Guarded Execution and Branch Prediction in Dynamic ILP Processors

Dionisios N. Pnevmatikatos Gurindar S. Sohi
pnevmati@cs.wisc.edu sohi@cs.wisc.edu

Computer Sciences Department
University of Wisconsin-Madison
Madison, WI 53706

Abstract

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 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 guarded execution and dynamic branch prediction and show that the use of guarded execution significantly increases the number of instructions between mispredicted branches. Third, we propose 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, 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 (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* [8] (also called *conditional execution* or *predicated* execution). By eliminating some branch instructions, the effective block size (the number of instructions between branches) is increased, thereby increasing the 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 programs. Recent architectures, such as the DEC Alpha and the SPARC V9, have combined the two, offering simple guarded instructions such as the *conditional move* instruction, while allowing for dynamic branch prediction. The presence of both guarded and speculative execution creates 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 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 guarded execution has a lot of 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 guarded execution and dynamic branch prediction. The third purpose is to propose 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 guarded execution. Section 4 presents quantitative results assessing the ability of guarded execution to enhance opportunities for parallelism extraction, the potential overhead of guarded execution, and the interplay between guarded execution and dynamic branch prediction. In section 5 we present a new method of specifying guarded execution. Our proposed technique allows guarded execution, in its full form, to be integrated easily (with a few additional instructions) 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 6 presents concluding remarks.

2. Guarded Execution

A guarded instruction is a normal instruction augmented with a guard condition specifier. 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. Introducing guarded execution into scalar processors can be a very powerful concept; Figure 1 presents a small example. Figure 1(a) shows the C-code for the inner loop of the Cmppt function of the SPEC92 benchmark Eqntott. Figure 1(b) shows the corresponding MIPS-like assembly code. Figure 1(c) shows the same code using guarded instructions (if-conversion [1] 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 a conditional instruction is the condition register. Comparing Figures 1(b) and 1(c) we can see that 4 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.

Guarded execution (or simply *guarding*), for scalar processors was proposed by Hsu and Davidson [8, 9] 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, and guarding is used as a general technique to fill multiple branch delay slots. Hsu and Davidson assume that the instruction set supports guarded stores and guarded jumps and allows for guard conditions that are the conjunction or disjunction of two operands in 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 DAGs. 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.

Vector processors have long benefited from guarded execution. Here, vector masks are used to express (multiple) guard conditions [14]. Using these vector masks, loops with if-statements can be vectorized. Recently proposed VLIW machines, for example the Cydra-5 [3, 13], and the IBM VLIW machine [4], have also used guarded execution to facilitate the software pipelining of loops with conditional branch instructions.

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 [11-13]. 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.

With guarded execution, code for a scalar processor has fewer branches, larger basic blocks, and fewer 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 dynamic) is reduced, the number of instructions between mispredicted branches (or the window size) can increase. A larger window size provides more opportunities for 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 is

```
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
L2 slt     at,a0,a2
   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.

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 existing instruction sets, it is generally not possible to find a sufficient number of bits to explicitly specify an additional source operand for all instructions. (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 and the SPARC V9 architectures are prime examples, offering a *conditional move* (CMOVE) instruction.

The second problem is that guarding 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 because it has no prior knowledge of which instruction is guarded until the instruction is fetched and examined, and the corresponding condition (if any) is evaluated. After the condition evaluation, instructions from the not-traversed path are transformed into NOPs (in the earlier stages of the pipeline). 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 increase the overall execution time. Execution time can 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 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. 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 the instructions, can increase the register lifetime. A possible solution to this problem is to add a separate predicate register file [12, 13], 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.

Guarding also complicates the register renaming logic. When a guarded instruction is squashed, it does not generate its renamed result, and the squashing must be followed by a repair action to keep the mapping table(s) consistent. To simplify the control logic, one solution is for a guarded instruction to always generate a result (either the old or the new value of the destination register). However, this solution requires an additional read port in the register file, to read the old value of the destination operand.

