Exploring the Benefit of GUARD Instructions

Dan Blandford and Takayuki Osogami
({dkb1, osogami}@cs.cmu.edu)

December 2, 2001

1 Abstract

We present a new method, called GUARD-conversion, for adding predication to existing instruction sets. GUARD-conversion is similar to the method used by Pnevmatikatos and Sohi [4], but is simpler to implement since it allows the direct replacement of branches with GUARD instructions. We discuss the details of implementing GUARD-conversion and the tradeoffs resulting from different levels of GUARD-conversion.

We evaluate the effects of GUARD-conversion on the performance of an instruction-level parallel processor (the SimpleScalar [1] out-of-order processor). We test the performance of five of the SPEC95 integer benchmarks using varying levels of GUARD-conversion. We observe that moderate levels of GUARD-conversion can decrease the number of branch mispredictions by an average of about 10%, though the decrease is as much as 27% for some benchmarks. We also observe that GUARD-conversion produces a small but noticeable improvement on the total execution time for most of the benchmarks we tested.

2 Introduction

Most advanced microprocessors achieve high performance by exploiting instruction level parallelism within basic blocks. Since the sizes of basic blocks are limited, it is important to go beyond basic blocks in order to extract further parallelism. Two policies are well known for this purpose: one is speculative execution and the other is guarded execution. In speculative execution, results of branch instructions are predicted, and instructions on the predicted path are executed. In guarded execution (also known as IF-conversion), control dependencies are converted to data dependencies in order to increase the effective size of each block.

Guarded execution was first proposed by Hsu and Davidson [2][3]. However, its usage was limited until a breakthough by Pnevmatikatos and Sohi [4], who pointed out the drawbacks of existing approaches for guarded execution, attributed these drawbacks to the methods of specifying guarded execution, and proposed a new method of specifying guarded execution. They proposed the use of a "GUARD instruction" that, using bit masks, specifies later instructions which are converted into NOPs if a condition is not satisfied. This method is decribed in detail in section 3 as "GUARD with bitmasks".

In this paper, we will propose a new method of specifying guarded execution, called "GUARD conversion". Our method is a simplification of the idea of Pnevmatikatos and Sohi: rather than using bitmasks to specify the instructions that are to be guarded, we specify an offset which determines the number of instructions to be guarded. This allows us to directly replace branch instructions in assembly code with GUARD instructions. This is an improvement over the technique of Pnevmatikatos and Sohi since it does not require the use of special compare instructions to set up the guard condition.

We have evaluated GUARD conversion using the SimpleScalar [1] out-of-order simulator, modified for guarded execution. We show that GUARD conversion improves the performance of speculative execution under reasonable settings. While Pnevmatikatos and Sohi [4] evaluated the performance of their GUARD instruction only via indirect metrics such as the dynamic window size (the number of instructions between mispredictions), we have also evaluated via a direct metric, that is, the total simulation time.

The rest of the paper is organized as follows: In section 3, we describe existing and new methods of specifying guarded execution. Section 4 describes details of GUARD conversion and its implementation on the SimpleScalar simulator. We show the simulation settings and the results of our experiments in section 5. Section 6 concludes the paper.

3 The GUARD instruction

Here we present three methods, including one of our own creation, for performing IF-conversion to remove branch instructions from assembly code. Examples of each of these methods are shown in Figure 1.

Conditional Move. The first IF-conversion method we will consider uses a "conditional-move" (CMOV) instruction. The CMOV operation takes three parameters: a condition, a source, and a target. If the condition evaluates to "true", the operation copies the source into the target; otherwise, it does not modify the target. Many IF-blocks have the effect of performing some calculation and then assigning a value to a register. To remove the branch instruction from such an IF-block, the compiler writes code that will perform the calculation, then, if the original IF-condition evaluated to true, use a CMOV instruction to copy the result of the calculation into the target register.

