

Table 6.9: IPCs and speedup achieved by conditional move instructions for an 8-issue in-order processor using a correlation-based predictor.

Table 6.10 shows the speedups obtained using the four possible combinations of GUARD instructions for a 4- and an 8-issue issue processor using a counter-based predictor. For comparison purposes, the table also lists the speedups obtained using full guarding. Comparing the speedups for the four GUARD support combinations we see that the impact of either a limited size guard mask or of adding the GUARDBOTH instruction, is negligible for all programs except *Gcc*; for *Gcc* and an instruction set supporting GUARDBOTH instructions, reducing the guard mask size from infinite to 21 bits reduces the speedup by 0.6 and 1 percentage points for a 4- and an 8-issue processor respectively. Comparing the performance of GUARD instructions to that of full guarding we see that GUARD always performs better; for a 4-issue processor, GUARD instructions achieve a speedup of more than 13% compared to the 9.1% speedup of full guarding. For an 8-issue processor these averages are about 15 versus 11.2%. It is also important to note that even for a 4-issue processor, GUARD instructions achieve speedups of more than 10% for more than half of the programs.

The impact of a correlation-based predictor on the performance of GUARD instructions is similar to that on full guarding. Table 6.11 shows the speedups obtained using GUARD instructions for a 4- and an 8-issue issue processor, along with the speedups for full guarding. As expected, the magnitude of the speedups is reduced, and GUARD instructions achieve a speedup of about 10% for both 4- and 8-issue processors, compared to 6.5 and 7.8% for full guarding.

Considering the result in Tables 6.10 and 6.11, we see that GUARD instructions perform consistently better that full guarding. This is a result of the shorter (both in number of instructions, and in dependence path) condition evaluation sequences that are possible using our GUARD assignment algorithm.

Another way in which GUARD instructions can improve the performance of a program, is by allowing the processor to squash some computation before it is even fetched. This ability requires the "skip-logic" circuit described in Chapter 4, Section 4.4. Early squashes will only happen for GUARD instructions with a false condition, and then, only when their guarded computation has not already been fetched from the instruction cache. For example, for a 4-issue processor using a 5-stage pipeline, when the GUARD instruction is decoded and executed, it may be the first in a parcel of four instructions. While the GUARD instruction is being decoded and executed, the fetching of the next parcel of four instructions is already underway in the I-fetch stage of the pipeline. Therefore, the processor can avoid fetching the squashed computation only if the computation is 8 instructions away from the GUARD instruction. This distance increases when the issue width of the processor is increased, or when the distance (in pipeline stages) between the execution stage of GUARD instructions and instruction fetch stage is increased.

Table 6.12 evaluates the ability of a processor supporting GUARD instructions to squash some of the guarded computation early. The table lists the instruction count of each program when compiled using 21-bit guard masks and all three flavors of GUARD instructions, The four rightmost columns of the table show the number of instructions that are squashed early for a 4- and an 8-issue processor. The number of early squashes is shown in two ways: (i) as a percentage of all the squashed instructions (in the two columns labeled "% squashed") and (ii) as a percentage of the total instruction count of the program (in the columns labeled "% instructions").

As a percentage of all the squashed instructions, early squashes happen frequently on a 4-issue processor. For example, for *Wc*, 40% of the squashed computation is bypassed by the early squashing mechanism. *Gcc* shows the smallest percentage of early squashes, with less than 10% of the squashed computation being squashed early. On the average, 28% of all the squashed instructions in the programs are squashed early. As expected, a wider issue processor reduces this percentage substantially. For an

Program	4-Issue Speedups (%)					8-Issue Speedups (%)				
	Full Guard	Unlimited Mask		21-bit Mask		Full Guard	Unlimited Mask		21-bit Mask	
		All three	GT/GF	All three	GT/GF		All three	GT/GF	All three	GT/GF
Compress	3.5	5.8	5.8	5.8	5.8	4.8	6.0	6.0	6.0	6.0
Elvis	7.7	11.8	11.8	11.8	11.8	8.2	12.6	12.6	12.6	12.5
Eqntott	23.1	23.2	23.2	23.2	23.2	22.6	23.1	23.1	23.1	23.1
Espresso	5.7	12.1	12.1	12.1	12.1	10.4	12.3	12.3	12.3	12.3
Gcc	5.5	9.9	9.5	9.3	8.8	7.2	10.7	10.2	9.7	9.5
Sc	4.2	8.6	8.5	8.5	8.3	6.5	10.8	10.8	10.8	10.7
Wc	34.9	34.9	34.9	34.9	34.9	42.7	42.7	42.7	42.7	42.7
Xlisp	5.4	10.1	10.1	10.1	10.1	7.7	9.3	9.3	9.3	9.3
Average	9.1	13.9	13.8	13.8	13.7	11.2	15.0	14.9	14.9	14.8

Table 6.10: GUARD speedups compared to the speedups obtained using explicit guard conditions, for the counter-based predictor.

Program	4-Issue Speedups (%)					8-Issue Speedups (%)				
	Full Guard	Unlimited Mask		21-bit Mask		Full Guard	Unlimited Mask		21-bit Mask	
		All three	GT/GF	All three	GT/GF		All three	GT/GF	All three	GT/GF
Compress	3.0	5.5	5.5	5.5	5.5	4.2	5.9	5.9	5.9	5.9
Elvis	8.4	11.6	11.6	11.6	11.4	8.0	12.5	12.5	12.5	12.5
Eqntott	10.3	10.5	10.4	10.5	10.3	8.9	8.9	8.9	8.9	8.8
Espresso	5.1	9.1	9.1	9.3	9.3	8.9	9.7	9.7	9.7	9.7
Gcc	5.2	8.9	8.6	8.6	8.5	5.7	10.2	9.7	9.6	9.3
Sc	3.9	7.0	6.9	7.0	6.9	6.1	10.0	9.8	9.8	9.8
Wc	22.2	22.2	22.2	22.2	22.2	25.4	25.4	25.4	25.4	25.4
Xlisp	4.1	7.8	7.8	7.8	7.8	5.8	8.2	8.2	8.2	8.2
Average	6.5	10.1	10.1	10.1	10.0	7.8	11.1	11.0	11.0	10.9

Table 6.11: GUARD speedups compared to the speedups obtained using explicit guard conditions, for the correlation-based predictor.

Program	Instructions	Squashed Instructions	Early Squashes			
			% squashed		% instructions	
			4-issue	8-issue	4-issue	8-issue
Compress	92,395	5,285	36.9	21.8	2.1	1.2
Elvis	20,248	1,338	41.1	12.3	2.7	0.8
Eqntott	1,556,456	92,453	29.3	9.8	1.7	0.6
Espresso	635,907	80,060	19.5	9.3	2.5	1.2
Gcc	92,543	15,898	9.5	6.6	1.6	1.1
Sc	456,879	48,155	26.8	14.4	2.8	1.5
Wc	2,738	521	40.4	9.7	7.7	1.8
Xlisp	258,182	20,680	28.1	1.7	2.3	0.1
Total/Average	3,115,351	264,393	28.1	10.4	2.9	1.0

Table 6.12: Percent of instructions that can be squashed before they are fetched.