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 [12] addressed some of these issues for statically scheduled machines (such as VLIW), taking in account mainly the basic block size and the execution frequency. 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 [15]. 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, although, in general, given sufficient resources, these branches will be executed in parallel with other useful computation.

We believe that guarded execution is very useful to ILP processors (and will confirm 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 disadvantages of guarded execution, which we have outlined above, *are not fundamental disadvantages 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. We used a trace-driven simulator that simulates the user-level 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 [7], 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.

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.

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

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.

Our first metric is the *basic block expansion*. This is the increase in the number of *useful* instructions between branches, due to guarding. In the basic block expansion we do not count any guarded instructions that are dynamically transformed into NOPs. The basic block expansion plus the basic block size gives the effective guarded block size, i.e. the number of useful instructions between branches after the if-conversion. The advantages of this metric are (i) it is highly correlated with parallelism that can be extracted [2, 6, 12] and (ii) it is dependent only on the program and the compiler and not on the underlying hardware implementation.

Our second metric is the *path expansion*. This is the number of instructions in a block that *do not* contribute to useful computation, because they are dynamically transformed into NOPS. This metric gives an indication of the additional instruction fetch bandwidth required during the program execution.

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 *useful* instructions 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].

In our guarded block formation we do not perform any loop unrolling, or function inlining. Should 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 [12] in two ways. First, we require that all branches internal to the block (except the last one) be eliminated by the if-conversion; a Hyperblock is allowed to contain multiple branches and exit points. We treat what would be a Hyperblock as a sequence of basic blocks and guarded blocks. Our metrics, namely the basic block expansion and the path expansion, 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.

[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. The effects of this limitation on our metrics were negligible.

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.

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-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, an 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 instructions 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.

One way to provide support for all instructions in a 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. In this method, the compiler must 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 for. A compact and efficient way to specify guarded instructions, that does not suffer from these limitations, will be described in section 5.

[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.

4. Evaluation of Guarding

4.1. Branch Elimination Potential and Overhead of Guarding

We first consider the branch elimination potential of guarding. Table 2 presents the percentage of conditional and unconditional branches that are eliminated from the (dynamic) instruction stream when full and restricted instruction set support for guarding is available. Table 2 also shows the percentage of conditional backward branches in each program. These branches correspond to loop back-arcs and are important, because if-conversion cannot eliminate any of them, unless the loops are unrolled. From Table 2 we see that full guarding is able to eliminate an average of about 31% of all (dynamic) conditional branches, and 35% of the (dynamic) unconditional branches. To obtain the overall percentage of eliminated branches, the individual percentages can be combined using a weighted average, using the conditional to unconditional branch ratio for weight (this ratio can be computed from Table 1; across all programs it is about 3.46 to 1). Across all programs, the overall percentage of eliminated branches is about 32% for full guarding. Restricted guarding is not as powerful as full guarding, for obvious reasons; on the average, it is able to eliminate only 14.76% of all conditional and only 2.34% of the unconditional branches. An exception is Eqntott, which spends most of its time in an inner loop that is amenable to restricted guarding.

Program	Loop Branch.	Full		Restricted	
		Cond.	Uncon.	Cond.	Uncon.
Compress	26.48	24.86	84.29	18.24	0.00
Eqntott	29.07	40.55	54.98	40.04	1.02
Espresso	38.08	16.76	29.03	10.17	1.17
Gcc-cc1	24.84	31.92	17.04	9.64	0.37
Sc	24.63	43.07	17.74	9.83	0.18
Sunbench	15.79	35.65	47.10	11.35	0.03
Supermips	5.03	50.69	19.36	17.15	0.60
Tektronix	16.83	37.53	41.60	17.08	7.48
Tex	25.09	12.80	24.03	5.99	1.00
Thissim	11.52	62.31	33.70	23.26	1.43
Tycho	18.28	15.64	33.84	7.10	1.31
Xlisp	27.03	13.64	14.33	13.87	14.14
Yacc	38.64	19.53	38.95	8.18	1.71
Arithmetic Mean	23.17	31.15	35.07	14.76	2.34

Table 2. Percent of dynamic branches eliminated by full and restricted guarding.

