Analyzing the Individual/Combined Effects of Speculative and Guarded Execution on a Superscalar Architecture

M. Srinivas
Silicon Graphics Inc.
2011 N. Shoreline Blvd
Mountain View, CA 94043
srinivas@mti.sgi.com

Alexandru Nicolau
Dept. of Computer Science
University of California
Irvine , CA 92715
nicolau@ics.uci.edu

Abstract

Speculative execution is a technique by which instructions are executed before the condition that controls it is evaluated. This can increase the performance if some of the idle cpu cycles are now used to execute speculated instructions. Guarded execution is a technique in which the branch instruction is eliminated and control dependences are converted to data dependences. This can help reduce some of the side-effects involved with branch instructions besides creating larger compilation units. However, excessive application of either one of them can result in dismal performance. Conventional approaches have used a one-time feedback metric and made all decisions based on it. We present a new way of designing feedback metrics and show how it can be used to regulate the effects of dynamic speculation and the side-effects of applying guarded execution statically. The proposed method presents a 0.3-0.6 fold improvements over a conventional scheme using SPEC benchmarks.

1. Introduction

For many non-numeric programs with branch intensive code, there is insufficient parallelism available within a basic block to fully exploit the parallelism available in a superscalar or super-pipelined processors. In order to improve performance the scheduling phase must be extended beyond basic block boundaries [7]. Priority should be laid to minimize the effects of branches in the code. Conventional techniques which attempt to parallelize the branch code (and alleviate this problem) mainly include *guarded* execution and *speculative* execution.

Guarded execution [2, 4, 11] is a technique in which the control dependences (usually in the form of branch instructions) are converted to data dependences (usually in the form of additional source operands). These additional source operands are referred to as predicate operands. The guarded instruction is executed conditionally depending on the value of this predicate operand. As a result, the original branch instruction is now replaced with guarded instructions. This optimization has several advantages. It eliminates any misprediction penalties that could have occurred with the prior branch instruction. It also increases the effective basic block size and increases the possibility of higher functional unit utilization between branch instructions.

Speculative execution refers to unconditional execution of instructions that were originally supposed to have executed conditionally. Sophisticated branch prediction (either static or dynamic) mechanisms are required to effectively utilize the benefits of speculated code and to undo the effects of speculatively executing instructions in the unpredicted path. It is similar to guarded execution except that instructions now can be executed much earlier than the condition controlling their execution is known.

Speculative execution can be enhanced either by software or hardware renaming. Software renaming involves replacing the destination register of the concerned instruction and storing its result into an additional register. This extra register can either be from the pool of free registers (at that time) or dedicated registers. The advantages of doing renaming at compile-time is that it can be coupled with other optimizations especially peephole optimizations like forward substitution, redundant load-store removal, strength reduction optimization etc. Many of the conventional software pipelining algorithms which move instructions past predicates use this technique [3], [8]. Hardware renaming (as is available in many modern superscalar architectures, eg. R10000 architecture [16, 14]) use a set of dedicated (or shadow) registers which store the speculative state of the machine. The visible architectural registers are then restored to the correct state from these shadow registers in case of any exceptions/misprediction. The benefit of hardware

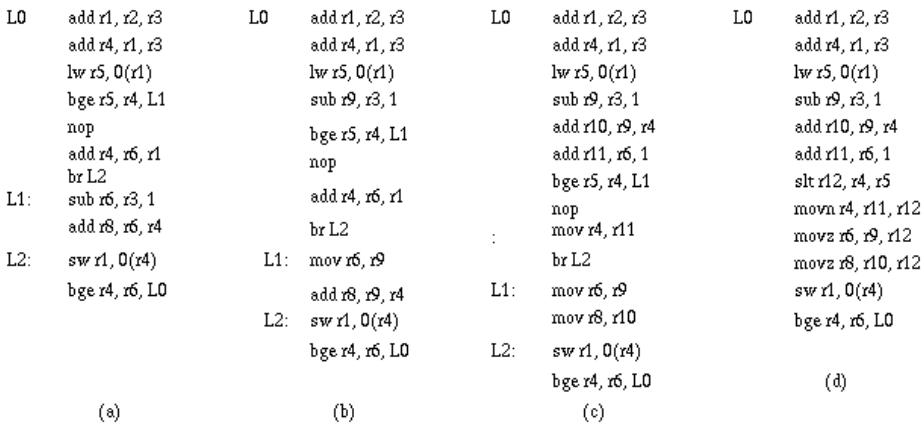

Figure 1: (a) Assembly code segment (b) After renaming and forward substituting sub r6,r3,1 instruction (c) After speculatively executing all instructions (d) After applying guarded execution.