One disadvantage of this method is that it requires the program to always perform the calculation, even if the result of that calculation is not used. This can cause problems if the calculation in question might produce an exception. The example in Figure 1 (bottom left), for example, might incorrectly produce an exception if `d` is an invalid address but `a` $= 0$. In practice, this means that

```
a = b + c          ADD B, C, A
if(a > 0)          BLEZ A, 1
   a = d[0]        LOAD A, D
e = f              ADD 0, F, E
```

```
ADD B, C, A        ADD B, C, A          ADD B, C, A
CMPLE A, 0, G      CMPLE A, 0, g1       GLEZ A, 1
LOAD H, D          GUARD g1, 10000...   LOAD A, D
CMOV G, H, A       LOAD A, D            ADD 0, F, E
ADD 0, F, E        ADD 0, F, E
```

Figure 1: Three methods for using IF-conversion to remove BRANCH instructions from assembly code. **Top**: The initial code, in pseudocode and assembly language. **Left**: The code after using CMOV for IF-conversion can have correctness problems if the LOAD causes an exception. **Center**: Pnevmatikos and Sohi use a GUARD instruction with a register and bitmask. **Right**: Our method of IF-conversion converts the BRANCH instruction directly to a GUARD instruction.

the CMOV method of IF-conversion is not applicable when the calculation in question has a possibility of producing an exception.

Another disadvantage of this method is that it requires the use of extra registers. This method requires one temporary register for the end result of the calculation, and it may require several additional registers to store intermediate results from the calculation. If the compiler does not have enough free registers, it may be unable to apply IF-conversion using this method.

GUARD with bitmasks. The second method, used by Pnevmatikatos and Sohi [4], defines a GUARD instruction which takes two parameters. The first parameter is the index of a special GUARD register. Their simulator has an array of 32 GUARD registers, each of which had a size of only one bit. The second parameter is a bitmask. The bitmask indicates which of the succeeding instructions are to be made conditional. If the specified GUARD register evaluates to "true", the instructions indicated by the bitmask are converted to NOPs. An example of the results of this method can be seen in Figure 1 (bottom center).

This method solves several of the problems with the CMOV instruction. Pnevmatikos and Sohi's GUARD instruction prevents the calculation in question from taking place if the GUARD condition evaluates to "true", so there is no danger of producing spurious exceptions. There is also no requirement for additional registers since the GUARD registers are separate from the main register bank. The use of this method does require an additional instruction to initialize one of the registers from the GUARD register bank.

Also, although the bitmask representation used by this method is powerful, it is not immediately clear how a compiler would exploit this power. The maximum size of the bitmask is about 21 (assuming a MIPS-like instruction format), so a

GUARD instruction can apply to at most 21 succeeding instructions. This can allow the compiler to perform IF-conversion on several branches at once with a single GUARD instruction—but only if there are several branches within 21 instructions that all use the same branch condition. It seems likely that, in most cases, the GUARD instruction will apply only to a block of instructions immediately following it.

GUARD conversion. In most cases, we expect that a GUARD instruction will apply only to a block of instructions immediately following it. If this is the case, it should not be necessary to use a bitmask to specify which instructions are guarded. Instead, we can simply specify the number of instructions to be guarded. This is exactly equivalent to the specification of a branch instruction: it specifies an offset indicating the number of instructions to be skipped if the condition holds true.

Using this idea, we propose a third method of IF-conversion: direct conversion of branch instructions to GUARD instructions. We can directly convert any branch instruction with a positive offset to a GUARD instruction in which the offset specifies the number of instructions to be guarded. For example, in Figure 1 (bottom right), the BLEZ instruction is replaced with a GLEZ instruction: "If $A \leq 0$, convert the next 1 instructions to NOPs."

This method has the advantage that it is extremely simple to implement. Branch instructions in most architectures already feature "hint" bits which indicate to the processor whether the branch is likely to be taken or not taken. We can make use of one of these hint bits to indicate whether the instruction should be treated as a branch or a GUARD instruction.