Figure 2 presents the basic block size (labeled ''BB size''), the basic block expansion (labeled ''BB expansion''), and the path expansion, 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. The average

basic block size is 4.82 instructions and the average basic block expansion is 2.51 instructions, corresponding to a 52% increase in the effective guarded block size. In most cases the basic block expansion is at least 25% of the basic block size; in one case (Thissim), the basic block expansion is larger than the basic block size (7.55 versus 6.55 instructions respectively). The basic block expansion, however, comes with a price — the path expansion. From Figure 2 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 full guarding to become widely used.

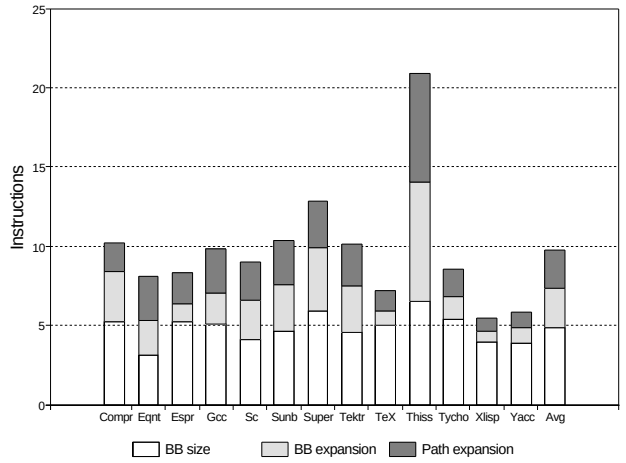

Figure 2. Basic block size, basic block expansion and path expansion with full instruction set support for guarding.

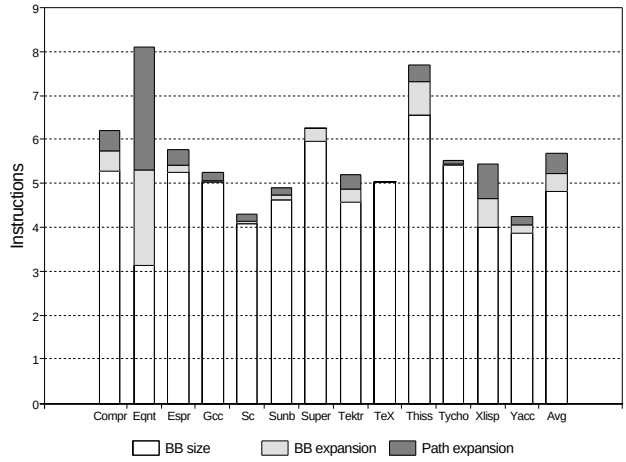

Figure 3. Basic block size, basic block expansion and path expansion with restricted instruction set support for guarding.

The magnitude of the path expansion is not as large with restricted guarding: on the average, only 8% of the instructions do not represent useful computation. However, restricted guarding also does not give a significant basic block expansion (an average increase of 8% over the basic block size).

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 (guarded) block size, assuming that its drawbacks can be dealt with. These results suggest that instruction set support for guarding is desirable, and that a limited support for guarded execution in the instruction set may not be sufficient.

4.2. Interaction of Guarding and Dynamic Branch Prediction

We now consider the interplay between guarding and dynamic branch prediction. 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.

We use two different prediction mechanisms: a 2-bit counter-based mechanism, and a GAp(8) pattern-based mechanism [16]. In either case, the predictor has 4K entries (which corresponds to 8K bits of storage). The two configurations were chosen to be reasonable in 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 the two predictors. The prediction accuracies do not include any unconditional branches, since these do not require dynamic prediction. Unfortunately, there are no (monotonic) trends in the table. In some cases guarding improves the prediction accuracy, in other cases it does not. It appears that both pattern-based and counter-based prediction mechanisms will continue to be beneficial even in the presence of guarding.

