Guarded Execution and Branch Prediction in Dynamic ILP Processors[†]

Dionisios N. Pnevmatikatos and Gurindar S. Sohi

Computer Sciences Department,
University of Wisconsin-Madison,
1210 West Dayton Street,
Madison, WI 53706
FAX (608) 262-9777

Contact Address

Dionisios N. Pnevmatikatos
Computer Sciences Department,
University of Wisconsin-Madison,
1210 West Dayton Street,
Madison, WI 53706
FAX (608) 262-9777
pnevmati@cs.wisc.edu

[†] This work was supported in part by NSF grant CCR-9303030 and by ONR grant N00014-93-1-0465.

Guarded Execution and Branch Prediction in Dynamic ILP Processors

Dionisios N. Pnevmatikatos and Gurindar S. Sohi

Computer Sciences Department,
University of Wisconsin-Madison,
1210 West Dayton Street,
Madison, WI 53706

Abstract

In this paper we evaluate the effects of guarded (or conditional, or predicated) execution on the performance of an instruction level parallel processor employing dynamic branch prediction. First, we assess the utility of guarded execution, both qualitatively and quantitatively, using a variety of application programs. Our assessment shows that guarded execution significantly increases the opportunities for both a compiler, and dynamic hardware, to extract and exploit parallelism. However, existing methods of specifying guarded execution have several drawbacks that limit its use. Second, we study the interaction of guarding and dynamic branch prediction. No clear trends emerge regarding the ability of guarding to uniformly eliminate branches with poor predictability. In some cases guarding eliminates branches with a poor prediction accuracy, in other cases it eliminates branches with good predictability. However, the use of guarding results in a significant increase in the dynamic window size (instructions between mispredicted branches). Third, we present a new method of specifying guarded execution. The proposed method uses special GUARD instructions, which can be used to incorporate guarded execution into existing instruction sets. GUARD instructions realize the full power of guarded execution, without the drawbacks of existing methods of specifying guarded execution.

1. Introduction

Many recent microprocessors rely heavily on instruction-level parallelism (ILP) to achieve high performance levels. Most of these processors employ dynamic parallelism detection and extraction techniques, in which the hardware has to examine a (small) number of instructions, determine if their operands are available and when they are available, issue the instructions to the available execution units.

Many studies have shown that the parallelism available within basic blocks is limited [2, 3, 21, 23], and it is clear that one has to look beyond basic block boundaries for more parallelism. The larger the number of instructions that can be examined, the greater the parallelism that can be extracted.

To get past branch instructions, processor designers have two options. The first option is the use of *speculative execution*. In this approach, the outcome of a branch instruction is predicted and instructions from the predicted path are examined for execution. This technique requires a branch prediction mechanism, which can be either static or dynamic, and the ability to undo the effects of instructions executed after an incorrectly predicted branch. The second option is the use of *guarded execution* [13] (also called *conditional execution* or *predicated* execution). By eliminating some branch instructions, the effective block size (the number of instructions between branches) is increased, increasing opportunities for parallelism extraction.

Traditionally, guarded execution and speculative execution (especially speculative execution with dynamic branch prediction), have been treated mutually exclusively. Furthermore, there have been very few studies of the utility of guarded execution for general-purpose application programs [18]. Recent microprocessors, however, have combined the two, offering simple guarded instructions such as the *conditional move* instruction, while allowing for dynamic branch prediction [4, 6]. The presence of both guarded and speculative execution opens up many opportunities (both static and dynamic) to exploit ILP. It also raises several questions regarding the interactions of the two techniques.

This paper has three purposes. The first purpose is to assess the pros and cons of guarded execution, both qualitatively and quantitatively, for a variety of application programs. One result of this assessment is that while guarding has tremendous potential, existing methods of specifying guarded execution have major drawbacks that limit the use of guarded execution. The second purpose is to examine the interaction of how guarding and dynamic branch prediction interact, and how guarding impacts dynamic branch prediction. The third purpose is to present a new way of specifying guarded execution which eliminates almost all the drawbacks of existing methods.

We present a qualitative discussion of guarded execution in section 2. In section 3, we present the evaluation methodology, the benchmarks, and the metrics that we use to quantitatively assess guarding. Section 4 presents quantitative results assessing the ability of guarded execution to enhance opportunities for parallelism extraction. It also assesses the potential overhead of guarded execution. Section 5 evaluates the interplay between guarded execution and dynamic branch prediction. In section 6, we present a new method of specifying guarded execution. Our proposed technique allows guarded execution, in its full form, to be integrated easily into existing instruction sets. We discuss how the proposed technique overcomes the drawbacks of existing methods of specifying guarded execution, and also evaluate it quantitatively. Finally, section 7 presents concluding remarks.

2. Guarded Execution

Guarded execution (or simply *guarding*), for scalar processors was first proposed by Hsu and Davidson [13, 14] 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 decision tree. These diverging control structures are not amenable to if-conversion [1], and guarding is used as a general and powerful technique to fill multiple delay slots of branch instructions. Hsu and Davidson assume that the instruction set supports guarded stores and guarded jumps and allow for guard conditions that are the conjunction or disjunction of two operands in either true or complementary form. Computation instructions are not needed in a guarded form because a single assignment property is maintained in the code generated for the decision tree.

The control structures of many common programs, however, are better represented by a DAG. For these structures, if-conversion is more appropriate. If-conversion converts control dependencies to data dependencies: a branch instruction and the instructions that are control dependent on it are replaced with an instruction that sets a condition (if it is not already available in a register), and a sequence of instructions guarded by this condition. The semantics of a guarded instruction are as follows: evaluate the guard condition; if it is met, then execute the instruction, otherwise treat the instruction as a NOP.

Vector processors have long benefited from guarded execution. Here vector masks are used to express (multiple) guard conditions. Using these vector masks, loops with if-statements can be vectorized. Recently proposed VLIW machines, for example the Cydra-5 [20], and the IBM VLIW machine [9], have also used guarded execution to facilitate the software pipelining of loops with conditional branch instructions [7, 16].

To incorporate guarded execution into a scalar instruction set, we need to be able to specify a guard condition for each (guarded) instruction. Proposed methods for specifying guarded execution suggest the use of an additional operand field for each instruction. This operand field is used to specify a register that holds the guard condition; the register could either be a general-purpose register, or part of a special predicate register file [18, 20].

Introducing guarded execution into scalar processors can be a very powerful concept; Figure 1 presents a small example. Figure 1(a) show the C-code for inner loop of the Cmppt function of the SPEC92 benchmark Eqntott. In Figure 1(b) we show the corresponding assembly code using a generic MIPS-like assembly language. In Figure 1(c) we show the same code using guarded instructions (if-conversion is used to transform the code). In Figure 1(c), c_move is a conditional move, and c_li is a conditional load immediate. The last operand of the conditional instruction is the condition register.

Comparing Figures 1(b) and 1(c) we can see that four static branches were eliminated (corresponding to the first, the second and the fourth if-statements and one of the return statements in the C-code), and that the basic blocks are considerably larger: the MIPS-like assembly contains 10 non-branch and 7 branch instructions, while the guarded version contains 13 non-branch and 3 branch instructions.

With guarded execution, code for a scalar processor has fewer branches, larger basic blocks, and no control dependencies. This results in several important advantages. First, the compiler has a larger (static) basic block, with more instructions to extract parallelism from. This allows the compiler to produce a better (and more parallel) schedule. Second, since the number of branches (static, and therefore dynamic) is reduced, the number of instructions between mispredicted branches (or the window size) can increase. A larger window size provides more opportunities for dynamic parallelism extraction.