8-issue processor, only about 10% of the squashed computation is squashed early.

However, if we compare the number of early squashes with the instruction count of the program, we see that a very small percentage of the instructions are squashed early. Even for the 4-issue processor, less than 3% of all the instructions in most of our benchmarks are squashed early. On the average early squashing accounts for about 3% of instruction count when the program is run on a 4-issue processor; an 8-issue processor reduces this percentage to 1%. These results are not surprising if we consider the relatively low percentage of overhead computation in our guarding regions. From Table 5.3 in Chapter 5) we can see that, on the average, only about 9% of all the instruction in our programs were squashed.

Overall, for our evaluation context, the value of early squashing is limited. Early squashing may be more useful in a context where the amount of squashed computation becomes more significant. For example, aggressive loop-unrolling and software-pipelining can increase both the number of squashed instructions, and the guarding distance, exposing more opportunities for early squashes.

6.3 Impact of misprediction penalty

One of the factors that limits the performance potential of guarding is the low misprediction penalty we used. A larger misprediction penalty, increases the number of stall cycles due to branches. Since the guarded programs execute fewer branches, they experience fewer mispredictions, and therefore will suffer fewer stalls than the original programs. In this section we evaluate the impact of misprediction penalties of 2, 3 and 8 cycles on the performance of guarding, on each of the three guarding methods: full guarding, conditional moves and GUARD instructions. For the GUARD instructions we use the most powerful but still realistic configuration, which allows all three flavors of GUARD instructions with a 21-bit mask.

Table 6.13 lists the execution time speedups for our benchmarks for a 4-issue processor using a counter-based predictor, and for misprediction penalties of 2, 3 and 8 cycles. When guarding is successful in eliminating the branch mispredictions, a larger misprediction penalty will have a smaller

Program	Misprediction Penalty								
	2 cycles			3 cycles			8 cycles		
	Guard	Cmov	GUARD	Guard	Cmov	GUARD	Guard	Cmov	GUARD
Compress	4.1	2.6	6.4	4.6	3.1	6.9	13.8	12.1	16.2
Elvis	7.8	4.2	11.9	7.9	4.2	11.9	13.9	9.9	18.1
Eqntott	31.4	28.8	31.5	39.6	36.1	39.7	84.3	79.7	84.4
Espresso	6.2	3.7	12.5	6.7	3.7	12.9	18.2	14.9	25.1
Gcc	6.3	3.3	10.0	7.1	3.4	10.7	28.5	24.1	32.8
Sc	4.3	2.8	8.5	4.4	2.8	8.5	14.4	12.7	19.0
Wc	42.7	27.5	42.7	50.5	34.5	50.5	89.9	69.6	89.9
Xlisp	5.8	3.8	10.4	6.3	3.9	10.6	28.7	25.9	34.0
Average	12.1	8.6	15.6	13.7	9.9	17.2	31.1	26.9	35.2

Table 6.13: Effect of misprediction penalty on the guarding speedups for a 4-issue processor using a counter-based branch predictor.

effect. For example, from Chapter 5, Table 5.6, we see that guarding eliminates more than 90% of all the misprediction in *Eqntott* and *Wc*, but has a smaller impact on *Elvis* and *Xlisp* and an almost negligible impact on *Sc*. These expectations are confirmed in Table 6.13: for *Sc* for example, increasing the misprediction penalty from 2 to 3 cycles increases the speedup for full guarding from 4.3 to only 4.4. The same increase in the misprediction penalty increases the speedup for *Eqntott*, from 31.4 to 39.6%. The effects of a larger misprediction penalty become more noticeable as the misprediction penalty increases; for an 8-cycle misprediction penalty, *Sc* shows a 14.4% speedup.

On the average, for a 2-cycle misprediction penalty, full guarding gives a speedup of 12% compared with 8.6 and 15.6% for conditional moves and GUARD instructions. For a 3-cycle misprediction penalty, the corresponding speedups are 13.7, 9.9 and 17%, and for an 8-cycle misprediction penalty they are 31, 27 and 35.2%. With the misprediction penalty of 2 or 3 cycles, conditional moves achieve speedups of less than about 4%, for all our benchmarks except *Eqntott* and *Wc*. A misprediction penalty of 8 cycles increases all the speedups to about 10% or more. Comparing full guarding and GUARD instructions, we see that GUARD instructions perform better for all programs except *Eqntott* and *Wc*; for these two programs the performance is about equal.

Table 6.14 shows the effects of misprediction penalty for an 8-issue processor. Since a wider-issue processor generally achieves higher instruction completion rates, the more-or-less fixed overhead caused by mispredictions will become a relatively larger part of the execution time of a program, allowing guarding to achieve better speedups. In Table 6.14 we see that speedups are generally higher. The average speedup for a 2-cycle misprediction penalty is 14.5% for full guarding, 9.6% for conditional moves, and 16.7% for GUARD instructions. For a 3-cycle misprediction penalty is the average speedups rise to 16.1, 11, and 18.4%, and for an 8-cycle misprediction penalty the speedups are 34, 28.4 and 37%.

Table 6.15 lists the execution time speedups for our benchmarks for a 4-issue processor using a correlation-based predictor. Since the correlation-based predictor achieves better prediction accuracies for the original programs, the impact of a larger misprediction penalty will be more even for the original and the guarded versions of the program, and the change in the speedups will be smaller. For a 2-cycle

Program	Misprediction Penalty								
	2 cycles			3 cycles			8 cycles		
	Guard	Cmov	GUARD	Guard	Cmov	GUARD	Guard	Cmov	GUARD
Compress	5.4	2.5	6.6	5.9	3.0	7.1	15.2	12.1	16.5
Elvis	8.3	5.6	12.6	8.4	5.6	12.7	14.5	11.5	19.0
Eqntott	31.0	28.5	31.5	39.3	35.9	39.8	84.2	79.8	85.0
Espresso	10.8	5.9	12.7	11.2	5.9	13.1	23.3	17.4	25.4
Gcc	8.0	2.5	10.5	8.7	2.7	11.2	31.1	23.8	34.0
Sc	6.5	3.1	10.7	6.6	3.1	10.7	17.0	13.1	21.5
Wc	51.5	33.5	51.5	60.4	41.3	60.4	105.1	80.7	105.1
Xlisp	8.1	4.6	9.6	8.4	4.7	9.9	31.6	27.1	33.4
Average	14.5	9.6	16.7	16.1	11.0	18.4	34.2	28.4	36.9

Table 6.14: Effect of misprediction penalty on the guarding speedups for an 8-issue processor using a counter-based branch predictor.

misprediction penalty, full guarding gives speedups of 8.5% compared with the 4.3% and 11.1% for conditional moves and GUARD instructions respectively. For a 3-cycle misprediction penalty, the corresponding speedups are 9.5%, 5.1% and 12.1% and for an 8-cycle misprediction penalty the speedups are 22, 17 and 25%.