Next, we address the issue of dynamic window size. This size is influenced both by the prediction mechanism, as well as the number of useful instructions between branch predictions[3]. Figure 4 presents the dynamic

[3]It is interesting to compare these dynamic window sizes with the window sizes that can be established with trace scheduling, which uses static branch prediction [5].

window sizes without guarding (basic blocks), with restricted guarding, and with full guarding, for each of the benchmark programs using the above pattern-based predictor.

The pattern-based predictor can establish a respectable window size (about 156 instructions, on average), even without guard instructions. With restricted guarding, the average window size increases to about 184 instructions, and with full guarding, it further increases to about 258 instructions; for almost all programs the (average) window size is greater than 150 useful instructions.

Program	Counter based			Pattern Based		
	BB	Full	Restr.	BB	Full	Restr.
Compress	87.20	88.57	89.00	88.71	90.38	90.76
Eqntott	83.76	97.54	97.54	93.15	98.09	98.09
Espresso	90.30	92.11	89.55	96.56	96.84	96.20
Gcc-cc1	87.65	87.15	87.75	88.76	89.13	88.96
Sc	94.91	94.43	94.87	95.90	95.64	95.87
Sunbench	91.35	89.34	91.39	98.03	96.84	97.35
Supermips	96.53	97.77	95.81	96.00	96.35	95.58
Tektronix	91.15	89.28	90.24	96.01	95.95	96.25
TeX	94.72	95.20	94.50	94.80	95.12	94.76
Thissim	96.03	93.59	95.24	96.87	96.06	96.09
Tycho	93.41	94.30	93.24	95.16	96.34	95.24
Xlisp	88.16	87.20	87.40	95.21	95.37	95.33
Yacc	93.68	94.78	93.69	95.60	96.36	95.64
Arithmetic Mean	91.45	92.40	92.32	94.67	95.26	95.08

Table 3. Branch prediction accuracies for counter- and pattern-based predictors with 4K entries.

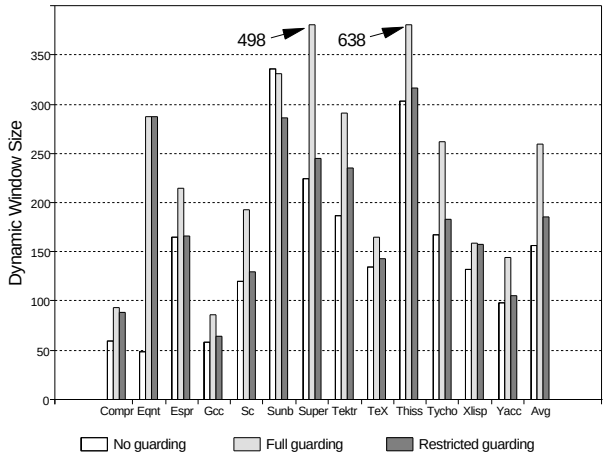

Figure 4. Effects of guarding on the window size for a pattern-based predictor with 4K entries.

One program, Sunbench, has an anomalous behavior: the window size with guarding is (slightly) smaller than the window size without guarding! The reason for this is that guarding changes the patterns of taken and not-taken branches. In the case of Sunbench, the new patterns are less predictable, resulting in the reduction of the window size.

5. 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, 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 instruction set. 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 knowledge of the type of 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. 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 propose the use of a ‘‘GUARD’’ instruction to specify a block of guarded instructions directly. A GUARD instruction has two operands, a register that specifies the guard condition, and a mask that specifies which of the following instructions (in the static and also dynamic code) are guarded by the specified condition. A GUARD instruction, therefore, provides a specification of the guard operands of each guarded instruction.