4 Implementation

SimpleScalar. We implemented our GUARD conversion method on the SimpleScalar out-of-order processor simulator [1]. SimpleScalar provides a detailed out-of-order simulator with support for branch prediction, and its instruction set is easy to modify.

The SimpleScalar simulator calculates the result of each instruction during the decode phase, updating all registers accordingly. To ensure that the simulation has the same timing as the processor it simulates, the simulator uses a complex system of input and output dependencies which determine when each instruction can be marked as "finished" and removed from the active instruction buffer. Thus, in adding GUARD instructions to the language, our primary concern was with the input and output dependencies produced by instructions that were guarded.

GUARD Dependencies. Under the standard SimpleScalar model, an instruction has an input dependency to any register that it reads and an output dependency to any register that it writes. Each instruction can have up to three input and two output dependencies. (No instruction in the language can

```
0x01  ADD A, B, C
0x02  GLEZ C, 3
0x03  LOAD D, E
0x04  LOAD F, G
0x05  ADD D, F, C
0x06  STORE C, F
```

Figure 2: Instructions 0x03 through 0x05 cannot execute until the GUARD instruction has had a chance to convert them to NOPs.

actually write to two different registers; instead, the two output dependencies are used to resolve LOAD and STORE operations.) When an instruction is decoded, its input dependencies are checked against the output dependencies of previous instructions. If any matches are found, the instruction is prevented from executing until the instructions on which it depends have been completed.

With the addition of GUARD instructions to the language there are some additional complications. Any instruction that is in danger of being converted to a NOP by a GUARD instruction has, in essence, a dependency on that GUARD instruction. For example, in Figure 2, none of instructions 0x03 through 0x05 can be executed until instruction 0x02 is executed.

To handle this in the simulator we introduced an extra register, the GUARD register. This register is never written to or read from; instead, it serves as a flag which allows us to tell the simulator about these GUARD dependencies. We label every GUARD instruction as having an output dependency to the GUARD register, and we label every guarded instruction as having an input dependency to that register. This requires that we allocate an extra input dependency slot for each instruction, but we do not need any extra output dependency slots since GUARD instructions previously did not have any output dependencies.

Other dependencies. There is another type of dependency that may cause a problem. When an unguarded instruction depends on a register that is written to by a guarded instruction, there may be some ambiguity regarding the instruction dependencies. For example, in Figure 3, instruction `0x05` depends on register `A`. If the GUARD condition is true (and instruction `0x04` is converted to a NOP), this means that instruction `0x05` depends on instruction `0x01`. If the GUARD condition is false, though, then instruction `0x05` depends instead on instruction `0x04`. (Of course, even if the GUARD condition is false, instruction `0x05` cannot execute until the GUARD instruction has finished, so without loss we can declare that instruction `0x05` depends on instruction `0x04` in either case.)

To resolve this ambiguity, we slightly alter our interpretation of the NOP instructions produced by our GUARD instructions. When converting an instruction to a NOP, we maintain the former instruction's output dependencies.

```
0x01  LOAD  A, B
0x02  ADD C, C, D
0x03  GLEZ D, 1      ??
0x04  ADD C, C, A
0x05  ADD A, E, F
```

```
0x01  LOAD  A, B
0x02  ADD C, C, D
0x03  GLEZ D, 1
0x04  ADD C, C, A
0x05  ADD A, E, F
```

```
0x01  LOAD  A, B
0x02  ADD C, C, D
0x03  GLEZ D, 1
0x04  NOP
0x05  ADD A, E, F
```

Figure 3: **Top:** Depending on the GUARD condition, instruction 0x05 might depend on instruction 0x04 or instruction 0x01. **Left:** If the GUARD condition is false, instruction 0x05 depends only on instruction 0x04. **Right:** If the GUARD condition is true, we use the NOP to form a dependence chain to ensure that the instructions execute in the proper order.