Existing proposals for guarding, however, have several problems that inhibit its (widespread) use in scalar processors. The first problem is that guarded execution, in its most general form, is not easy to integrate into existing instruction sets. Since each (guarded) instruction needs a guard operand, existing techniques for specifying guarding require each (guarded) instruction to have an additional source operand specifier. With an existing instruction set, it is generally not possible to find a sufficient number of bits to explicitly specify an additional source operand for a large variety of instruction types. (An additional source operand specifier also implies an additional read port on the register file.) This problem has forced instruction set designers to allow only a small number of guarded instructions. The DEC Alpha [6] and the SPARC V9 [4] architectures are prime examples, offering a *conditional move* (CMOVE) instruction. (Since a move instruction has only two operands (one source and one destination), bits to specify the third operand (the guard register) explicitly can easily be taken from unused instruction bits (the second source operand of computation instructions).)

The second problem is that guarding increases the total number of instructions executed dynamically. In general, instruction from both paths (traversed and not traversed) of a branch instruction are executed when they are transformed into guarded instructions; instructions from the not-traversed path are

```
for (i = 0; i < ninputs; i++) {
   aa = a[0]->ptand[i];
   bb = b[0]->ptand[i];
   if (aa == 2)
      aa = 0;
   if (bb == 2)
      bb = 0;
   if (aa != bb) {
      if (aa < bb)
         return -1;
      else
         return 1;
   }
} /* rof */
```

(a)

```
L0 lh      a0,0(a1)
   lh      a2,0(a3)
   bne     a0,t0,L1
   move    a0,zero
L1 bne     a2,t0,L2
   move    a2,zero
   slt     at,a0,a2
L2 beq     a0,a2,L4
   beq     at,zero,L3
   li      v0,-1
   jr      ra
L3 li      v0, 1
   jr      ra
L4 addiu   v0,v0,1
   addiu   a1,a1,2
   addiu   a3,a3,2
   bne     v0,v1,L0
```

(b)

```
L0 lh      a0,0(a1)
   lh      a2,0(a3)
   set_eq  c0,a0,t0
   c_move  a0,zero,c0
   set_eq  c1,a2,t0
   c_move  a2,zero,c1
   slt     at,a0,a2
   beq     a0,a2,L4
   set_ne  c3,at,zero
   c_li    v0,-1, c3
   c_li    v0, 1, ! c3
   jr      ra
L4 addiu   v0,v0,1
   addiu   a1,a1,2
   addiu   a3,a3,2
   bne     v0,v1,L0
```

(c)

Figure 1. The Cmppt inner loop. Part (a) shows the C-code, part (b) shows the corresponding MIPS-like assembly, and part (c) shows the same assembly using guarded instructions.

dynamically transformed into NOPs. That is, instructions from both paths are fetched and enter the processor pipeline, even though some of them may be transformed into NOPs in the earlier stages of the pipeline. The reason that the processor has to fetch and decode all these extra (non-useful) instructions is that the processor has no way of knowing that an instruction is guarded until it is fetched and examined. If the instruction is guarded, the processor can't determine if it should be transformed into a NOP until both the condition (name) and the condition value are known. These extra instructions, from the not-traversed path, could 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, these additional instructions can actually degrade the overall execution time. Execution time can also be degraded 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 concern is that guarding uses additional architecturally visible registers to hold the (guard) conditions for the subsequent guarded instructions. Without guarding, the register that holds the condition is used once, to decide the branch outcome and set the correct PC value; instructions that are control dependent on that condition will be discarded or fetched according to this target 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 span up to the last guarded instruction; this increases the register pressure. The problem is exacerbated by the instruction scheduler which, by rearranging the instructions, can increase the register lifetime. A possible solution to this problem is to add a separate predicate register file [18, 20], to relieve the pressure on the architectural registers. This solution, however, may result in extra instructions to transfer values between the two register files and clearly cannot be easily incorporated into existing architectures.

Because of these drawbacks, instruction set support for guarding is expected to be limited (unless one has the luxury of designing a new instruction set) and guarding can be profitably applied only in certain cases. Mahlke *et al* addressed some of these issues for statically scheduled machines (such as VLIW), taking in account mainly the basic block size and the execution frequency [18]. In their scheme, a *Hyperblock* of instructions is formed, using trace selection based on branch frequencies, such that the Hyperblock has a single entry point and one or more exit points. Branches that are not amenable to static prediction are eliminated using if-conversion. Finally, after the Hyperblock formation, the instructions are scheduled using conventional parallelism enhancing techniques.

The problem of wasted computation resulting from if-conversion was addressed by Warter *et al* [24]. They propose the use of if-conversion before the instruction scheduling phase of the compiler, to eliminate the control dependencies and expose parallelism to the optimizer. After the optimization phase, a *reverse if-conversion* transformation is proposed, in which guarded computation is transformed back into normal instructions covered by conditional branches. This technique improves the static schedule without increasing the execution time of any of the paths. However, reverse if-conversion can increase the static size of the program significantly and can cause high instruction cache miss ratios. Reverse if-conversion can also increase the number of executed branches: if instructions that were control dependent on a single branch instruction before if-conversion and scheduling, are moved apart during scheduling, the reverse if-conversion technique will introduce multiple branch instructions to ensure the correct execution in the new schedule. (In general, these branches will be executed in parallel with other, useful computation, given sufficient resources.)

We believe that guarded execution is very useful to ILP processors (and will conform our beliefs in the following sections), as it allows the compiler and the hardware to exploit more of the available instruction level parallelism. The potential drawbacks of guarding, however, are bothersome. We believe that the potential disadvantages of guarded execution, which we have outlined above, *are not a fundamental disadvantage of the guarding approach.* Rather, we believe that they are a problem caused by *existing methods of specifying guarded computation*. In section 6, we will present and evaluate a different scheme for specifying guarded computation, one that does not have the drawbacks mentioned above. Before doing that, however, a performance assessment of the guarding concept is in order.

3. Evaluation of Guarded Execution

In this section we describe the experimental framework that we used to quantitatively assess the guarding concept. Most of the work is done by a trace-driven simulator that simulates the execution of programs, with and without guarded instructions, and collects the necessary statistics.

3.1. Benchmarks

For benchmark programs, we used the entire integer SPEC92 benchmark suite, namely the programs Compress, Eqntott, Espresso, Gcc, Sc and Xlisp. We also used three architecture simulators, Tycho, a cache simulator [12], Supermips, a superscalar processor simulator based on the MIPS instruction set, and Thissim, a trace driven simulator similar to the one we used for this study. Finally, we used the TeX text formatter and the Yacc parser generator, as well as two Object Oriented Database benchmarks, Sunbench and Tektronix. We did not use Fortran programs since (numeric) Fortran programs typically have very regular (and uninteresting) control structures. (Guarded execution is of little interest for numeric programs with regular control structures since it is comparatively easy to extract significant amounts of ILP for such programs with existing techniques.)

Our benchmark programs were compiled for a MIPS based DECstation 3100, using the version 2.1 of the MIPS compiler. Table 1 shows our 13 benchmark programs and their basic statistics, including the number of instructions (excluding NOPs) that we allowed for execution, and the ratio of conditional and unconditional branches in the dynamic instruction stream.

3.2. Metrics

The best metric for evaluating any concept in processing is the total execution time. However, this metric requires many implementation assumptions, including the exact hardware configuration, functional unit latencies, etc. Other direct metrics, such as CPU time and speedup also require implementation assumptions that limit the utility of results (for example, an ideal memory system is assumed in many studies). To avoid making implementation assumptions, which introduce another set of parameters into the performance equation, we use indirect measures of performance. While these measures may not translate easily into a direct metric for an implementation, they do provide insight into the utility of the concept.