The trends are similar for an 8-issue processor using correlation-based predictor. Table 6.16 lists the execution time speedups for our benchmark programs. The average speedups for a 2-cycle misprediction penalty are 9.9, 4.8, and 12.1% for full guarding, conditional moves and GUARD instructions respectively. For a 3-cycle misprediction penalty, the average speedups are 10.9, 5.5, and 13.1% and for an 8-cycle misprediction penalty the speedups are about 24, 18 and 26%.

6.4 Evaluation of guarding on an out-of-order execution model

Guarding is usually used and studied in the context of an in-order issue processor model in which the conditions are always available when guarded instructions are about to be executed. An out-of-order processor runs ahead of the execution of instructions, buffering them in some local storage (a reorder buffer, register update unit, or reservation stations), until their operands are ready. Instructions are executed when they become ready, achieving a dynamic instruction schedule, reducing the importance of a good static instruction schedule, and therefore reducing the importance of guarding. In this section we evaluate the performance of guarding using an realistic and cost-effective out-of-order processor configuration. Our processor can issue up to four instructions per cycle and uses a 32-entry RUU and a 16-entry load/store queue which buffers stores and disambiguates the memory access addresses. We chose this configuration because it is powerful but still effective. Given that our IPCs are all much less than four instructions per cycle, an 8-issue out-of-order processor would have been an overkill. Also, the introduction of a correlation-based (or some other configuration of 2-level adaptive predictor) are a cost-effective way to improve the processors performance. Here we use a correlation-based branch predictor and a misprediction penalty of 3 cycles.

Program	Misprediction Penalty								
	2 cycles			3 cycles			8 cycles		
	Guard	Cmov	GUARD	Guard	Cmov	GUARD	Guard	Cmov	GUARD
Compress	3.5	1.7	6.0	4.0	2.1	6.5	12.2	10.1	14.8
Elvis	8.5	4.3	11.8	8.7	4.3	11.9	13.3	8.8	16.7
Eqntott	13.3	7.0	13.4	16.2	9.3	16.4	33.1	25.2	33.3
Espresso	5.4	2.2	9.6	5.8	2.3	9.9	12.5	8.8	16.9
Gcc	6.2	3.4	9.5	7.1	3.6	10.4	28.4	24.1	32.3
Sc	4.0	1.9	7.0	4.1	1.9	7.1	13.8	11.4	17.0
Wc	25.5	12.1	25.5	28.8	15.1	28.8	45.6	30.0	45.6
Xlisp	4.8	2.9	8.4	5.4	3.3	9.0	24.3	21.9	28.5
Average	8.5	4.3	11.1	9.5	5.1	12.1	21.9	17.0	24.8

Table 6.15: Effect of misprediction penalty on the guarding speedups for a 4-issue processor using a correlation-based branch predictor.

Program	Misprediction Penalty								
	2 cycles			3 cycles			8 cycles		
	Guard	Cmov	GUARD	Guard	Cmov	GUARD	Guard	Cmov	GUARD
Compress	4.7	1.6	6.4	5.2	2.0	6.9	13.5	10.1	15.3
Elvis	8.2	4.8	12.7	8.4	4.8	12.8	13.0	9.3	17.7
Eqntott	11.9	6.9	11.9	14.8	9.2	14.8	31.9	25.4	31.8
Espresso	9.2	3.1	10.0	9.5	3.1	10.3	16.6	9.7	17.4
Gcc	6.7	2.9	10.5	7.7	3.1	11.4	29.7	24.2	34.2
Sc	6.2	3.3	9.8	6.3	3.2	9.8	16.2	12.9	20.1
Wc	29.2	13.7	29.2	32.9	17.0	32.9	51.9	33.8	52.0
Xlisp	6.4	2.8	8.7	7.0	3.3	9.3	26.5	22.1	29.2
Average	9.9	4.8	12.1	10.9	5.5	13.1	23.8	17.8	26.2

Table 6.16: Effect of misprediction penalty on the guarding speedups for a 8-issue processor using a correlation-based branch predictor.

Program	Original Cycles	IPC				Speedup (%)		
		Orig.	Guard	Cmov	GUARD	Guard	Cmov	GUARD
Compress	55,112	1.49	1.57	1.46	1.62	5.4	-1.7	8.3
Elvis	7,931	2.33	2.49	2.40	2.52	6.3	2.8	7.6
Eqntott	632,239	1.84	2.01	1.97	2.03	8.2	6.5	9.3
Espresso	301,673	1.73	1.99	1.85	2.05	13.2	6.6	15.7
Gcc	44,276	1.62	1.67	1.64	1.68	2.7	1.1	3.5
Sc	235,269	1.71	1.82	1.68	1.84	6.0	-1.6	7.0
Wc	862	2.52	2.68	2.63	2.69	5.7	3.9	6.1
Xlisp	140,451	1.63	1.85	1.71	1.89	11.9	4.6	14.1
Total/Average	1,417,813	1.80	1.95	1.85	1.98	7.3	2.7	8.8

Table 6.17: Execution time and IPCs for the original and guarded programs and on a 4-issue, out-of-order issue processor.

Table 6.17 shows the execution time for the original program and the speedups for full guarding, conditional moves and GUARD instructions, for this processor. In this table we see that the dynamic scheduling ability of the processor eliminates much of the potential of guarding. A surprising result is that for *Espresso* and *Xlisp*, the speedups in the out-of-order issue processor are actually larger than the corresponding speedups on an in-order issue processor for all types of guarding. The reason for this behavior for *Espresso* is that our scheduler had exhausted all the registers and could not statically rename many of the registers in the guarded regions. The register renaming of the out-of-order issue eliminated the false dependencies in these guarded regions allowing the guarded code to show larger speedup than in the in-order issue processor. A similar effect takes place for *Xlisp*; due to frequent the function calls, the few registers are used frequently; dynamic register renaming eliminates all the false dependencies, allowing for better speedups.

For full guarding, the speedup for *Eqntott* is reduced by about 40% by the out-of-order abilities of the processor showing only an 8.2% speedup. The most impressive change occurs for *Wc*, for which the speedup for full guarding dropped from 25% to 5.7%. Overall, the average speedups using full guarding is 7.3%.

The impact of out-of-order execution on the performance of conditional moves is even smaller, due to the increased dependence paths through the code. The longer dependence paths result in a small slow-down instead of a speed-up for *Compress* and *Sc*. The average speedup using conditional moves drops to an average of 2.7%.

GUARD instructions also suffer reduced speedups out-of-order issuing ability of the processor. with an average speedup of about 8.8%; the minimum speedup is 3.5% for *Gcc*.