renaming is that a more precise branch-prediction, exception info of the instruction can be determined and benefited from. However, the scope is usually limited and cannot be coupled with other optimizations if done at static/compile time. Figure 1(a) show a sample MIPS assembly code segment. Figure 1(b) shows the resultant code segment after speculatively executing instruction *sub r6,r3,1* past the branch condition. The register *r6* is renamed to r9 since it's live on the fall-thru path. A copy instruction *mov r6,r9* is inserted and is assigned the result of the renamed instruction.

Forward substitution is a technique in which all subsequent uses of the destination register of the copy instruction are replaced by its source register. This results in reduction of a true dependence between the copy instruction and any subsequent instruction which uses the result of the prior renamed instruction. In Figure 1(b), after speculatively executing instruction *sub r6,r3,1* and inserting the copy instruction *mov r6,r9*, all the subsequent uses of register *r6* (*add r8,r6,r4* as shown in Figure 1(a)) are now replaced with register *r9*. As a result of this, the add instruction can now move past the copy instruction without violating the program semantics. Figure 1(c) shows the result after speculatively executing all instructions.

Figure 1(d) shows the result after applying guarded execution. The control dependences originally present in the form of conditional branches are eliminated and now treated as data dependences (in the form of conditional moves). There are several advantages in applying this transformation. The effective basic block size is now increased thereby increasing the opportunities of applying conventional optimization techniques. The effective code size has decreased as a result of it. This also reduces the number of entries in the branch target buffer (BTB). Previous studies have shown that this may help in improving the overall dynamic branch prediction of the hardware since there are now less branch instructions which compete against each other [9, 5].

As shown in Figure 1(d), the combined effect of speculative and guarded execution can result in a better schedule. However, the inherent relationship between the two is intricate and if not understood correctly can affect each other adversely. The over-usage of guarded execution can adversely effect the applicability of speculative execution and vice-versa.

This paper attempts to understand these issues. Section 2 discusses previous work done in this direction. Section 3 describes the relationship between speculative and guarded execution and explains the compiler/architecture issues associated with them. In Section 4, we present a new approach of looking at feedback heuristics and its relationship with speculative/guarded execution. In Section 5, we present the implementation details, algorithm and the machine model used. Finally, Section 6 presents the evaluation methodology, the benchmarks used and provides a quantitative assessment of the technique. Throughout this paper, we will use R10000 architecture to illustrate our techniques, and as a target for our benchmarking in Section 6. However, the concepts are general enough to be adapted in other architectures supporting predicated and out-of-order (OOO) execution.

2. Previous Work

There have been many previous studies which attempted to individually study the effectiveness of guarded and speculative execution. Pnevmatikatos and Sohi [9] studied the benefits of applying guarded execution and its relationship with dynamic branch prediction. They proposed new synthetic instructions (eg. guard instructions) which effectively masked/unmasked the execution of instructions below it (over a pre-specified distance) using mask vectors.

With their approach, no additional source operands are required to be embedded in the instructions since the guard instructions can do the job without requiring additional support. Mahlke et al. [6, 5] proposed a mechanism of combining predicated with speculative execution. In their approach, basic blocks with hard to predict frequencies are coalesced (or if_converted) to form larger blocks (or hyperblocks). Speculative execution is then performed on the resultant hyperblocks by unconditionally executing some instructions that were originally guarded by a predicate specifier. Our previous work [13] attempted to reduce some of the overhead costs associated with guarded instructions by selectively employing predicated branch instructions. As a result, unnecessary overhead costs incurred with prior if-conversion at basic block level is now reduced to the instruction width level. The study was more tuned towards a VLIW machine supporting long instruction words and where the costs of performing guarded execution can be determined easily.