Program	Dynamic Instructions (Millions)	Branch Ratio	
		Conditional	Unconditional
Compress	78.59	0.149	0.040
Eqntott	300.00	0.306	0.012
Espresso	300.00	0.176	0.014
Gcc	128.78	0.156	0.042
Sc	300.00	0.207	0.037
Sunbench	300.00	0.148	0.067
Supermips	300.00	0.111	0.056
Tektronix	300.00	0.136	0.082
TeX	214.69	0.143	0.055
Thissim	300.00	0.105	0.046
Tycho	300.00	0.123	0.061
Xlisp	300.00	0.157	0.091
Yacc	26.37	0.237	0.020

Table 1. Benchmark Program Characteristics

Our first metric is the *effective guarded block size*. This is the (dynamic) average size of the blocks after guarding. In this size we count only instructions from the original program that *contribute to useful computation*. We do not count any instructions required to set the condition registers. We also do not count any (static) NOPs or any guarded instructions that are dynamically transformed into NOPs. The advantages of this metric are (i) it is highly correlated with parallelism that can be extracted [5, 11, 18] and (ii) it is dependent only on the program and the compiler and not on the underlying hardware implementation.

Our second metric is the *static guarded block size*. This is the total number of instructions in a block, including instructions that contribute to useful computation, *as well as* instructions that are dynamically transformed into NOPS. This metric gives an indication of the instruction fetch bandwidth required during the program execution. Again, we only count instructions from the original program that were transformed into guarded instructions in this metric. We do not count any additional instructions that might be required to set condition registers.

When guarding is combined with dynamic branch prediction and speculative execution, two more metrics are of interest. These are: (i) the accuracy of the branch prediction scheme, and (ii) the number of instructions that contribute to *useful computation*, between mispredicted branches. We refer to the latter as the *dynamic window size*.

3.3. Guarded Instruction Use

To decide which instructions can be guarded, and what the guard condition should be, we apply the following algorithm. Starting at a node in the *control flow graph (CFG)* of a program, we traverse the CFG collecting nodes from all the possible paths in an attempt to create a single large block (of non-branch instructions) containing guarded instructions. This guarded block is terminated by a (possibly conditional) branch instruction. We also restrict the construction of guarded blocks so that they contain no more than 15 basic blocks of the original code[1]. Effectively, the guarded block formed by this algorithm is the maximal subgraph of the CFG, containing less than 15 nodes, and having at most two targets (that can be described by a single branch).

In our guarded block formation we do not perform any function inlining or loop unrolling; the MIPS compiler would already have unrolled for-loops four times. Should aggressive loop unrolling and function inlining be performed, the potential of guarded execution would be enhanced. The guarded blocks constructed by our algorithm differ from the ones constructed in the Hyperblock formation of [18] in two ways. First, we require that all branches internal to the block (except the last one) are eliminated by the if-conversion; a Hyperblock is allowed to contain multiple branches and exit points. We treat what would be a Hyperblock in [18] as a sequence of basic blocks and guarded blocks. Our metrics, namely the effective and the static guarded block size, are not affected by these differences as the useful computation remains the same, and the number of branches and the non-useful guarded computation depend solely on the if-conversion transformations. Second, the Hyperblock's heuristic basic block selection function may decide to apply if-conversion in different parts of the control flow graph than our simpler algorithm.

The above describes how the program flow is taken into account in the guarding process. Another input to the guarding process is the nature of the guarded instructions available in the instruction set. We distinguish between two types of guarding: *full guarding* and *restricted guarding*.

3.3.1. Full Guarding

In full guarding, we assume that all instructions are available in guarded form, and that the guard conditions can be set by the normal computation without any overhead. Under these two assumptions, if-

[1] This limit was set to ensure that the size of the guarded blocks will be reasonable, in that they do not contain too much unused computation. We observed that the effects of this limitation on the performance were negligible.

conversion is only limited by the structure of the CFG; any sub-graph meeting our restrictions can be transformed into a guarded block. The results obtained under these assumptions are an indicator of the best performance (according to our metrics) that one can expect from guarding.

3.3.2. Restricted Guarding

Because of opcode space limitation, many instruction set architectures cannot be extended to include guarded versions of all instructions. For partial guarding support, the most important subset of instructions are the ALU instructions, because they usually require fewer bits to encode[2]; the unused bits can be used to specify that the instruction is guarded and encode the condition register. Load and Store instructions usually contain an immediate field and two register specifiers, and do not leave any space to specify the condition register. In restricted guarding, only blocks with ALU operations can be guarded; memory accessing instruction can only appear in an unconditional part of the guarded block. Guarded blocks constructed with restricted guarding are therefore a subset of the blocks constructed with full guarding[3].

One way to provide support all instructions in guarded version is to synthesize them using normal instructions that store their results into temporary registers, and then using the supported conditional instructions (such as conditional moves) to commit these results. This method can be used as long as the compiler can guarantee that none of the unconditional instructions used in the synthesis will ever generate an exception. However synthesis of guarded instructions entails additional overhead, and it is not clear if this overhead will be more than compensated by the improvement in the effective basic block size. A compact and efficient way to specify guarded instruction, that does not suffer from these limitations will be described in section 6.

4. Branch Elimination Potential and Overhead of Guarding

We first consider the branch elimination potential of guarding. Table 2 presents the percentage of (dynamic) branches that are eliminated from the instruction stream with full and restricted guarding. On average, full guarding is able to eliminate an average of 30.74% of all dynamic branches, eliminating over half the branches in some cases. Restricted guarding is not as powerful as full guarding, for obvious reasons, in almost all cases; on the average, it is able to eliminate only 14.26% of all dynamic branches. An exception is Eqntott, which spends most of its time in an inner loop that is amenable to restricted guarding.

Figure 2 presents the basic block size, the effective guarded block size, and the static guarded block size, for each of the benchmark programs when full guarding is used; Figure 3 presents the same when restricted guarding is used. From Figure 2 we can see that full guarding is quite effective in increasing the effective guarded block size, increasing the block size by 52% on average (from 4.82 useful instructions in a basic block to 7.33 useful instructions in a guarded block). In most cases the effective block size is increased by at least 25%; in one case (Thissim), it is more than doubled. The increase in the effective block size, however, come with a price — an increase in the total number of instructions that are executed. This increase is measured by the static guarded block size. Comparing the static guarded block size to the effective guarded block size, we see that 33% of all instructions that would be executed (or at least fetched and decoded), with full guarding, do not contribute to useful computation. For most programs, 20-50% of the instructions executed are non-useful instructions. We feel that this overhead is significant, and needs to be dealt with, for guarding to become widely accepted.

[2] In the MIPS instruction set, an ALU instruction is specified with the 6-bit opcode SPECIAL, a 6-bit function field and three 5-bit register specifiers. That leaves 5 bits unused, exactly as many as we need for the guard register specifier.

[3] Our choice for restricted guarding is one meaningful way of restricting the guarding process. Clearly other meaningful ways are possible; we do not consider them in this paper.

Program	Percent of Eliminated Branches	
	Full	Restricted
Compress	24.86	18.24
Eqntott	40.53	40.03
Espresso	16.76	10.17
Gcc	31.56	9.16
Sc	41.46	8.02
Sunbench	35.68	11.33
Supermips	50.70	17.15
Tektronix	36.54	15.77
TeX	12.80	5.99
Thissim	62.31	23.26
Tycho	15.05	6.45
Xlisp	13.64	13.63
Yacc	17.82	6.22
Arithmetic Mean	30.74	14.26

Table 2. Percent of Dynamic Branches Saved by Full and Restricted Guarding

The magnitude of the overhead is not as large with restricted guarding: only 8% of the instructions do not represent useful computation, on average. However, restricted guarding also does not increase the effective guarded block size as well as full guarding (8% versus 52%, on average).