6.4.1 Impact of limited size reference counters

One issue regarding the out-of-order execution of GUARD instructions is the type and extent of hardware support for the out-of-order execution of GUARD instructions. As described in section 4.5, an out-of-order processor has to track the guard masks of all the outstanding GUARD instructions, and should not issue any of the instructions that are marked in these masks until the GUARD instruction is

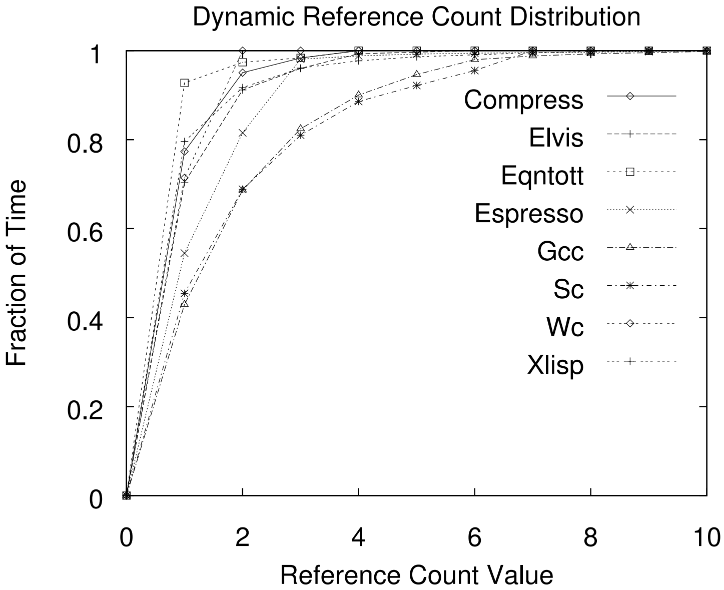


Figure 6.1: Dynamic cumulative distribution of GUARD reference counts for guarded instructions. The X-axis is the reference count and the Y-axis is the ratio of guarded instructions with at most that reference count.

executed. Given limited resources (either a limited size Guard Mask Buffer or a reference counter with fixed counting capacity), the processor must stall when these resources are exhausted. First we address the case where a reference count is attached to each RUU entry.

Figure 6.1 plots the distribution of reference counts for the guarded instructions in the program. To generate this plot, we computed the static GUARD reference count for each instruction in the program, i.e., for each instruction we counted how many GUARD instructions have it marked in their guard mask; we then incremented the corresponding reference-count bin by the execution frequency of the instruction. Note, that this scenario corresponds to the worst case, where all the GUARD instructions for a particular instruction are outstanding. The plot in Figure 6.1 shows that the GUARD reference count did not exceed 7 for 99% of all the guarded instructions. Therefore a 3-bit counter per instruction would be sufficient to allow the processor to continue issuing instructions most of the time, even with the pessimistic assumption that all GUARD instructions are outstanding. In reality however, the processor will have time to execute some of these GUARD instructions, and the run-time maximum reference-count values will be smaller. Here we should point out that since an overflow in a reference count will cause the processor to stall issuing *all* other instructions, it is important to design for the worst case rather than for the average. For example, a 2-bit reference counter will perform adequately for most benchmarks, but may overflow for up to 10% of the guarded instructions in *Gcc* and *Sc*.

6.4.2 Impact of a limited size Guard Mask Buffer

Next, we consider the effects of the size of the Guard Mask Buffer (GMB) on the performance of an out-of-order issue processor. To evaluate this impact, we first explore the upper bound using conservative assumptions, and then we perform simulation of five GMB sizes and measure the actual impact on the

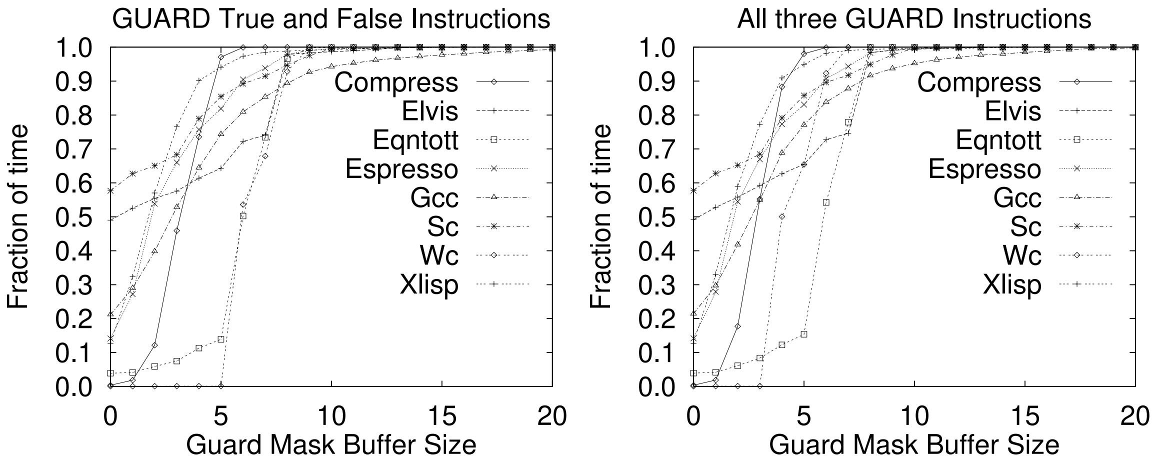


Figure 6.2: Cumulative distribution of time versus Guard Mask Buffer occupancy.

execution time of our benchmarks.

To get an upper bound on the impact of the GMB size on the execution time, we profiled the GMB occupancy during the execution of a program, assuming a perfect instruction fetch mechanism (which would keep all the 32 entries in the RUU filled with instructions), and an in-order completion of all instructions. This scenario corresponds to the worst case, where all the GUARD instructions in the processor are outstanding. Figure 6.2 plots the cumulative fraction of the execution time versus the size of GUARD instructions in the system. The left graph corresponds to an instructions set supporting only the GUARDTRUE and GUARDFALSE instructions, while the right plot corresponds to an instructions set supporting the GUARDBOTH instruction as well. In both graphs we see that the majority of the cycles are spent with less than 8 GUARD instructions outstanding. Comparing the two graphs we see that the introduction of the GUARDBOTH instruction affects only a few programs noticeably, which is expected considering the relatively small difference between the GUARD overheads with and without the GUARDBOTH instruction in Table 5.16, Chapter 5.

The plots in Figure 6.2 are pessimistic for two reasons: (i) the RUU of a real processor will not be full all the time, and (ii) some of the GUARD instructions in the RUU will find their operand available, and will be executed, releasing the corresponding entry in the GMB. To measure the real impact of the GMB size on the performance, we simulated a GMB of 4, 6, 8, 12 and 16 entries. Table 6.18 lists the corresponding slow-downs as a percentage of the total execution time of the program. For an instruction set supporting GUARDTRUE and GUARDFALSE instructions only, we see that a 4 entry buffer is not large enough to hold the number of outstanding GUARD masks, and that the GMB overflows have a significant impact on the performance of most benchmarks. A 6-entry GMB is large enough to handle the outstanding GUARD instructions, showing small slow-downs of less than 2% for most of our benchmarks; however, the slowdown for *Gcc* is 7%, which would negate the 3.5% speedup shown in Table 6.17. The 8 entry GMB shows less than 0.7% slow-downs for most programs, but still shows a 3.8% slowdown for *Gcc*. A 12 entry GMB gives almost never overflows for most programs and shows only a 1.5% slowdown for *Gcc*, for a net speedup of about 2%. Increasing the GMB size to 16 entries reduces the slowdown for *Gcc* to 0.3%.