In addition, we give the newly-created NOP an input dependency to the same register to which it has an output dependency. In effect, we treat the instruction as a "MOV A, A" instruction that requires no time to execute.

New Instructions. The SimpleScalar instruction set contains eight types of branch instructions. For each type of branch instruction, we created a GUARD instruction with equivalent semantics. To introduce these GUARD instructions into precompiled executables, we used a preprocessing phase in the simulator itself. As the simulator loaded instructions into memory, our code converted some of the branch instructions into GUARD instructions. We converted only GUARD instructions that satisfied three requirements.

First, we required that a GUARD instruction not contain any branches (conditional or unconditional) in the region that it was guarding. The purpose of GUARD instructions was to avoid the need for branch prediction and speculation; if we use GUARD-conversion on one branch only to stumble into another branch in the guarded region, we might as well have applied prediction to the original branch.

Second, for simplicity, we required that a GUARD instruction not contain any other GUARD instructions in the region it was guarding. Our GUARD instructions were implemented using global variables, and keeping track of multiple GUARD instructions at once would add unnecessary complexity to our code.

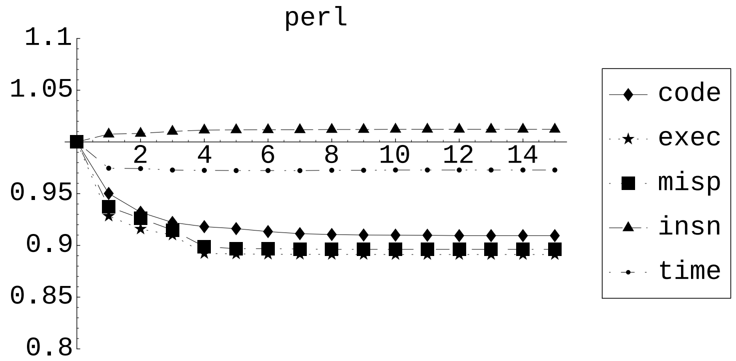

Figure 4: The effect of different levels of GUARD conversion on performance. Labels used: **code**: the number of unconverted branches in the program code. **exec**: the number of branches executed. **misp**: the number of mispredictions. **insn**: the number of instructions executed. **time**: The total simulation time.

Third, we required that any instruction which we converted have a positive offset which was less than some threshold MAX-GUARD. Converting a few instructions into NOPs is a small price to pay to avoid branch mispredictions, but converting hundreds or thousands of instructions into NOPs (rather than jumping over them with a branch instruction) would have seriously impacted our efficiency.

5 Experimentation

We tested our simulator on several of the SPEC95 integer benchmarks to see how much of an improvement the GUARD instruction could produce. We used a branch misprediction penalty of 12 cycles; for all other values we used the defaults provided by the SimpleScalar simulator. The default branch predictor was a standard bimodal table.

For each benchmark we tested various levels of GUARD conversion. To control the level of GUARD conversion we used a threshold variable MAX-GUARD. We converted only GUARD instructions that guarded at most MAX-GUARD instructions. Thus MAX-GUARD= 0 corresponded to converting no branches to GUARDs, whereas MAX-GUARD= 15 corresponded to converting all branches with an offset of 15 or less to GUARD instructions. Since we also used the restriction that no GUARD instruction could contain a branch or GUARD instruction in the set of instructions it guarded, in practice the maximum level of GUARD-conversion was attained with a MAX-GUARD of about 8.

For each test we performed, we recorded five statistics: the number of unconverted branches in the program code, the number of unconverted branches

	code	exec	misp	insn	time
cc1	91.6%	93.4%	92.3%	101.6%	99.1%
compress95	84.3%	97.9%	93.5%	100.3%	100.2%
ijpeg	88.6%	80.6%	73.2%	101.4%	95.4%
li	88.5%	98.1%	99.4%	100.4%	100.3%
perl	91.3%	89.1%	89.7%	101.1%	97.2%

Figure 5: The effect of using GUARD-conversion with MAX-GUARD= 6. Labels used: **code**: the number of unconverted branches in the program code. **exec**: the number of branches executed. **misp**: the number of mispredictions. **insn**: the number of instructions executed. **time**: The total simulation time.

encountered during execution, the number of branch mispredictions, the number of instructions committed, and the number of cycles the program required. We represented all statistics as percentages, with 100% being the value corresponding to no GUARD conversion.