This paper extends previous work in several directions. Previous approaches made all decisions whether to if-convert (or not) based on a one-time feedback metric which usually averaged out the branch behavior for the entire loop iteration space. For example, a loop with a branch which has 50-50% execution behavior would have been if-converted even though the actual dynamic branch traces (of individual segments) would have behaved drastically different (eg. TTTFFFTTFF). In this paper, we take one step closer in refining the behavior of these non monotonic sections splitting them (if necessary) into several better predicted (or monotonic) sections. The splitting part is guided by several heuristics and can be used to regulate the effects of dynamic speculation and the side-effects of applying guarded execution statically. We also show how these split branch instructions can more effectively control different segments of the same loop such that portion of traces where branch behavior are predictable are never compromised. The motivation behind being that speculation can then be applied on some sections, while applying guarded execution on other sections. This is clearly not possible if we had either if-converted or based all decisions on a one-time feedback metric.

3. Interaction between Speculative and Guarded Execution

There exists a subtle but important relationship between speculative and guarded execution. Excessive application of one can critically affect the other. In this section, we attempt to address some of the issues relating the two using R10000 architecture as reference, wherever necessary. Most of the issues explained in this section helped in understanding the individual contributions of the two and led to better diagnosis of current feedback metrics.

Guarded execution necessitates the presence of additional registers (in our case the extra condition code registers) which can be used as operands in the instructions. This might complicate the register allocator as it now has to take into consideration extra registers. Also a clear demarcation of the different live ranges (i.e in the presence of conditional instructions) can be complicated task especially now that the register lifetimes are conditional. This can impede some of the decisions to speculatively execute an instruction. Most conservative assumptions need to be made unless a full-blown predicate analyzer is available to understand and treat different control paths differently. This can also impede the application of some of the compile-time optimizations (e.g. redundant code removal, possible removal of output dependencies etc.). Since most commercial processors (eg. R10000, DEC Alpha etc.) provide a limited predicated execution support, there is also an issue of providing a gamut of extra fictional operations to synthesize the full predicated execution support in the compiler. These fictional operations then need to be expanded to their equivalent non-fully predicated versions sometime before the final code layout phase. Issues like cycle times, operand properties, resource usage patterns etc. then need to be specially addressed for these operations. They also make the job of the scheduler hard, as it has to take into account hidden constraints (cycles etc.) in assigning priorities while scheduling. On the other hand, prior application of guarded execution (or if-conversion) can help some transformations. It has been proved that software pipelining is one such transformation which benefits from it [10, 15]. Prior application reduces messy control flow, makes the job of cyclic scheduler much easier, facilitates code motion among operations under different predicates simultaneously, and finally, reduces code explosion significantly (which would have otherwise not been possible due to inserted copies).

Modern out-of-order superscalar architectures do provide different forms of supporting dynamic speculation. The R10000, for example, uses the previous history outcomes of branches to decide which path to speculate from. The architecture also provides additional branch instructions for compiler writers and library designers to tag some branches with very high prediction accuracy at compile time. The *branch-likely* instruction is one such example. An example of a branch-likely instruction is *beql rs, rt, L1;* $delay_{slot}$ which means that the hardware counter will branch to L1 if *rs == rt*. However, the operation in $delay_{slot}$ is executed conditionally, i.e. the operation is executed only if the branch is taken else it's nullified. The branch likelies are always predicted taken, hence they don't have a specific history counter or an entry in the branch target buffer. Since, it's always predicted taken, the instructions following the target branch L1 are speculatively executed.