A GUARDBOTH instruction can have a significant impact on the performance, but only for the

Program	GUARD TRUE/FALSE					GUARD TRUE/FALSE/BOTH				
	4	6	8	12	16	4	6	8	12	16
Compress	0.41	0.00	0.00	0.00	0.00	0.41	0.00	0.00	0.00	0.00
Elvis	23.84	1.93	0.74	0.00	0.00	23.56	1.36	0.32	0.00	0.00
Eqntott	39.03	0.24	0.00	0.00	0.00	34.48	0.04	0.00	0.00	0.00
Espresso	7.33	1.26	0.27	0.06	0.00	6.84	1.18	0.24	0.02	0.00
Gcc	16.65	7.24	3.87	1.49	0.03	13.96	6.17	2.99	0.71	0.02
Sc	9.46	1.39	0.27	0.03	0.00	9.14	1.17	0.10	0.00	0.00
Wc	53.52	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
Xlisp	2.80	0.89	0.35	0.04	0.00	1.79	0.58	0.25	0.04	0.00

Table 6.18: Impact of the Guard Mask Buffer size on the execution time (percent slow-down) for Guard Mask Buffers of 4, 6, 8, 12 and 16 entries, and for instruction sets supporting the GUARDTRUE and GUARDFALSE instructions only, or the GUARDBOTH instruction as well.

smaller GMBs. For *Wc* for example, the slowdown for a 4 entry GMB drops from 53% to 0.0% when the GUARDBOTH instruction is introduced, since the number of GUARD instructions is reduced, requiring fewer GMB entries. However, as soon as the buffer becomes sufficiently large, the difference diminishes. For a GMB with 12 entries, the introduction of GUARDBOTH instructions has a significant impact only on *Gcc*, reducing the slowdown from 1.5% to 0.7%.

The size of the Guard Mask Buffer is related to the number of RUU entries; if a processor implements a larger RUU, the number of the GMB entries should be increased accordingly, to ensure that the GMB will overflow only infrequently (if at all) and allow the processor to keep issuing instructions. The interpretation of the results in Table 6.18 should be that the GMB should have between one fourth and one half of the total entries of the RUU to minimize the issue stalls.

6.5 Summary

In this chapter we examined the impact of guarding on the execution time of our benchmarks. For our evaluation context, we found that this impact varies widely with the benchmark. Two of our benchmarks, namely *Eqntott* and *Wc*, have a very simple control and computation structure. All forms of guarding (full, conditional moves or GUARD instructions) were able if-convert these two programs effectively, and show significant improvements in the execution time. For the rest of the programs however, the impact of guarding is smaller.

For an 8-issue in-order processor using counter-based branch and a misprediction penalty of one cycle, full guarding achieves an average speedup of 11%. For the same configuration, conditional moves achieve an average speedup of 7.1%, and GUARD instructions achieve a speedup of 14.8%. A misprediction penalty of three cycles boosts these speedups to 16%, 11% and 18.4% respectively, while a misprediction penalty of eight brings them to 34, 28 and 37%.

On first glance, conditional moves appear to be a viable way to achieve most of the benefits of guarding. However, if we exclude *Eqntott* and *Wc*, the average speedup of conditional moves drops to less than 4%, even for a misprediction penalty of 3 cycles.

A better branch predictor reduces the potential of guarding. When we used the correlation based predictor, the speedups for full guarding, conditional moves and GUARD instructions dropped to 7.8%, 3.5 and about 11% respectively. A misprediction penalty of three cycles restores some of the potential, increasing these speedups to 10%, 5.5% and 13% respectively. A misprediction penalty of eight cycles increases the speedups to 23, 18, and 26%.

Finally, an out-of-order execution processor limits the potential of guarding. For a 4-issue processor the average speedups for full guarding, conditional moves and GUARD instructions are 7.3, 2.7 and 8.8% respectively, while for *Compress* and *Sc* conditional moves gave negative speedups (i.e., slowdown). However, the better out-of-order speedup of GUARD instructions come with a cost: to allow the out-of-order execution of GUARD instructions, the processor has to either implement a 3-bit reference count per RUU entry. If a processor implements a Guard Mask Buffer, it should have about one third of the number of RUU entries in order to keep the amount of issue stalls small.

Chapter 7
Conclusions

The commercial acceptance of guarded execution largely depends both on its incorporation in the existing instruction sets, and on its potential to improve the execution time of general program workloads. In this thesis we have defined the GUARD instructions, that allow the incremental extension of existing instruction sets to include guarded execution. The main idea of GUARD instructions is to use a forward specification scheme, where the processor is informed about the status of instructions later in the dynamic instruction stream. Guarding using GUARD instructions performs better than ordinary guarding using explicit operands for two reasons: (i) GUARD instructions allow more efficient guard condition evaluation, reducing the instruction count in many cases and (ii) they allow the processor to squash computation before it even enters in the pipeline.

In addition, GUARD instructions do not require the additional register file read port that ordinary guarding requires; the rest of the hardware required to implement GUARD instructions in an in-order processor is simple: a shift-register for the scalar mask register, and a simple 3-level logic circuit to evaluate the condition and update the scalar mask register. The implementation of GUARD instructions in an out-of-order issue processor requires the addition of either a Scalar Mask Buffer, or of a reference count per RUU entry, to keep track of the dependencies through the guard masks.

We evaluated the performance of GUARD and we compared it to the performance of full guarding and that of using conditional move instructions to synthesize guarded execution. Our evaluation was performed in what we believe is an every-day environment; in particular, we did not assume that profiling information was available. We found that for in-order processor using counter-based branch prediction, the execution time can be reduced by up to 16% for full guarding, and 11% for conditional moves; a better predictor reduced these speedups to 11% and 5.5%, respectively. GUARD instructions were able to surpass the performance of full guarding showing a improvement of 18.4% for a simple counter based predictor and 13% for the correlation predictor. We also found that out-of-order execution reduced the guarding potential to 7.3% for full guarding and 2.7% conditional moves, while GUARD instructions sustained a speedup of 8.8%.

Our results indicate that even in a conservative environment, guarded execution can improve the performance of ordinary, integer intensive programs; if the programs have a regular structure, the potential of guarded execution increases dramatically.

However, guarding comes at some implementation cost. For in-order processors, the complexity is concentrated in the register file read ports, and the forwarding control logic and paths. For an out-of-order processor, the RUU or reservation station entries have to be wider. The implementation of GUARD instructions in an out-of-order processor also introduce additional complexity, since the processor should include a Guard Mask Buffer and a Scalar Mask Rename Buffer to allow the issue mechanism to issue GUARD and branch instructions without stalling. In this thesis we merely outlined the hardware cost in a high level pipeline design. To determine whether the hardware cost of guarding is justified by the performance improvement is provides, one should perform a more detailed cost analysis.