The number of unconverted branches dropped by between 8% and 20% as we increased MAX-GUARD. Unfortunately, this did not always result in a corresponding drop in the number of branches encountered during execution. In three cases (cc1, ijpeg, and perl), the number of branches encountered during execution dropped proportionally to the number of branches converted to GUARD instructions in the code. We also observed a small improvement in the total execution time. In the other two cases (compress95 and li), however, the number of branches encountered during execution dropped by less than 3%. In those cases, conversion of branches to GUARDs had little effect on performance. We conclude that those benchmarks spent most of their time in tight loops which did not contain branches that could be converted to GUARDs. We didn't observe any improvement in the total execution time for these two cases.

In all cases, as we increased MAX-GUARD, the number of branch mispredictions encountered decreased while the number of instructions executed increased.

The effect of different levels of GUARD-conversion on one of the benchmarks we tested (perl) is shown in Figure 4. Graphs for the remaining benchmarks are shown in the Appendix, in Figures 6 and 7.

Based on the data we gathered from the benchmarks, we concluded that 6 was a good value for MAX-GUARD. In Figure 5 we present the exact effect of using GUARD-conversion with MAX-GUARD= 6 on the benchmarks we tested.

6 Conclusion

In this paper, we proposed the GUARD conversion, a new method of specifying guarded execution and explored its performance. Under reasonable settings, with a branch misprediction penalty of 12 and all other parameters set to the default SimpleScalar values, GUARD conversion decreased the number of branch mispredictions encountered by an average of 10%. The decrease in execution time, however, was 1.6% on average.

One reason for the insignificant difference in execution time is that the accuracy of branch prediction is high, and the cost of misprediction penalties is not a major part of the execution time spent by the processor. As a result, Amdahl's law indicates that we are unlikely to see much improvement.

6.1 Future Work

An interesting avenue for future work would be to combine GUARD conversion with branch prediction. Many branch predictors, in addition to a boolean prediction, can have some idea of their confidence in that prediction. If a predictor's confidence in its prediction is low, it would be interesting to allow that predictor to dynamically convert the branch instruction to a GUARD instruction.

In order to support this it would be necessary to examine all branch instructions in the code at compile-time. Any branch instruction which satisfied our conditions would be marked with a "could possibly convert to a GUARD instruction" bit. Most branch instructions contain extra bits that are used as branch-prediction hints anyway, so allocating a bit for this purpose would probably not be difficult. As a branch instruction was decoded, the branch predictor would simply need to check that bit to know if it could convert the branch to a GUARD.

References

[1] D. Berger and T. M. Austin. The simplescalar tool set, version 2.0. Technical Report 1342, University of Wisconsin, 1997.

[2] P. Y. T. Hsu. Highly Concurrent Scalar Processing. Ph.D. Thesis, University of Illinois at Urbana-Champaign, January 1986.

[3] P. Y. T. Hsu and E. S. Davidson. Highly Concurrent Scalar Processing. In *Proceedings of the 13th Annual Symposium on COmputer Architecture*, pages 386–395, June 1986.

[4] D. N. Pnevmatikatos and G. S. Sohi. Guarded execution and branch prediction in dynamic ilp processors. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 120–129, April 1994.