It is therefore debatable as to how much we would like

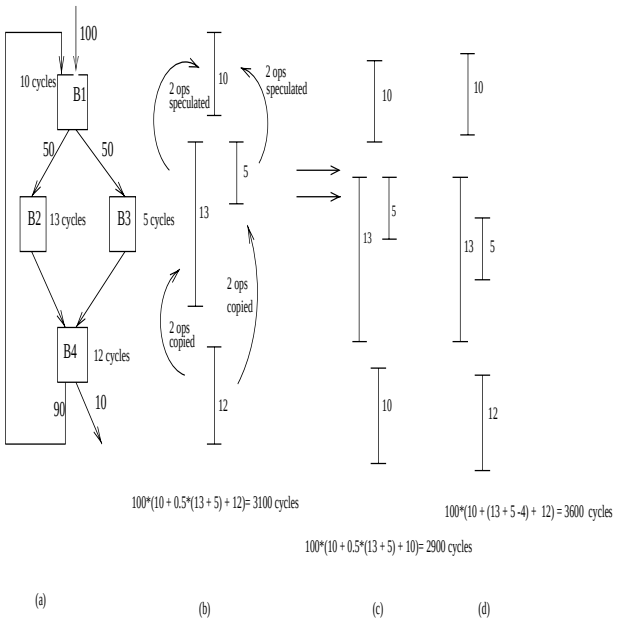

Figure 2: (a) A sample control flow graph (CFG). The annotations on the edges represent the execution frequencies. The annotations on the basic blocks represent the schedule lengths obtained using a local scheduler. Assume that block one has four vacant slots. (b) Shows the graphical representation of the schedules for the above CFG. Note that edges have been removed for simplicity. Assume that the loop is executed 100 times. (c) Shows the resultant schedule after speculatively executing two operations from basic block B2 and B3 to B1 and copying two operations from B4 to B2 and B3 respectively. (d) Shows the resultant schedule after applying guarded execution.

to perform speculation at compile-time versus doing it dynamically (with the added support of extra registers which cannot be used at compile-time and a better prediction accuracy). Either way, favoring predicated execution (if not used profitably) may force an added pressure on the limited general purpose integer and floating point register files. This may cause unnecessary register spilling and an increase in memory bandwidth.

The other problem associated with guarded execution is that it may result in an increase in the number of instructions that get executed dynamically. This may add up the total number of executed machine cycles and can worsen the existing performance if there is a scarcity of functional units. Guarded execution when applied to code with uneven schedule lengths may critically affect the one with the shorter length.

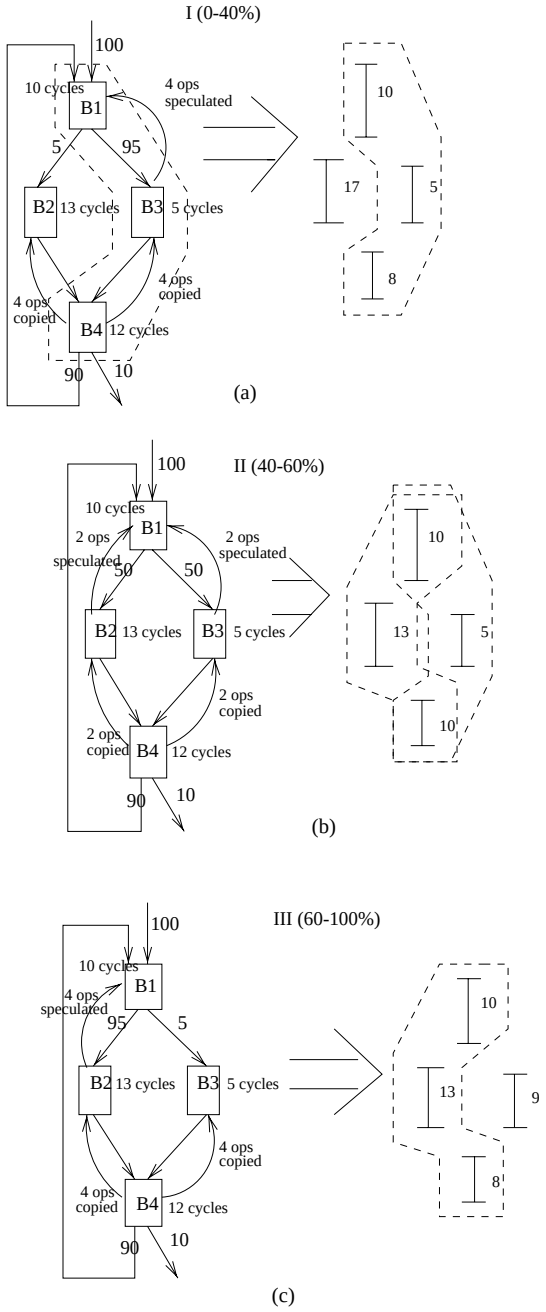

Figure 3: The example in Figure 2(a) is broken into three subgraphs (a), (b), (c) showing different code motion possibilities. The dotted lines show the most frequently executed trace(s) for that subgraph. (a) Shows the resultant schedule layout and blocks selection for the first 40% of the loop iteration space. (b) Shows the resultant schedule and block selection for the next 40%-60% of the loop iteration space. (c) Shows the resultant schedule and block selection for the final 40% of the loop iteration space.

4. Feedback Heuristics

Conventional approach to decide whether to if-convert the conditional branch (or region) is based on several factors. Most notably among them include frequency estimates, absence of any undesirable characteristics (like calls present inside the basic block) [6] etc. Our previous approach was based purely on program characteristics and not on any profiler estimates. We believe that the profiler or feedback heuristics can only alleviate the existing performance and can be easily embedded within the framework. The decision process was embedded within the region scheduler framework [1] and decision to if-convert was dependent on currently existent partial schedules and candidate block's characteristics. More details can be available on [13].