7.1 Comparison with the IMPACT work

The results presented in this thesis show that guarding does improve the performance but by a relatively small amount. The IMPACT group publications show guarding to provide speedups over a base case in the range of 3 to 8 [HMG^{+}95]. Much of the difference can be attributed to different assumptions made in our and their work about the base case, the execution model and the compilation environment.

First, our processor models are radically different: in this work we used processors that support speculative execution of instructions; it is unclear whether IMPACT's base case allows speculation. We assumed that the memory is multi-ported by replication, which limits the number of stores that can be executed per cycle to just one; the IMPACT work on the other hand assumes a memory system that allows any mix of N (where N is the issue width of the processor) loads and stores (presumable independent) can be executed per cycle. The IMPACT work also uses a larger (sometimes infinite) number of registers. In addition, the IMPACT compiler uses profile information to guide the of the control flow graph transformations described in Chapter 3. The profiling information allows the compiler to determine the best transformation to use in each case. However, it is unclear how these transformations would perform in the absence of the profiling information.

Another reason for the difference in performance is that the IMPACT work assumes a very simple branch handling mechanism. They use a counter based predictor and they usually allow only a single branch to be executed per cycle. Predicting and traversing multiple branches per cycle is indeed complicated, but if the branches are predicted one at a time, executing more than one branch per cycle (that is, verifying the corresponding predictions) is not very hard. Other research in VLIW embraces the execution of multiple branches per cycle [KSR94, SK95]. For example, the PlayDoh instruction set allows the execution of multiple branches per cycle, under the constraint that the compiler must guarantee that at most one of them can be taken. When the execution of multiple branches per cycle is coupled with an accurate dynamic branch predictor, the performance of the base case will improve; the performance of the guarded code may also improve but by a smaller amount, since the base case is much more branch limited than the guarded case. Thus, a better branch handling mechanism will reduce the benefits of guarding.

However, the biggest difference can be attributed to the execution model. The IMPACT work assumes a VLIW type architecture, where a good compiler schedule is crucial to the performance. To achieve a good schedule, the IMPACT compiler uses software pipelining, unrolling and other profile-based transformations of the control flow graph of the programs. Our in-order processor model is similar but uses speculative execution and out-of-order completion of instructions to overcome some of the scheduling limitations. Also, as our results indicate, an out-of-order processor with a reasonable number of RUU (or reservation station) entries, coupled with a fairly accurate (2-level adaptive) dynamic branch predictor will dynamically unwind the code and expose the instruction level parallelism to the execution engine. A good run-time schedule will improve the performance of the original program, depleting in this way the performance potential of guarding.

7.2 Future work

Although we have addressed some of the key properties of GUARD instructions, there are several ways the work in this thesis can be extended. First, we used a scheduler to do the guarding region selection, if-conversion and scheduling, and we used the same guarding regions both for full guarding and GUARD instructions. An integrated compiler has the potential to improve the quality of the generated code. For

GUARD instructions, the compiler could customize the guarding region selection to explore the implicit AND-ing property of the scalar mask; it could also attempt to schedule the GUARD instructions, so as to fit all the guarded instructions in the guard mask of a single GUARD instruction, reducing the number of the required GUARDUPPER instructions.

Another feature of GUARD instructions that was not exploited in this work, is the early-out feature. For our guarding regions we found that early-out is not very frequent, because the guarding distances are relatively short. If the code structure is regular, the compiler can use aggressive unrolling, which would increase the guarding distances, and rely on the early squashing abilities of the processor to filter out the squashed computation.

The out-of-order execution of GUARD instructions requires hardware to detects when all the guard dependencies of each instruction have been resolved. Since the guard dependencies are converted control dependencies, the same hardware can be extended to allow the on-the-fly conversion of code with branches into branchless code inside the RUU. Furthermore, the dependency resolution hardware opens up the possibility for *speculative* execution of guarded instructions, in which the guarded instruction can be speculative executed immediately, and verified later, when the guard condition becomes available. This approach would be similar to the Predicate State Buffering [ANHN95], but it would be a micro-architecture feature, rather than being explicit in the instruction set architecture as in the case of Predicate State Buffering.

Bibliography

[AKPW83] J. R. Allen, Ken Kennedy, Carrie Porterfield, and Joe Warren. Conversion of Control Dependence to Data Dependence. In *Proceedings 10th Annual ACM Symposium on Principles of Programming Languages*, pages 177–189, January 1983.

[ANHN95] Hideki Ando, Chikako Nakanishi, Tetsuya Hara, and Masao Nakaya. Unconstrained Speculative Execution with Predicated State Buffering. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 126–137, Santa Margherita Ligure, Italy, June 22–24, 1995.

[APS95] Todd M. Austin, Dionisios N. Pnevmatikatos, and Gurindar S. Sohi. Streamlining Data Cache Access with Fast Address Calculation. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 369–380, Santa Margherita Ligure, Italy, June 22–24, 1995.

[BL93] Tomas Ball and James R. Larus. Branch Prediction For Free. In *Proceedings of the SIGPLAN 1993 Conference on Programming Language Design and Implementation*, pages 300–313, April 20–23, 1993.

[BL94] Thomas Ball and James R. Larus. Optimally Profiling and Tracing Programs. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(4):1319–1360, July 1994.

[BRRP82] A C. D. Glaeser B. R. Rau and R. L. Picard. Efficient Code Generation For Horizontal Architectures: Compiler Techniques and Architectural Support. In *Proceedings of the 9th Annual Symposium on Computer Architecture*, pages 131–139, April 1982.

[Cas93] Brian Case. SPARC V9 Adds Wealth of New Features. In *Microprocessor Report*, volume 7, February 1993.

[CHPC95] Po-Yung Chang, Eric Hao, Yale N. Patt, and Pohua P. Chang. Using Predicated Execution to Improve the Performance of a Dynamically Scheduled Machine with Speculative Execution. In Lubomir Bic, Wim Böhm, Paraskevas Evripidou, and Jean-Luc Gaudiot, editors, *Proceedings of the IFIP WG 10.3 Working Conference on Parallel Architectures and Compilation Techniques, PACT '95*, pages 99–108, Limassol, Cyprus, June 27–29, 1995. ACM Press.

[CMC+91] Pohua P. Chang, Scott A. Mahlke, William Y. Chen, Nancy J. Warter, and Wen-Mei W. Hwu. IMPACT: An Architectural Framework for Multiple-Instruction-Issue Processors. In *Proceedings of the 18th Annual International Symposium on Computer Architecture*, pages 266–275, Toronto, Ontario, May 27–30, 1991.

[CMMP95] Thomas M. Conte, Kishore N. Menezes, Patrick M. Mills, and Burzin A. Patel. Optimization of Instruction Fetch Mechanisms for High Issue Rates. In *Proceedings of the*

22nd Annual International Symposium on Computer Architecture, pages 333–344, Santa Margherita Ligure, Italy, June 22–24, 1995.