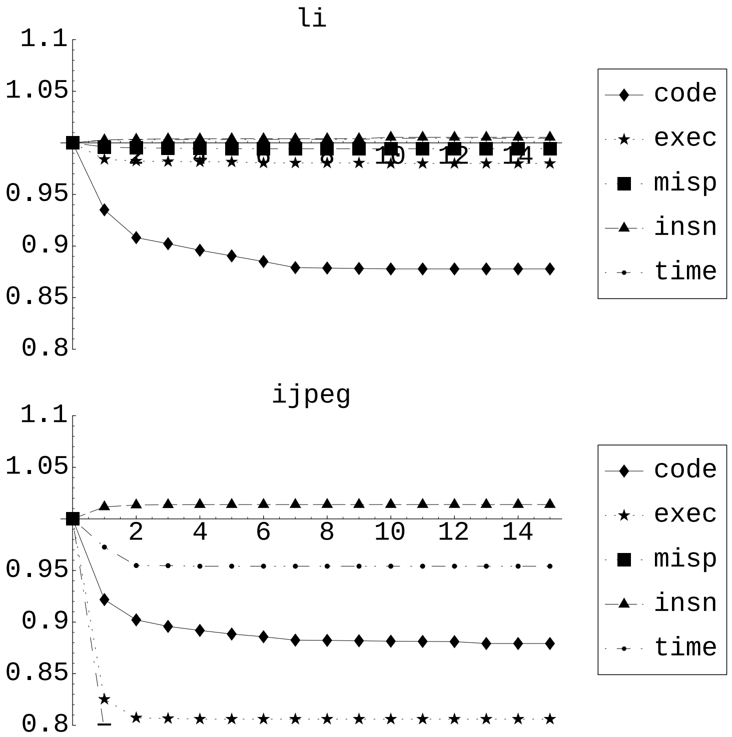

Figure 6: The effect of different levels of GUARD conversion on performance. Labels used: **code**: the number of unconverted branches in the program code. **exec**: the number of branches executed. **misp**: the number of mispredictions. **insn**: the number of instructions executed. **time**: The total simulation time. The misprediction level for the ijpeg benchmark drops to less than 75% for large values of MAX-GUARD.

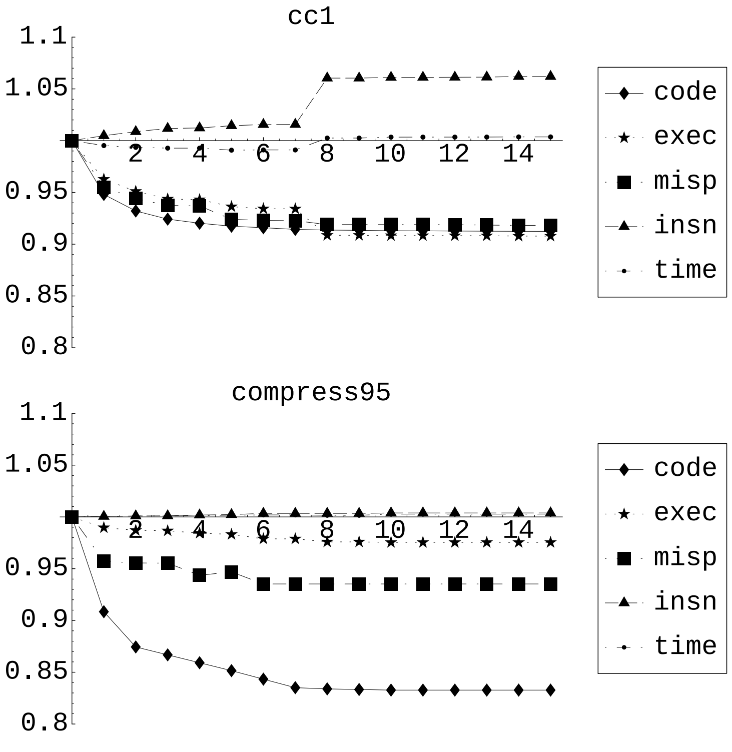

Figure 7: The effect of different levels of GUARD conversion on performance. Labels used: **code**: the number of unconverted branches in the program code. **exec**: the number of branches executed. **misp**: the number of mispredictions. **insn**: the number of instructions executed. **time**: The total simulation time.