Consider the example fragment as shown in Figure 2(a). The annotations on the edges emanating from each basic block represents the execution frequencies. The annotations on the basic block represents the schedule lengths obtained using any local scheduler. Assume that the schedules on each basic block are tight except that from basic block B1 which has four available slots. Assume that the loop is executed 100 times. The acyclic schedule for this fragment is shown in Figure 2(b). The resultant schedule is 3100 cycles assuming that both the branches are equally taken. Given the fact that each path has an equal probability of being taken, two operations from basic block B2 and B3 are now speculatively executed into block B1. This in turn leaves two empty slots in each of them which is then filled by two operations copied from B4 to B2 and B3 respectively. This reduces the schedule of B4 by two cycles and the resultant schedule to 2900 (a net improvement of approximately 7%). The schedules are shown in Figure 2(c). Figure 2(d) shows the resultant cycles after performing guarded execution. Note that the overall schedule worsened as a result of applying guarded execution. It is therefore important to note that guarded execution should not be employed when the disparities between schedule lengths for two mutually exclusive paths are high (as shown in the above example), when the probabilities of exclusive paths taken ($50\% - 50\%$ in this case) do not compensate the cost involved in employing guarded execution etc. However, an alternate example can be shown where the resultant schedules can be improved as a result of applying guarded execution. But, the point to note is that either way, the approach to if-convert or not was based on a one-time feedback metric (static or profile) and based heavily on it. This may not be sufficient and a more precise information might be required.

Consider the same example as shown in Figure 2. Assume, for simplicity, that the first 40% of the loop iteration space, the true branch is taken 95%. The next $40\% - 60\%$ has a *toggle* effect where there is indecisive branch behavior and both the paths are equally taken. And, the final $60\% - 100\%$ of the loop iteration space, the false branch is taken 95%. The net effect is that both the branch paths are equally taken, yet, we see that the actual branch path behavior in the entire loop iteration space is quite different. The desirable effect would be to facilitate mechanism in which the operations from the true branch will be given more priority in the first 40% of the loop iteration space while giving operations in the false path more priority in the last 40% of the loop iteration space. This isn't clearly achievable if either we decide to if-convert or give equal priority in speculatively executing operations from both the paths (assuming that giving more priority to one can critically affect the other). This is a new concept and we believe that the performance effects of such a scheme can be tremendous.

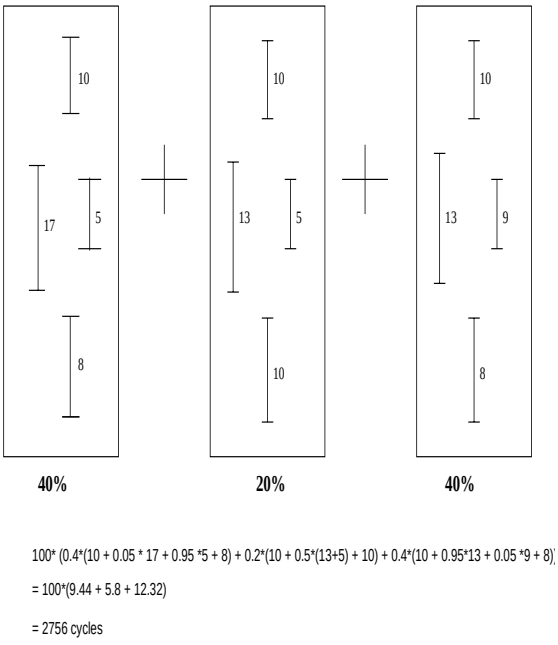

Figure 4: The $\%$ figures below the boxes show the percentage of the loop iteration space that these schedules comprise. The resultant schedule is the sum of the individual schedules (i.e. the three separate boxes)

The effect is illustrated in Figure 3. Figure 3(a) shows the equivalent control flow graph for the example in Figure 2 for the first 40$\%$ of the loop iteration space. The dotted lines show that block 1 => block 3 => block 4 is the most frequently executed trace. As a result, four operations are speculated from block 3 to block 1. This leaves opportunity for four operations to be code duplicated from block 4 to block 2 and 3 respectively. This degrades the schedule for block 2 but the belief is that since it's infrequently executed, the overall schedule shouldn't be worsened by much. Figure 3(b) shows the resultant graph for the next 40$\%$-60$\%$ of the loop iteration space. Since, both paths are equally