[CNO+88] R. P. Colwell, R. P. Nix, J. J. O'Donnell, D. B. Papworth, and P. K. Rodman. A VLIW Architecture for a Trace Scheduling Compiler. *IEEE Transactions on Computers*, 37(8):967–979, August 1988.

[Com88] R. Comerford. How DEC Developed Alpha. *IEEE Spectrum*, 29(7):43–47, July 1988.

[Cor94] International Business Machines Corporation. *The PowerPC Architecture: Specifications for a New family of RISC Processors*. Morgan Kaufmann Publishers, San Fransisco, 1994.

[DHB89] James C. Dehnert, Peter Y.-T. Hsu, and Joseph P. Bratt. Overlapped Loop Support in the Cydra 5. In *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 26–38, Boston, Massachusetts, April 3–6, 1989.

[Dij75] E. W. Dijkstra. Guarded Commands, Nondeterminacy and Formal Derivation of Programs. *Communications of the ACM*, 18:453–457, August 1975.

[Ebc87] Kemal Ebcioglu. A Compilation Technique for Software Pipelining of Loops with Conditional Jumps. In *Proceedings of the 20th Annual Workshop on Microprogramming (Micro 20)*, pages 69–79, December 1–4, 1987.

[Ebc88] Kemal Ebcioglu. Some Design Ideas for a VLIW Architecture for Sequential Natured Software. In Cosnard *et al.*, editor, *Parallel Processing (Proceedings of IFIP WG 10.3 Working Conference on Parallel Processing)*, pages 3–21. North Holland, April 1988.

[FF92] Joseph A. Fisher and Stefan M. Freudenberger. Predicting Conditional Branch Directions From Previous Runs of a Program. In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 85–95, Boston, Massachusetts, October 12–15, 1992.

[Fis81] Joseph A. Fisher. Trace Scheduling: A Technique for Global Microcode Compaction. *IEEE Transactions on Computers*, C-30(7), July 1981.

[FOW87] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The Program Dependence Graph and Its Use in Optimization. *ACM Transactions on Programming Languages and Systems*, pages 319–349, July 1987.

[Fra93] Manoj Franklin. *The Multiscalar Architecture*. PhD thesis, University of Wisconsin–Madison, November 1993.

[FS92] Manoj Franklin and Gurindar S. Sohi. The Expandable Split Window Paradigm for Exploiting Fine-Grained Parallelism. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 58–67, Gold Coast, Australia, May 19–21, 1992.

[Gwe95] Linley Gwennap. Intel's P6 Uses Decoupled Superscalar Design. *Microprocessor Report*, 9(2):9–15, February 1995.

[HCP94] Eric Hao, Po-Yung Chang, and Yale N. Patt. The Effect of Speculatively Updating Branch History on Branch Prediction Accuracy, Revisited. In *Proceedings of the 27th Annual International Symposium on Microarchitecture*, pages 228–232, San Jose, California, November 30–December 2, 1994.

[HD86] Peter Y. T. Hsu and Edward S. Davidson. Highly Concurrent Scalar Processing. In *Proceedings of the 13th Annual International Symposium on Computer Architecture*, pages 386–395, Tokyo, Japan, June 2–5, 1986.

[HMG+95] Richard E. Hank, Scott A. Mahlke, John C. Gyllenhaal, David I. August, and Wen-Mei W. Hwu. Control Flow Optimization with Predicated Execution in Future Microprocessors. In *Proceedings of the IEEE*, December 1995.

[HP87] Wen-Mei W. Hwu and Yale N. Patt. Checkpoint Repair for High-Performance Out-of-Order Execution Machines. *IEEE Transactions on Computers*, C-36(12):1496–1514, December 1987.

[Hsu86] Peter Y. T. Hsu. *Highly Concurrent Scalar Processing*. PhD thesis, University of Illinois at Urbana-Champaign, January 1986.

[HT72] R. G. Hintz and D. P. Tate. Control Data STAR-100 Processor Design. In *COMCON, IEEE*, pages 1–4, September 1972.

[Joh90] M. Johnson. *Superscalar Microprocessor Design*. Prentice Hall, Englewood Cliffs, New Jersey, 1990.

[Kan87] Gerry Kane. *MIPS R2000 RISC Architecture*. Prentice Hall, Englewood Cliffs, New Jersey, 1987.

[KSR94] Vinod Kathail, Michael Schlansker, and B. Ramakrishna Rau. HPL PlayDoh Architecture Specification: Version 1.0. Technical Report HPL-93-80, Hewlett Packard Laboratories, Palo Alto, CA, February 1994.

[Kuc78] D. J. Kuck. *The Structure of Computers and Computations*. John Wiley and Sons, 1978.

[Lam88] Monica S. Lam. Software Pipelining: An Effective Scheduling Technique for VLIW Machines. In *Proceedings of the SIGPLAN 1988 Conference on Programming Language Design and Implementation*, pages 318–328, June 1988.

[Lar93] James R. Larus. Efficient Program Tracing. *IEEE Computer*, 26(5):52–61, May 1993.

[LFK+93] P. G. Lowney, S. M. Freudenberger, T. J. Karzes, W. D. Lichtenstein, R. P. Nix, J. S. O'Donnel, and J. C. Ruttenberg. The Multiflow Trace Scheduling Compiler. *The Journal of Supercomputing*, 4(11):376–408, November 1993.

[Lin92] David Chu Lin. Compiler Support for Predicate Execution in Superscalar Processors. Master's thesis, University of Illinois at Urbana-Champaign, 1992.

[LS84] Johnny K. F. Lee and Alan Jay Smith. Branch Prediction Strategies and Branch Target Buffer Design. *IEEE Computer*, 17(1):6–22, January 1984.

[LW92] Monica S. Lam and Robert P. Wilson. Limits of Control Flow on Parallelism. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 46–57, Gold Coast, Australia, May 19–21, 1992.

[Mac93] Advanced RISC Machines. *ARM 610 RISC Processor*, January 1993.

[MCH+92] Scott A. Mahlke, William Y. Chen, Wen-Mei W. Hwu, B. Ramakrishna Rau, and Michael S. Schlansker. Sentinel Scheduling for VLIW and Superscalar Processors. In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 238–247, Boston, Massachusetts, October 12–15, 1992.

[MHB+94] Scott A. Mahlke, Richard E. Hank, Roger A. Bringmann, John C. Gyllenhaal, David M. Gallagher, and Wen-Mei W. Hwu. Characterizing the Impact of Predicated Execution on Branch Prediction. In *Proceedings of the 27th Annual International Symposium on Microarchitecture*, pages 217–227, San Jose, California, November 30–December 2, 1994.

[MHM+95] Scott A. Mahlke, Richard E. Hank, James E. McCormick, David I. August, and Wen-Mei W. Hwu. A Comparison of Full and Partial Predicated Execution Support for ILP Processors. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 138–149, Santa Margherita Ligure, Italy, June 22–24, 1995.