From Table 2 and Figures 2 and 3 we can conclude that full guarding is a powerful technique able to eliminate a significant fraction of the branches of a program and achieve a significant increase in the

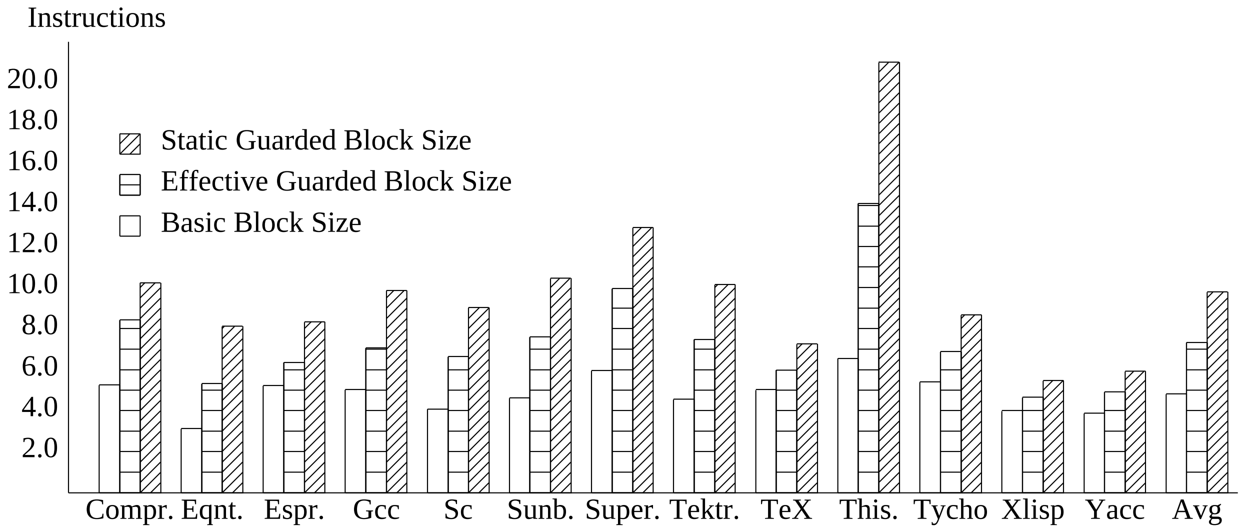


Figure 2. Performance of Guarding with Full Instruction Set Support

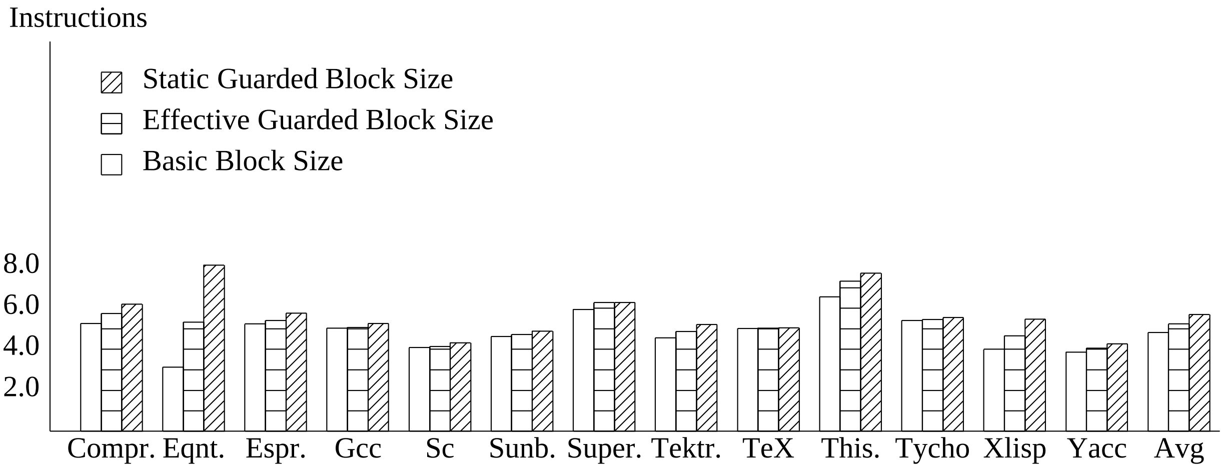


Figure 3. Performance of Guarding with Restricted Instruction Set Support

(guarded) block size. These results suggest that instruction set support for guarding is desirable. They also suggest that a limited support for guarded execution in the instruction set may not be sufficient.

5. Interaction of Guarding and Dynamic Branch Prediction

We now consider the interplay between guarding and dynamic branch prediction. There are two main issues that we address here. First, how does guarding impact the prediction mechanism. Guarding can impact the prediction mechanism in two (not entirely unrelated) ways. One, it can reduce the number of branches that are predicted. If the branches that guarding eliminates are ‘‘bad’’ branches, i.e., branches with poor predictability, guarding can improve prediction performance (as measured by the branch prediction accuracy); if they are ‘‘good’’ branches, guarding can degrade prediction performance. Two, since the number of (static and dynamic) branches that need to be predicted is changed, the mechanics of the prediction mechanism could change completely. For example, if the prediction mechanism is counter-based [17, 22], a prediction mechanism with a smaller number of table entries may suffice. If the prediction mechanism is pattern-based [19, 25], its behavior might be radically altered since the pattern of dynamic branches is changed. Second, what is the impact of guarding on the dynamic window size (useful instructions between mispredicted branches) that can be established, and how the size of this window varies with the guarding and the prediction strategy.

We use two different prediction mechanisms: a 2-bit counter-based mechanism, and a GAs(8,x) pattern-based mechanism [25]. For either case, we used three different table sizes: 1K, 4K, and 16K entries. (These translate into prediction tables of 2K, 8K, and 32K bits, respectively.) The above configurations are chosen since they are reasonable in terms of size and hardware complexity while achieving respectable performance.

We first address the issue of the branch prediction accuracy. Table 3 presents the branch prediction accuracies without guarding, with restricted guarding, and with full guarding, for counter-based predictors with 1K, 4K, and 16K entries. Table 4 presents the same for pattern-based predictors. Unfortunately, there are no (monotonic) trends in the tables that allow us to make concrete statements about the (positive or negative) impact of guarding on the prediction mechanism. In some cases guarding improves the prediction accuracy, in other cases it does not. In one case (Espresso), full guarding improves the prediction accuracy whereas restricted guarding does not. The results also do not allow us to conclude much about the impact of guarding on the sizes of the prediction tables. Even though we had expected larger table sizes to be less beneficial than smaller table sizes, in the presence of guarding (especially for counter-based mechanisms which hash the address of the static branch to access the prediction table), this expectation was not borne out by the data present in the tables. It appears that both pattern-based and counter-based prediction mechanisms will continue to be beneficial (but not to a point where they do not need further improvement) even in the presence of guarding, and that larger table sizes will continue to be beneficial (at least in the range of sizes that we considered).

Program	Size 1K entries			Size 4K entries			Size 16K entries		
	BB	Full	Restr.	BB	Full	Restr.	BB	Full	Restr.
Compress	87.20	88.57	89.00	87.20	88.57	89.00	87.20	88.57	89.00
Eqntott	83.76	97.53	97.04	83.76	97.54	97.54	83.76	97.54	97.54
Espresso	90.29	92.09	89.54	90.30	92.11	89.55	90.30	92.12	89.55
Gcc-cc1	85.89	86.13	86.25	87.65	87.15	87.75	88.24	87.81	88.29
Sc	94.70	94.24	94.52	94.91	94.43	94.87	94.95	94.37	94.89
Sunbench	91.30	87.51	90.21	91.35	89.34	91.39	91.36	89.34	91.39
Supermips	96.13	97.76	95.67	96.53	97.77	95.81	96.53	97.77	95.81
Tektronix	90.60	86.44	87.01	91.15	89.28	90.24	91.15	89.28	90.24
TeX	94.12	94.83	94.01	94.72	95.20	94.50	94.79	95.25	94.56
Thissim	96.03	93.59	95.24	96.03	93.59	95.24	96.03	93.59	95.24
Tycho	93.41	94.30	93.24	93.41	94.30	93.24	93.41	94.30	93.24
Xlisp	88.01	87.16	87.35	88.16	87.20	87.40	88.31	87.21	87.40
Yacc	93.66	94.61	93.57	93.68	94.78	93.69	93.68	94.78	93.69
Arithmetic Mean	91.16	91.90	91.74	91.45	92.40	92.32	91.51	92.45	92.37