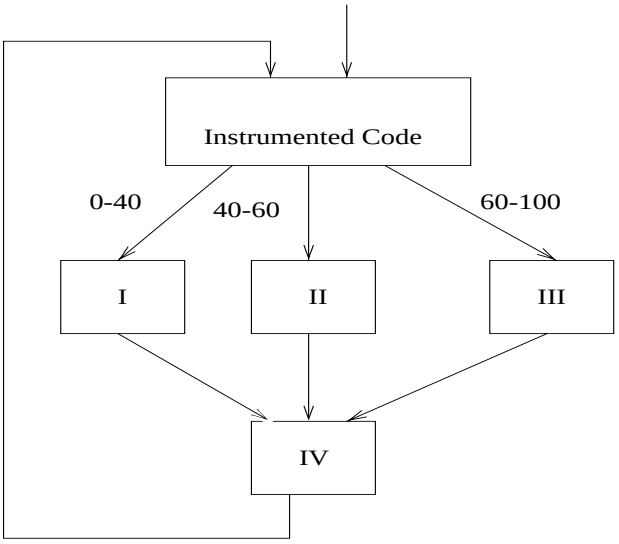

Figure 5: Split branch code added to combine the three different subsections (as shown in Figure 3(a),(b) and (c)) separately and benefit from.

taken, two operations each from block 2 and block 3 are speculated into block 1. As a result, two operations are now moved from block 4 to block 2 and 3 respectively. Figure 3(c) shows the execution trace for last 40% of the loop iteration space indicating that trace from block 1 => block 2 => block 4 is now more frequently executed. Operations are now speculated from block 2 resulting in a slightly degraded schedule for block 3.

Combining each schedule separately in Figure 3(a), (b) and (c) that we obtain by prioritizing each dynamic path separately times the percentage of the loop that each schedule is being executed gives the resultant schedule cycles as shown in Figure 4. This results in a further improvement of 6% over the schedule obtained by performing speculative execution assuming a one-time feedback metric. Figure 5 shows the schematic representation indicating the insertion of split branch code to combine the effects of the three different subsections (as shown in Figure 3(a),(b) and (c)) separately. Note, that we assumed strict conditions. In reality, the scope of improvements can be more. The schedules may not be as tight (as depicted in the example) leaving scope for further overlap, aggressively speculating instructions from the target branch onwards (i.e. executing instructions from the next iteration) and so on. At the same time, we didn't take into consideration costs involved in misprediction recovery, cache misses etc. The effects of branch misprediction and other OOO effects is beyond the scope of this paper.

```
for each procedure {
  detect all loops and create a loop-list L
  for each branch b_j in L do {
    if forward_branch {
      if branch_frequency(b_j) is highly probable (>0.95)
        generate branch likely instruction
      else if branch_frequency(b_j) <0.65 {
        if monotonic(b_j) and costs of guarded execution
        (ref Figure 2(d)) less expensive than weighted
        schedule estimates (ref Figure 2(b) and (c))
          generate if-converted code
        else if non-monotonic(b_j) and instrumentable(b_j)
          if costs of adding extra instrumented code
          (ref Figure 4) less expensive than either
          Figure 2(b),(c) and (d))
            generate split branch code (ref Figure 5)
      }
    }
    else if backward_branch {
      if branch_frequency(b_j) is highly probable (>0.95)
        generate branch likely instruction
  }
}
```

Figure 6: Algorithm

5 Implementation

In this section, we describe the algorithm, implementation details and the machine model used in the performance study.

Each loop is instrumented with additional feedback metrics which would tell the following factors, branch execution frequency, distribution (or classification) of loop iteration space into classes with similar branch execution behavior (i.e. greater, equal or less than a threshold value). The previous branch outcomes are recorded using bit vectors. The patterns are studied and then encoded as special (or instrumented) instructions to control (or regulate) the execution of branch instructions in a specific class. The branch-likely instructions are inserted to regulate control flow and give more priority to instruction traces for the portion of the loop execution where the probability (or profitability) of that instruction trace is very high. For, regions with anomolous (or less regular) branch behavior, the corresponding instruction traces are allowed to execute using the non branch-likely equivalent versions. We expect that the corrections will be made as recorded in the branch history table and the amount of hardware speculation will be as per the current prediction accuracy for that branch.