[MIP90] MIPS Computer Systems, Inc. *UMIPS-V Reference Manual (pixie and pixstats)*. Sunnyvale, California, 1990.

[MLC+92] Scott A. Mahlke, David C. Lin, William Y. Chen, Richard E. Hank, and Roger A. Bringmann. Effective Compiler Support for Predicated Execution Using the Hyperblock. In *Proceedings of the 25th Annual International Symposium on Microarchitecture*, pages 45–54, Portland, Oregon, December 1–4, 1992.

[Nic85] Alexandru Nicolau. Percolation Scheduling: A Parallel Compilation Technique. Technical Report TR 85-678, Department of Computer Science, Cornell University, May 1985.

[PFS93] Dionisios N. Pnevmatikatos, Manoj Franklin, and Gurindar S. Sohi. Control Flow Prediction for Dynamic ILP Processors. In *Proceedings of the 26th Annual International Symposium on Microarchitecture*, pages 153–163, Austin, Texas, December 1993.

[Pri94] Charles Price. *MIPS IV ISA Manual*, October 1994.

[PS91] Joseph C. H. Park and Mike Schlansker. On Predicated Execution. Technical Report HPL-91-58, Hewlett Packard Laboratories, Palo Alto, CA, May 1991.

[PS94] Dionisios N. Pnevmatikatos and Gurindar S. Sohi. Guarded Execution and Branch Prediction in Dynamic ILP Processors. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 120–129, Chicago, Illinois, April 18–21, 1994.

[PSR92] Shien-Tai Pan, Kimming So, and Joseph T. Rahmeh. Improving the Accuracy of Dynamic Branch Prediction Using Branch Correlation. In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 76–84, Boston, Massachusetts, October 12–15, 1992.

[Rau94] B. Ramakrishna Rau. Iterative Modulo Scheduling: An Algorithm For Software Pipelining Loops. In *Proceedings of the 27th Annual International Symposium on Microarchitecture*, pages 63–74, San Jose, California, November 30–December 2, 1994.

[RG91] B. R. Rau and C. D. Glaeser. Some Scheduling techniques and an Easily Schedulable Horizontal Architecture for High Performance Scientific Computing. In *Proceedings of the Fourteenth Annual Workshop on Microprogramming*, pages 183–198, October 1991.

[RL92] Anne Rogers and Kai Li. Software Support for Speculative Loads. In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 38–50, Boston, Massachusetts, October 12–15, 1992.

[Rus78] R. M. Russel. The CRAY-1 Computer System. *Communications of the ACM*, 21(1):63–72, January 1978.

[RYYT89] B. Ramakrishna Rau, David W. L. Yen, Wei Yen, and Ross A. Towle. The Cydra 5 Departmental Supercomputer: Design Philosophies, Decisions, and Trade-offs. *IEEE Computer*, 22(1):12–35, January 1989.

[SHL92] Michael D. Smith, Mark Horowitz, and Monica S. Lam. Efficient Superscalar Performance Through Boosting. In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 248–259, Boston, Massachusetts, October 12–15, 1992.

[SK95] Michael Schlansker and Vinod Kathail. Critical Path Reduction for Scalar Programs. In *Proceedings of the 28th Annual International Symposium on Microarchitecture*, pages 57–69, Ann Arbor, Michigan, November 29–December 1, 1995.

[SKA94] Michael Schlansker, Vinod Kathail, and Sadun Anik. Height Reduction of Control Recurrences for ILP Processors. In *Proceedings of the 27th Annual International Symposium on Microarchitecture*, pages 40–51, San Jose, California, November 30–December 2, 1994.

[SLH90] Michael D. Smith, Monica S. Lam, and Mark A. Horowitz. Boosting Beyond Static Scheduling in a Superscalar Processor. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 344–354, Seattle, Washington, May 28–31, 1990.

[Smi81] James E. Smith. A Study of Branch Prediction Strategies. In *Proceedings of the 8th International Symposium on Computer Architecture*, pages 135–148, Minneapolis, Minnesota, May 12–14, 1981.

[Smi92] Michael D. Smith. *Support for Speculative Execution in High-Performance Processors*. PhD thesis, Stanford University, November 1992.

[Soh90] Gurindar S. Sohi. Instruction Issue Logic for High-Performance, Interruptible, Multiple Functional Unit, Pipelined Processors. *IEEE Transactions on Computers*, 39(3):349–359, March 1990.

[SPE91] SPEC newsletter, December 1991.

[SV87] Gurindar S. Sohi and Sriram Vajapeyam. Instruction Issue Logic for High-Performance, Interruptible Pipelined Processors. In *Proceedings of the 14th Annual International Symposium on Computer Architecture*, pages 27–34, Pittsburgh, Pennsylvania, June 2–5, 1987.

[TLS90] P. Tirumalai, M Lee, and M Schlansker. Parallelization of Loops With Exits on Pipelined architectures. In *Proceedings of Supercomputing '90*, November 1990.

[Tom67] R. M. Tomasulo. An Efficient Algorithm for Exploiting Multiple Arithmetic Units. *IBM Journal*, pages 25–33, January 1967.

[TYS+94a] Adam R. Talcott, Wayne Yamamoto, Mauricio J. Serrano, Roger C. Wood, and Mario Nemirovsky. The Impact of Unresolved Branches on Branch Prediction Scheme Performance. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 12–21, Chicago, Illinois, April 18–21, 1994.

[Tys94b] Gary Scott Tyson. The Effects of Predicated Execution on Branch Prediction. In *Proceedings of the 27th Annual International Symposium on Microarchitecture*, pages 196–206, San Jose, California, November 30–December 2, 1994.

[Wat72] W. J. Watson. The TI-ASC – A Highly Modular and Flexible Super Computer Architecture. In *Proceedings of the AFIPS Fall Joint Computer Conference*, pages 221–228, 1972.

[WMHR93] Nancy J. Warter, Scott A. Mahlke, Wen-Mei W. Hwu, and B. Ramakrishna Rau. Reverse If-Conversion. In *Proceedings of the SIGPLAN 1993 Conference on Programming Language Design and Implementation*, pages 290–299, 1993.

[YGS95] Cliff Young, Nicholas Gloy, and Michael D. Smith. A Comparative Analysis of Schemes for Correlated Branch Prediction. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 276–286, Santa Margherita Ligure, Italy, June 22–24, 1995.

[YMP93] Tse-Yu Yeh, Deborah T. Marr, and Yale N. Patt. Increasing the Instruction Fetch Rate via Multiple Branch Prediction and a Branch Address Cache. In *Proceedings of ICS-7*, July 1993.

[YP92] Tse-Yu Yeh and Yale N. Patt. Alternative Implementations of Two-Level Adaptive Branch Prediction. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 124–134, Gold Coast, Australia, May 19–21, 1992.

[YP93] Tse-Yu Yeh and Yale 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*, pages 257–266, San Diego, California, May 17–19, 1993.