Table 3. Branch Prediction Accuracies for a Counter-based Predictor

Program	Size 1K entries			Size 4K entries			Size 16K entries		
	BB	Full	Restr.	BB	Full	Restr.	BB	Full	Restr.
Compress	88.46	90.38	90.34	88.71	90.38	90.76	89.23	90.48	90.77
Eqntott	92.73	97.95	97.71	93.15	98.09	98.09	93.97	98.16	98.16
Espresso	96.03	96.50	95.68	96.56	96.84	96.20	96.74	96.95	96.43
Gcc-cc1	83.82	84.82	84.19	88.76	89.13	88.96	91.48	91.63	91.62
Sc	94.33	94.42	94.39	95.90	95.64	95.87	96.60	96.25	96.49
Sunbench	96.61	95.39	94.18	98.03	96.84	97.35	98.07	97.45	97.77
Supermips	93.76	93.46	93.62	96.00	96.35	95.58	97.20	98.12	96.89
Tektronix	93.81	94.40	93.14	96.01	95.95	96.25	96.91	96.68	96.82
TeX	91.64	92.52	92.19	94.80	95.12	94.76	95.87	96.03	95.86
Thissim	96.48	95.96	95.17	96.87	96.06	96.09	97.04	96.14	96.33
Tycho	94.15	95.41	93.83	95.16	96.34	95.24	95.16	96.34	95.23
Xlisp	92.82	93.90	93.90	95.21	95.37	95.33	95.63	95.45	95.41
Yacc	94.87	95.80	95.10	95.60	96.36	95.64	95.84	96.56	95.79
Arithmetic Mean	93.04	93.91	93.34	94.67	95.26	95.08	95.36	95.86	95.65

Table 4. Branch Prediction Accuracies for a Pattern-based Predictor

We now address the issue of dynamic window size. The dynamic window size is influenced both by the prediction mechanism, as well as the number of useful instructions between branch predictions[4]. Figure 4 presents the dynamic window sizes[5] without guarding (basic blocks), with restricted guarding, and with full guarding, for each of the benchmark programs, using the counter-based prediction mechanism; Figure 5 presents the same for the pattern-based prediction mechanism. The predictors in Figures 4 and 5 have 4K entries each.

With the counter-based predictor, a window size of about 101 instructions, on average, can be established without guarding. With restricted guarding, the average window size increases to about 123 instructions, and with full guarding it further increases to about 193 instructions. Almost all programs (the exception being Xlisp), benefit significantly from guarded execution, especially full guarding. The benefits for Eqntott are especially impressive, with the window size increased by over a factor of 10.

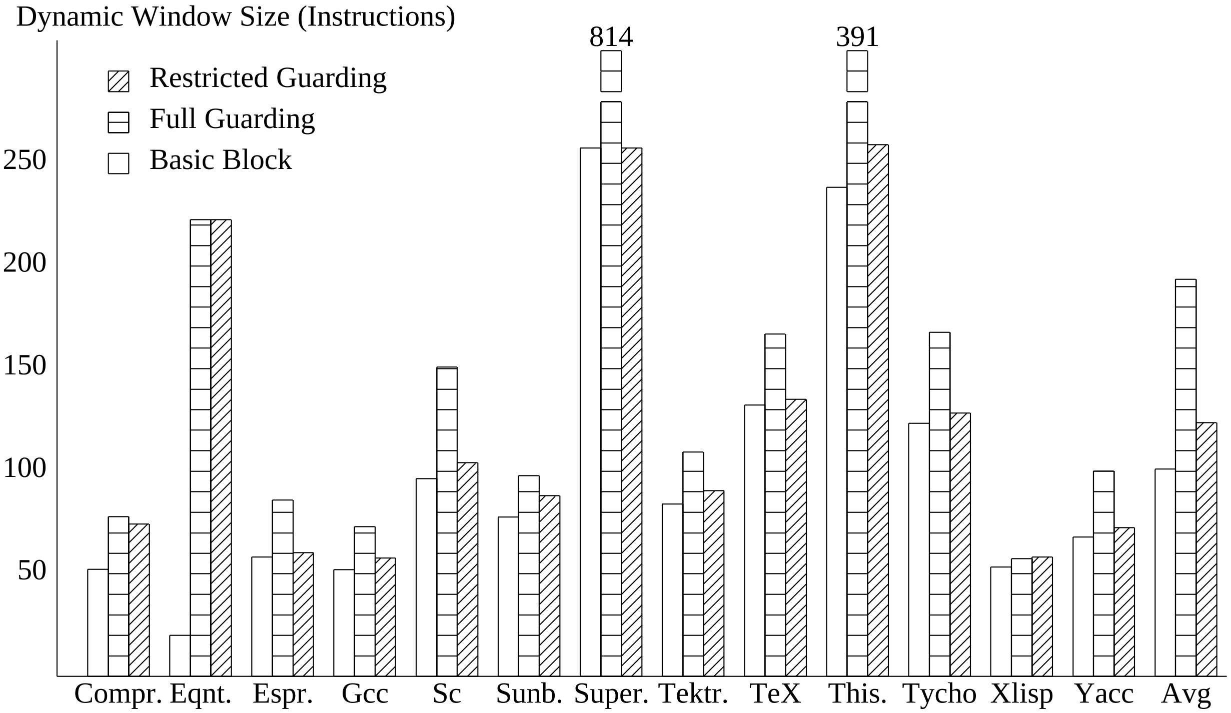


Figure 4. Effects of Guarding on the Window Size for a Counter-based Predictor with 4K Entries

With a pattern-based predictor, the results are even more impressive. A pattern-based predictor can establish a respectable window size (about 156 instructions, on average), even without guard instructions, because of its superior prediction abilities. With full guarding, it can establish a window size of about 258 instructions, on average; for almost all programs the (average) window size is greater than 150 useful

[4] It is interesting to compare the dynamic window sizes that can be established with guarding and dynamic branch prediction with the window sizes that can be established with trace scheduling, which uses static branch prediction [10].

[5] The window size for each program is the average of all window sizes that are established dynamically.

instructions.

One program, Sunbench, has an anomalous behavior with a pattern-based predictor: the window size with guarding is (slightly) smaller than the window size without guarding! The reason for this behavior is that, after the if-conversion, the dynamic pattern of taken and not-taken branches is dramatically changed. This modified sequence is less predictable using a pattern based predictor, resulting in a larger number of mispredicted branches, and a smaller window size. Since counter-based prediction is sensitive to the identity of the branch, and not the dynamic taken/not-taken pattern, it does not display this anomalous behavior.