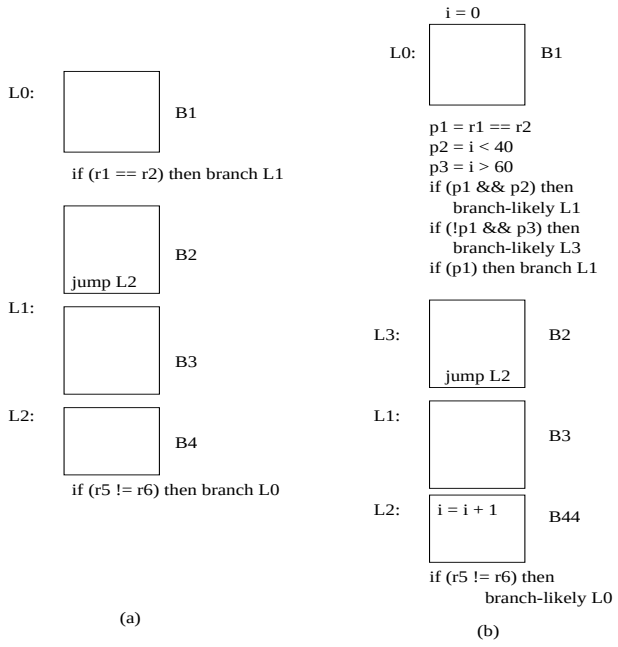

Figure 7: (a)Pseudo assembly segment for example shown in Figure 2(a) (b) Instrumented version of the same

The algorithm is presented in Figure 6. The branches are classified as either monotonic (or not) if their corresponding *toggle* factor (gathered from previous runs) is below/above a threshold limit. The *instrumentable* routine determines if the toggle patterns of this branch are periodic enough to be instrumented using algebraic counters. If the toggle patterns are complex enough (or do not follow any specific progression) then the branch is not considered as a candidate for splitting. Currently, the algorithm detects simple algebraic (or arithmetic) correlations in the toggle bit vector which can be expressed easily using unique counters. The algorithm can be extended to handle more complex correlations and will be the focus of future study.

The pseudo assembly fragment in Figure 7(a)) for the example in Figure 2(a) illustrates the effect. We see a forward branch to label L1 and a backward branch to label L0. Continuing with the previous assumptions about the loop and the branch behavior, instrumented code is inserted as shown in Figure 7(b). We see that extra predicates (*p1*,*p2*,*p3*), counter *i* and branch likely instructions are inserted to exploit the most profitable traces within the sections of the loop where the branch execution behavior is very regular. On the other hand, sections of the loop where the branch execution behavior is less regular, non branch-likely versions are used. In this example, we chose to execute the schedules (as shown in Figure 2(b)) for the sections of the loop where the branch behavior was anomolous (i.e between 40% and 60% of the loop iteration space). However, in theory, we can choose to execute the guarded (or if-converted) versions as well. We chose not, since the penalties for if-converted code (in this example) are high.

6. Performance Evaluation

Our study included benchmarks from the SPEC, splash and unix utilities. The underlying architectural model is the MIPS R10000 superscalar processor. The R10000 processor can issue up to 4 instructions and has a separate on-chip 32-KB instruction and 32-KB data cache. The chip provides two arithmetic logic units (ALU), three floating-point units and an address- calculation unit. The three floating-point units perform the functions of an adder, multiplier and divide/square-root respectively. Two separate integer and floating point register files feed into each of the three (integer and floating point) units separately. The register files comprises of 64 registers each 64-bits wide, out-of-which only 32 registers are visible to the external user. The chip uses the other 32 registers for its internal use which includes supporting the speculative state of the machine, performing out-of-order execution etc. The FP queue (consisting of 16 entries) feeds into the FP register file while the address queue (16 entries) and integer queue (16 entries) feed into the integer register file. The chip allows an in-order fetch and dispatch of up to four instructions per cycle. The branch prediction table is a 512-entry, 2-bit buffer which maintains the four different states (strongly taken, strongly not-taken, weakly taken, weakly not-taken) of the previous branch outcomes.