Figure 5 presents a small example illustrating the use of the GUARD instruction. The simple control flow graph in Figure 5 consists of four basic blocks forming two

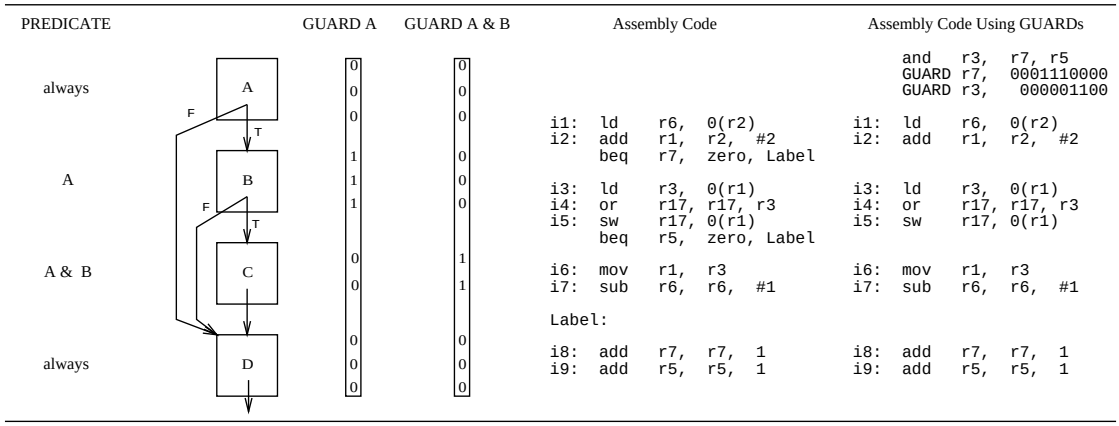

Figure 5. Example of the Use of the GUARD Instruction

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 5. In these masks, a 1 indicates that the corresponding instruction is guarded by the condition register, and a 0 indicates 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. The key advantage of the GUARD instruction is that it specifies the guard condition for many (subsequent) instructions. The benefits of this specification 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 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 5, 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 GUARD instructions, the processor has to maintain a mask of active and inactive instructions. This mask, called the *scalar mask*, is just a shift register: if the *i*-th bit in the shift register is 1, the *i*-th instruction 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:

$$scalar\_mask_i = scalar\_mask_i \ \&\ ((mask_i \ \&\ cond) \mid (!\ mask_i))$$

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 scalar mask is shifted by one position, with a one being shifted in.

An aggressive ILP processor, will need the ability to execute multiple GUARD instructions per cycle. Although the scalar mask is a centralized resource, the operations performed on it are very simple (three, 2-input gate levels). The generalization of the scalar mask update equation for 4 simultaneous GUARD instructions takes five, 2-input gate levels (using an AND-tree to combine all the results).

The scalar mask is key in permitting the processor to effectively squash unnecessary computation. The processor can identify the unnecessary computation by performing counting the leading zeros of 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 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 scalar 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. In our example of Figure 5, the conjunction of conditions *A* and *B* that guards basic block C is achieved

by setting the corresponding mask bits in both GUARD instructions. This is achieved by changing the first GUARD instruction to ''`GUARD r7, 0001111100`''. This simple change obviates the need for the AND instruction that was required to compute the condition *A* & *B*.

The scalar mask is part of the processor state, and has to be saved and restored on interrupts and context switches. The saved scalar 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 scalar mask. An alternative to exposing the scalar 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 scalar 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 [10].

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 whether the *i*th instruction following the GUARD instruction is guarded by the specified condition. Opcodes 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 5 and 6 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 4 presents the overhead of GUARD instructions to achieve full guarding (labeled ''Ovrhd''), and the average number of instructions guarded per GUARD instruction (labeled ''Instr per Guard''). Guarding using GUARD_TRUE and GUARD_FALSE instructions increases the dynamic instruction count by 13.9%, on average, guarding 5.1 instructions with every GUARD_TRUE and GUARD_FALSE instruction. When a GUARD_BOTH instruction is available, the dynamic instruction count is increased by 10% and each GUARD_BOTH instruction guards 7 instruction on the average. When all three types of instructions are available (labeled ''Combination''), the instruction count overhead is 8.6%, and on the average each GUARD instruction guards 7.9 instructions. In the results of Table 4 we did not attempt to optimize the use of GUARD instructions in our experiments. In particular, we did not evaluate use of the implicit AND property to reduce the number of condition setting instructions; we are experimenting with this optimization, and we expect that it will decrease the overhead of guarded execution even further.