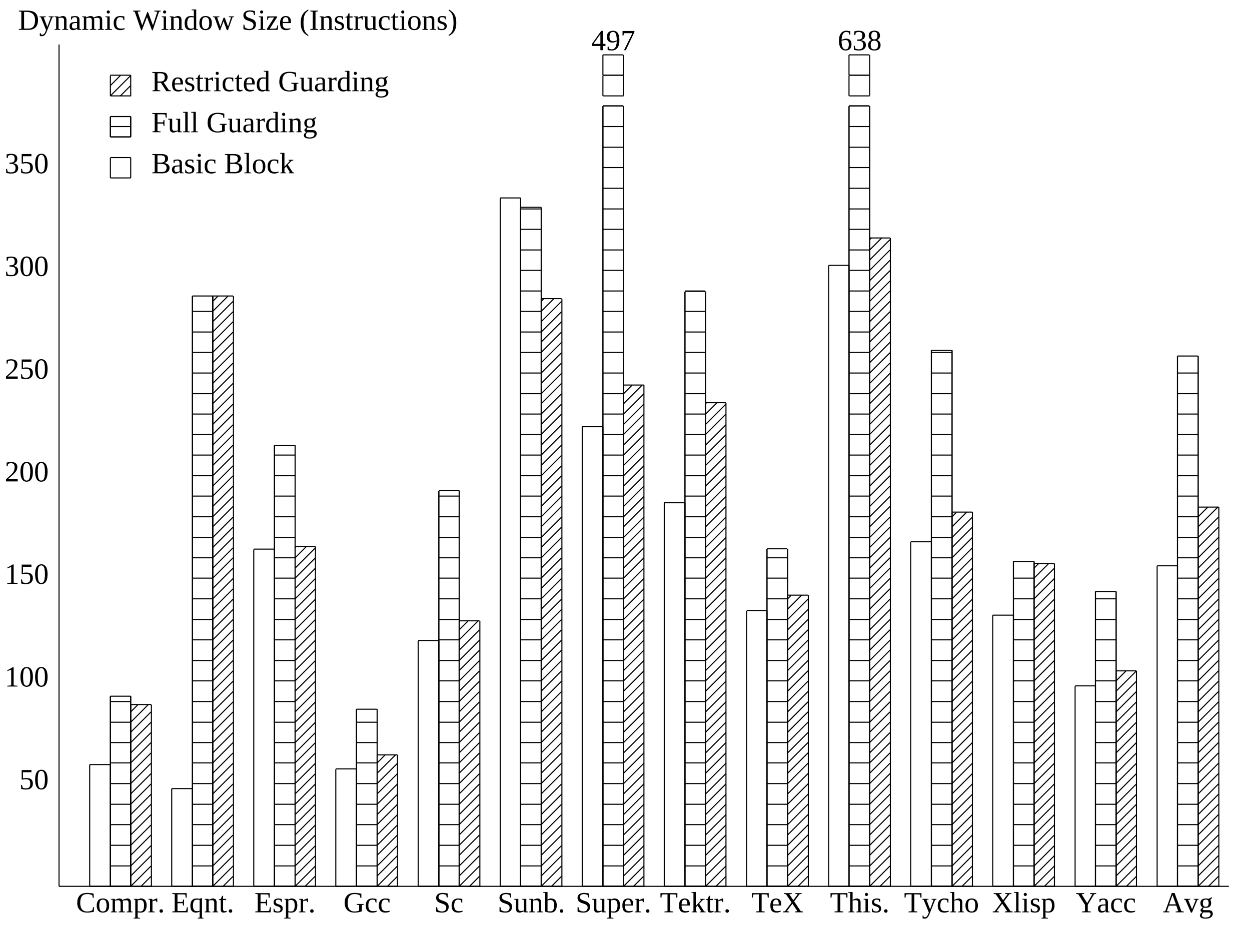


Figure 5. Effects of Guarding on the Window Size for a Pattern-based Predictor, with a Single 8-bit Shift Register in the First Level and 4K Entries in the Second Level

6. An Alternate Specification of Guarded Execution

The previous sections suggested that guarded execution is a very useful concept, especially when coupled with a good dynamic branch prediction scheme, both in increasing the block size available for software optimizations, as well as in increasing the (dynamic) window size from which parallelism can be extracted. However, as we saw, guarded execution, as specified using explicit guard condition operands with each (guarded) instruction has several drawbacks. First, the specification of a guarded instruction requires valuable instruction space that may not be available without a complete rehaul of the ISA. Second, the implementation of guarding requires an additional read port in the register file. Third, the number of instructions that have to be executed is increased, thereby increasing the required instruction fetch and decode (and possibly execute) bandwidth. Fourth, there could be increased pressure on the architectural register file, if it is used to hold guard conditions.

As we had alluded to earlier, many of these drawbacks are an artifact of how guarding is specified. The performance problems, in particular, are due to the fact that the processor has no idea of what instructions it is going to encounter (soon) in the dynamic instruction stream. Without this information, the processor has to fetch and decode all the instructions in a block to determine their guarding status.

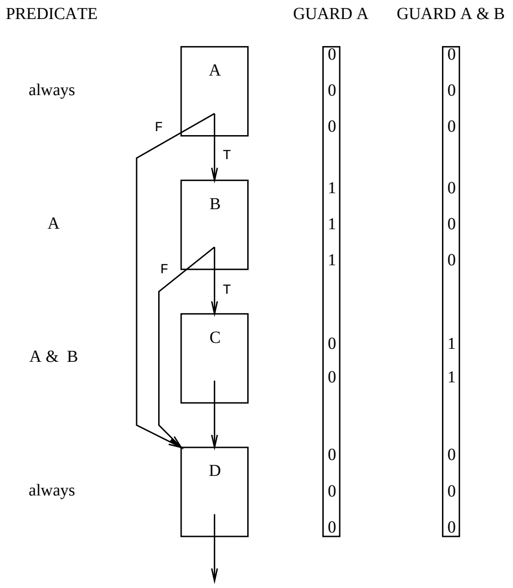
The solution that we propose below allows the guarding concept, in its full form, to be integrated easily into existing instruction sets (it requires the addition of a very small number of instructions, and no modifications to existing instructions), and also overcomes the performance problems mentioned above. The solution makes use of two observations. First, instructions guarded by the same condition are likely to be in close proximity in the static and dynamic instruction stream. (Our results indicated guarded blocks of a few tens of instructions. Even very aggressive code motion techniques are unlikely to increase this considerably, into the many tens, or hundreds of instructions.) Second, most instructions in close proximity are likely to be guarded by the same guard condition (in true or complement form), or by a very few number of guard conditions; many are likely to have the same guard condition.

Given the close proximity of guarded instructions, and the low information content of the guard condition specification, we can specify a block of guarded instructions directly using one or more ‘‘GUARD’’ instructions. A GUARD instruction has two operands, a register that specifies the guard condition, and a mask that specifies which of the instructions following this GUARD instruction (in the static and also dynamic code) are guarded by the specified condition. A GUARD instruction, therefore, provides an (implicit) specification of the guard operands of each guarded instruction.

Figure 6 presents a small example illustrating the use of the GUARD instruction. The simple control flow graph in Figure 6 consists of four basic blocks forming two nested if structures. The column labeled PREDICATE indicates the condition that has to hold for a basic block to execute. To specify the guarded execution of basic blocks B and C we need two GUARD instructions for each of the conditions *A* and *A* & *B*. The corresponding guard masks are shown vertically in Figure 6. In these masks, a 1 indicates that the corresponding instruction is guarded by the condition register, and a 0 means that the instruction is not dependent on that condition. The figure also shows the assembly code without guarding, and with guarding specified using the GUARD instruction. Notice that the non-branch instructions in both cases are identical in every respect; the only difference between the two codes is the elimination of the branch instructions, the use of the and instruction to set a guard condition in r3, and the use of GUARD instructions to specify guarding.

One way to implement the GUARD instruction is to treat it as a meta-instruction, whose sole effects are to specify the condition register of subsequent guarded instructions. In this case, the instruction can be interpreted by the hardware which will expand all the instructions specified in the guard mask adding the condition register, and will keep the expanded guarded instructions close to the execution stages in a decoded instruction cache [8]. This ‘‘macro’’ expansion can take place on demand, when the instruction cache misses, so it will not affect the critical path of instruction fetching or execution. This implementation overcomes the opcode space limitations and enables the processor designer to explore the full spectrum of possible implementations of guarding and exploit its potential.

However, the GUARD instruction has even more potential. By defining the semantics of the GUARD instruction such that the condition is evaluated by the processor when the GUARD instruction is executed, the processor is allowed to evaluate the guard condition for *all* instructions guarded by this condition at



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
```

Assembly Code Using GUARDs

```
     and   r3,  r7, r5
     GUARD r7,  0001110000
     GUARD r3,   000001100
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
```

Figure 6. Example of the Use of the GUARD Instruction

once[6]. The benefits of this definition are twofold. First, since the condition is evaluated only once, instructions guarded by the condition do not need to specify and read the condition register, eliminating the decoded instruction format, the decoded instruction cache, and the additional read port in the register file. 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 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 6, if the condition in r7 evaluates to false, then the processor could jump to i8 after it is done with i2, since i3-i7 will dynamically be transformed into NOPs.)

To support the GUARD instruction, the processor has to maintain a mask of active and inactive instructions. This mask, called the *active mask*, is just a shift register: if the *i*-th bit in the shift register is 1, the *i*-th instruction (starting 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. When an GUARD instruction is executed, its mask and guard condition (cond) are used to update the mask as follows:

$$active\_mask_i = active\_mask_i \ \& \ ((mask_i \ \& \ cond) \mid (\, ! \, mask_i))$$

The intuition behind this equation 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 active_mask is shifted by one position, with a one being shifted in.

The active mask is key in permitting the processor to effectively squash unnecessary computation. The processor can identify such unnecessary computation by performing a population count (*count*) on

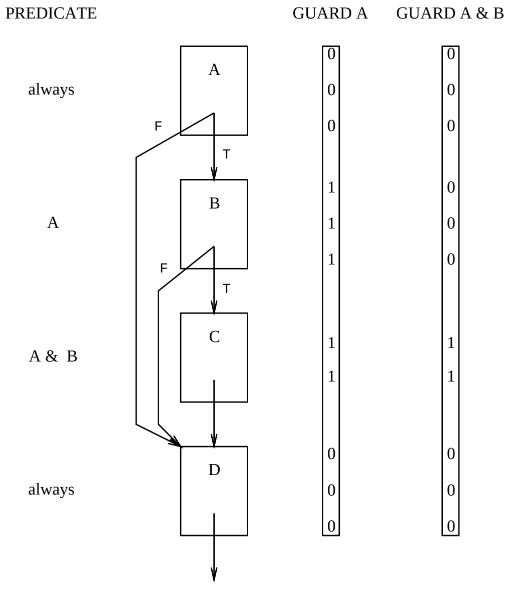
[6] Assuming, of course, that the number of instructions to be guarded is less that the number of bits in this mask. If there are more instructions than bits in the mask, we would have to use several, suitably-placed, GUARD instructions.

the leading zeros of the active 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 the next useful instruction). Therefore, instructions that will dynamically be transformed into NOPs are not even fetched into the pipeline.

The GUARD instruction allows for two additional optimizations. First, when multiple GUARD instructions cover the same instruction, the conditions are implicitly AND-ed in the active mask. This ability can reduce the amount of logic manipulation instructions required to set all the necessary conditions. Second, since the condition register is evaluated exactly once when the GUARD instruction is executed, the register holding the condition can be reused immediately after the GUARD instruction. This ability, coupled with the implicit AND ability of the GUARD instruction, could alleviate the register pressure problem.

Figure 7 illustrates the use of the implicit AND property of the GUARD instruction on the simple example of Figure 6. Note that the conjunction of conditions *A* and *B* that guards basic block C is achieved by setting the mask bits in both GUARD instructions, obviating the need to perform the logical AND instruction that was required to compute the *A* & *B* condition in Figure 6.

The active_mask is part of the processor state, and has to be saved and restored on interrupts and context switches. The saved active mask, together with the saved program counter value provide sufficient information to restart the process correctly. It is fairly straightforward to introduce user-level instructions to save and restore the active_mask. An alternative to exposing the active mask to the processor state, is to require that interrupts will be accepted only on PC values that correspond to a ‘‘clear’’ state (i.e. to an active mask with all the bits set), in which case the PC value is sufficient to fully describe the state of the processor and to restart the process. In this approach, handling of traps (which cannot be deferred until the state of the processor becomes clean) requires that processor reverts to the last PC for which the state was clean, in a manner similar to the checkpoint repair of [15].



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
```

Assembly Code Using GUARDs

```
     GUARD r7,  0001111100
     GUARD r3,   000001100
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
```

Figure 7. Example of the Implicit AND-ing of Guard Conditions

The exact number of GUARD instructions that need to be added to an instruction set, and the nature of encoding of the mask field, are something that need more study. In this paper, we evaluate the utility of two flavors of GUARD instructions. The first flavor uses a unary encoding of the mask: bit *i* specifies if the *i*th instruction following the GUARD instruction is guarded by the specified condition. Opcodes of GUARD_TRUE and GUARD_FALSE are needed to specify true and false guard conditions for a guarded code block (since a guarded block will contain instructions from both the taken and the not-taken path). (The GUARD instructions in Figures 6 and 7 are GUARD_TRUE instructions.) The second flavor uses a single opcode, GUARD_BOTH, but encodes the mask so that the guard conditions (true, false, and unconditional) of 3 instructions can be specified in 5 bits. For a MIPS-like instruction format, up to 21 bits can be comfortably used for a mask. With this mask size, a GUARD_TRUE (or GUARD_FALSE) instruction can guard up to 21 instructions (but both guard instructions will be required to guard instructions from the taken and not taken paths of a branch in general). A single GUARD_BOTH instruction can guard up to 12 instructions (4 sets of 5 bits each) from both paths of a single branch.

Table 5 presents the overhead of GUARD instructions to achieve full guarding as we did in Section 3. Guarding using GUARD_TRUE/GUARD_FALSE instructions increases the dynamic instruction count by 13.9%, on average; guarding using a GUARD_BOTH instruction by 9.72%. This is a significant savings when compared to the (average) 33% overhead that we had with the traditional way of specifying guarding[7]. A point worth mentioning regarding the results of Table 5 is that we did not attempt to optimize the use of GUARD instructions in our experiments. In particular, we did not attempt to use the implicit AND property that we illustrated in Figure 7, and we also restricted our use to a single flavor (GUARD_TRUE/GUARD_FALSE or GUARD_BOTH, but not a mixture of the two) of GUARD instructions. We are experimenting with these optimizations; we expect they will decrease the overhead of specifying guarded execution even further.

Program	GUARD_TRUE/GUARD_FALSE Overhead (Percent)	GUARD_BOTH Overhead (Percent)
Compress	12.35	10.80
Eqntott	39.28	29.51
Espresso	7.24	6.92
Gcc	11.33	7.91
Sc	17.46	14.96
Sunbench	18.81	9.74
Supermips	7.33	4.62
Tektronix	17.91	8.46
TeX	7.85	5.59
Thissim	11.41	5.95
Tycho	11.03	6.37
Xlisp	7.95	6.82
Yacc	10.79	8.74
Arithmetic Mean	13.90	9.72

Table 5. Overhead of the GUARD Instructions

[7] A point to keep in mind is that even if the 33% overhead was deemed to be tolerable, the traditional way of specifying guarding can't easily be integrated into existing instruction sets.

7. Conclusions

We studied the use of guarded execution, or guarding, in dynamic ILP processors in this paper. We had three goals in mind. One, a qualitative and quantitative assessment of the guarding concept using a variety of application programs with complex control structures. Two, the interaction of guarding and dynamic branch prediction. Three, proposing a new way of specifying guarded computation that alleviates (or even eliminates) many of the drawbacks of existing methods of specifying guarded execution.