Program	GUARD_TRUE/ GUARD_FALSE		GUARD_BOTH		Combination of all three	
	Ovrhd (%)	Instrs per Guard	Ovrhd (%)	Instrs per Guard	Ovrhd (%)	Instrs per Guard
Compress	12.3	4.7	10.8	5.4	9.0	6.5
Eqntott	39.2	2.3	29.5	3.1	22.5	4.1
Espresso	7.2	6.7	6.9	7.0	6.3	7.6
Gcc-cc1	11.3	6.0	7.9	8.6	7.0	9.6
Sc	17.4	4.2	14.9	4.9	12.9	5.7
Sunbench	18.8	4.0	9.7	7.8	8.8	8.6
Supermips	7.8	8.8	6.0	11.6	5.7	12.2
Tektronix	17.9	4.1	8.4	8.8	7.2	10.3
TeX	7.8	4.7	5.5	6.7	5.0	7.4
Thissim	11.4	8.9	9.1	11.2	7.4	13.7
Tycho	11.0	4.3	6.3	7.4	5.9	7.9
Xlisp	7.9	3.9	6.8	4.6	6.3	4.9
Yacc	10.7	3.8	8.7	4.7	7.8	5.3
Arithmetic Mean	13.9	5.1	10.0	7.0	8.6	7.9

Table 4. Overhead of the GUARD Instructions

6. 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, a quantitative assessment of the interaction between 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 suggests 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 258 instructions on average, using a pattern-based predictor with 4K entries. Without any form of guarding,

a pattern-based predictor could establish windows of only 156 instructions. 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 184 instructions on average, with a pattern-based predictor.

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 [12]). 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 10%, respectively and 8.6% when they are combined. We are carrying out more studies to reduce this overhead even further.

Acknowledgements

This work was supported in part by NSF grant CCR-9303030, and by ONR grant N00014-93-1-0465. We would like to thank Mark Hill and the reviewers for their helpful comments.

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*, pp. 177-189, June 1983.

[2] 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 Int’l Symposium on Computer Architecture*, pp. 266-275, May 1991.

[3] J. C. Dehnert, P. Y. T. Hsu, and J. P. Bratt, ‘‘Overlapped loop support in the Cydra-5,’’ *Proc. ASPLOS-III*, pp. 26-38, April 1989.

[4] 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 *et al*. North Holland, pp. 3-21, April 1988.

[5] J. A. Fisher and S. M. Freudenberger, ‘‘Predicting Conditional Branch Directions from Previous Runs of a Program,’’ *Proc. ASPLOS V*, pp. 85-95, October 1992.

[6] J. A. Fisher, ‘‘Trace Scheduling: A Technique for Global Microcode Compaction,’’ *IEEE Transactions on Computers*, vol. C-30, pp. 478-490, July 1981.

[7] M. D. Hill and A. J. Smith, ‘‘Evaluating Associativity in CPU Caches,’’ *IEEE Transactions On Computer*, vol. 38, December 1989.

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

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

[10] 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.

[11] Advanced RISC Machines, ‘‘ARM 610 RISC Processor,’’ *Document No: ARM DDI 004C*, January 1993.

[12] 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,’’ *Proc. of the 25th Annual Workshop on Microprogramming and Microarchitecture*, pp. 45-54, December 1992.

[13] 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.

[14] R. M. Russel, ‘‘The CRAY-1 Computer System,’’ *CACM*, vol. 21, pp. 63-72, January 1978.

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

[16] T. Yeh and Y. Patt, ‘‘A Comparison of Dynamic Branch Predictors that use Two Levels of Branch History,’’ *Proc. 20th Int’l Symposium on Computer Architecture*, pp. 257-266, May 1993.