Our evaluation of guarding suggested that guarding is a very powerful concept, and can be of great use to dynamic ILP processors. Specifically, the use of an arbitrary set of guarded instructions (or full guarding) can increase the effective block size, measured as the number of instructions between branches that actually contribute to useful computation, by about 52%, on average, for our benchmark programs. This increased size provides more flexibility for software optimizations. Using full guarding also allows a dynamic ILP processor to establish dynamic windows (or useful instructions between mispredicted branches) of about 193 and 258 instructions, on average, using counter- and pattern-based predictors (with 4K entries in the prediction table), respectively. Without any form of guarding, counter- and pattern-based predictors could establish windows of only 101 and 156 instructions, respectively. However, with full guarding, the processor has to fetch and decode 33% more instructions, on average; these instructions do not contribute to useful computation. Restricted guarding, in which only blocks with no memory instructions are guarded, results in only 8% additional instructions, but it also does not allow us to reap the benefits of guarding fully: the effective block size is increased by only 8%, and the dynamic window sizes are increased to 123 and 184 instructions, on average, with counter- and pattern-based predictors, respectively.

The impact of guarding on the dynamic branch prediction is harder to quantify. We found no uniform patterns: in some cases guarding eliminated branches that resulted in a poor prediction accuracy (and thereby improved the overall prediction accuracy), in other cases guarding eliminated branches with a high prediction accuracy. The impact of guarding on the size of the prediction table was also hard to quantify. We had expected that because guarding eliminates many static branches, the size of the tables required by a branch prediction strategy, to achieve it best prediction accuracy, will decrease. This expectation was not borne out by our results, at least in the range of table sizes that we considered (1K, 4K and 16K entries).

Finally, we proposed a new way of specifying guarded execution using GUARD instructions. GUARD instructions can easily be added to existing instruction set architectures, and allow the full power of guarding to be realized with a smaller overhead than existing methods of specifying guarded execution. (It is possible to realize the full power of guarding with as few as three additional instructions: a GUARD_BOTH, and move instructions to save and restore the active_mask, as compared to tens of instructions to incorporate guarding using a traditional specification [18]). For our benchmark programs, two flavors of GUARD instructions allowed the full power of guarding to be realized (large effective block sizes and large dynamic windows), with an overhead of about 13.9% and 9.72%, respectively. We are carrying out more studies to reduce this overhead even further.

References

[1] R. Allen, K. Kennedy, C. Porterfield, and J. Warren, ‘‘Conversion of Control Dependence to Data Dependence,’’ *Proc. 10th Annual ACM Symposium on Principles of Programming Languages*, 1983.

[2] T. M. Austin and G. S. Sohi, ‘‘Dynamic Dependency Analysis of Ordinary Programs,’’ in *Proc. 19th Annual Symposium on Computer Architecture*, Queensland, Australia, May 1992.

[3] M. Butler, T. Yeh, Y. Patt, M. Alsup, H. Scales, and M. Shebanow, ‘‘Single Instruction Stream Parallelism Is Greater than Two,’’ *in Proc. 18th Annual International Symposium on Computer Architecture*, 1991.

[4] Brian Case, ‘‘SPARC V9 Adds Wealth of New Features,’’ *Microprocessor Report*, vol. 7, February 1993.

[5] P. P. Chang, S. A. Mahlke, W. Y. Chen, N. J. Warter, and W. W. Hwu, ‘‘IMPACT: An Architectural Framework for Multiple-Instruction-Issue Processors,’’ *Proc. 18th International Symposium on Computer Architecture*, pp. 266-275, 1991.

[6] R. Comerford, ‘‘How DEC Developed Alpha,’’ *IEEE Spectrum*, vol. 29, pp. 43-47, July 1992.

[7] J. C. Dehnert, P. Y. T. Hsu, and J. P. Bratt, ‘‘Overlapped loop support in the Cydra-5,’’ *Proc. Architectural Support for Programming Languages and Operating Systems*, pp. 26-38, 1989.

[8] D. R. Ditzel, H. R. McLellan, and A. D. Berenbaum, ‘‘The Hardware Architecture of the CRISP Microprocessor,’’ in *Proc. 14th Annual Symposium on Computer Architecture*, Pittsburgh, PA, pp. 309-319, June 1987.

[9] K. Ebcioglu, ‘‘Some Design Ideas for a VLIW Architecture for Sequential Natured Software,’’ in *Parallel Processing (Proceedings of IFIP WG 10.3 Working Conference on Parallel Processing)*, ed., Cosnard *at al*. North Holland, pp. 3-21, April 1988.

[10] J. A. Fisher and S. M. Freudenberger, ‘‘Predicting Conditional Branch Directions from Previous Runs of a Program,’’ *Proc. ASPLOS V*, 1992.

[11] J. A. Fisher, ‘‘Trace Scheduling: A Technique for Global Microcode Compaction,’’ *IEEE Transactions on Computers*, vol. C-30, July 1981.

[12] M. D. Hill and A. J. Smith, ‘‘Evaluating Associativity in CPU Caches,’’ *IEEE Transactions On Computer*, December 1989.

[13] P. Y. T. Hsu and E. S. Davidson, ‘‘Highly Concurrent Scalar Processing,’’ *Proc. 13th Annual Symposium on Computer Architecture*, pp. 386-395, June 1986.

[14] P. Y. T. Hsu, ‘‘Highly Concurrent Scalar Processing,’’ Ph.D. Thesis, University of Illinois at Urbana-Champaign, January 1986.

[15] W. W. Hwu and Y. N. Patt, ‘‘Checkpoint Repair for High-Performance Out-of-Order Execution Machines,’’ *IEEE Transactions on Computers*, vol. C-36, pp. 1496-1514, December 1987.

[16] M. S. Lam, ‘‘Software Pipelining: An Effective Scheduling Technique for VLIW Machines,’’ *Proceedings of the SIGPLAN 1988 Conference on Programming Language Design and Implementation*, pp. 318-328.

[17] J. K. F. Lee and A. J. Smith, ‘‘Branch Prediction Strategies and Branch Target Buffer Design,’’ *IEEE Computer*, vol. 17, pp. 6-22, January 1984.

[18] S. A. Mahlke, D. C. Lin, W. Y. Chen, R. E. Hank, and R. A. Bringmann, ‘‘Effective Compiler Support for Predicated Execution Using the Hyperblock,’’ *Proceedings of the 25th Annual Workshop on Microprogramming and Microarchitecture (Micro 25)*, pp. 45-54, 1992.

[19] S.-T. Pan, K. So, and J. T. Rahmeh, ‘‘Improving the Accuracy of Dynamic Branch Prediction Using Branch Correlation,’’ *Proc. Architectural Support for Programming Languages and Operating Systems (ASPLOS-V)*, October, 1992.

[20] B. R. Rau, D. W. L. Yen, W. Yen, and R. Towle, ‘‘The Cydra 5 Departmental Supercomputer: Design Philosophies, Decisions, and Trade-offs,’’ *IEEE Computer*, vol. 22, pp. 12-35, January, 1989.

[21] E. M. Riseman and C. C. Foster, ‘‘The Inhibition of Potential Parallelism by Conditional Jumps,’’ *IEEE Transactions on Computers*, vol. C-21, pp. 1405-1411, December 1972.

[22] J. E. Smith, ‘‘A Study of Branch Prediction Strategies,’’ *Proc. 8th International Symposium on Computer Architecture*, pp. 135-148, May 1981.

[23] D. W. Wall, ‘‘Limits of Instruction-Level Parallelism,’’ *Proc. ASPLOS IV*, pp. 176-188, 1991.

[24] N. J. Warter, S. A. Mahlke, W. W. Hwu, and B. R. Rau, ‘‘Reverse If-Conversion,’’ *Proceedings of the SIGPLAN 1993 Conference on Programming Language Design and Implementation*, pp. 290-299.

[25] T. Yeh and Y. Patt, ‘‘A Comparison of Dynamic Branch Predictors that use Two Levels of Branch History,’’ *Proc. 20th Annual Int’l Symposium on Computer Architecture*, May 1993.