The benchmarks studied were *espresso, grep, compress and xlisp*. The benchmark programs were initially compiled by the GNU compiler and pre-processed to generate MIPS-like intermediate code. The intermediate code is then instrumented with feedback information and the resultant MIPS- binary is then fed to the superscalar simulator [12] and R10000 like- architectural options are then studied. The resultant performance summaries are generated only when the underlying benchmark runs to completion successfully. It's assumed and carefully considered that no inputs would cause any undesirable traps.

Table 1 shows the execution characteristics of the benchmarks studied. The columns show the percentage composition of the branches in the program and the taken/non-taken branch execution characteristics of each one of them. The % ratios of the branches in the program are measured as a ratio of the number of branches executed to the total dynamic instruction stream executed. The branch prediction accuracy for most of the benchmarks are in the mid-high nineties range. This is a strong indication to suggest that a good intermix of branch- likelies (in most frequently executed traces) and guarded execution (where instructions traces are less regular but suffer from insufficient parallelism) can have a tremendous effect in the dynamic performance.

Table 1: Benchmark characteristics

Benchmarks	Dynamic Instructions (millions)	Branch Instructions (%)	Correctly predicted branches (%)
Compress	0.41	20.81	91.98
Espresso	786.58	19.26	94.57
Xlisp	5256.53	23.12	89.21
Grep	0.31	22.28	92.0

Table 2 shows the latencies assumed in the study. The branch prediction table is assumed to be 512-entry, 2-bit buffer which maintains the four previous states of branch outcomes. Table 3 and Table 4 present the reservation station and functional unit usage summary for different schemes. The three different schemes studied in this work include *2-bit* branch prediction (as currently used in the R10000 architecture), the proposed approach of combining branch splitting with guarded execution (point to note is that this approach is in addition to the 2-bit prediction scheme) and *perfect* branch prediction. The last scheme is mainly for theoretical purposes and is used mainly to evaluate the performance of the other two schemes. The *perfect* branch prediction is not 100% BTB hit ratio. The branch target buffer is limited in size and can only store the history information for branch instructions whose target addresses have absolute value. Other types of branch instructions which include subroutine calls, returns and register-relative jumps (used in the context of switch statements) cannot be registered in the BTB. However, with the perfect prediction scheme, the remaining branch instructions are also predicted correctly.

In Table 3, the sub columns *BR, LDST, ALU* represent the reservation buffers for the branch, load/store and integer alu unit respectively. We didn't show the respective FP counterparts since they weren't relevant for the integer benchmarks. As we see in the later two columns, the % usage of BR buffers is order of magnitude higher than the 2-bit prediction scheme. However, the % usage of buffers for column two is still much less than the perfect scheme. This can be attributed to the additional stalls in the pipeline whenever a non-absolute branch instruction is encountered and the cycles for mis-prediction recovery. However, the % times the buffers are full is not a good indication to suggest performance. In the case of perfect branch prediction scheme, the next branch and all the succeeding instructions (including branches) are decoded and waiting pending issue of dependent instructions. This may deteroriate the throughput as shown in the Table 4. The *perfect* prediction scheme with much higher % of reservation buffers being full resulted in only a slight increase in the final IPC figures.

Table 4 shows the respective functional unit usage summary and the IPC (instructions per cycle) figures for the above mentioned schemes. As is evident (notably in *espresso* and *grep*), columns two and three have a better functional unit usage summary resulting in an overall improvement in the IPC (average of a factor of 1.5-2.0) when compared to those generated using the native *gcc* compiler. The *compress* benchmark showed tremendous improvements of performance. It had several nested branches with minimal code interspersed between them (especially the *main* routine). The resultant figures for compress are obtained when used to compress the superscalar tool (a total of 0.3 MG in memory) used in this study.

Table 2: Latencies

Instruction	Latency
alu	1
ld/st	2
sft	1
fp add	3
fp mul	3
fp div	3
cache miss penalty	6

7. Conclusions

This paper described a general approach of combining guarded with speculative execution for dynamic superscalar processors. We showed that conventional mechanisms of deciding when to speculate or apply guarded execution are less accurate and presented a finer- grain representation of diagnosing feedback metrics and showed how this can be used to regulate the effects of dynamic speculation and side-effects of applying guarded execution statically. We then studied the effects on a R10000-like architecture and showed how its existing 2-bit branch prediction scheme when combined with more accurate methods of aiding hardware speculation coupled with guarded execution can result in an order of (0.3-0.6 fold) improvements in dynamic performance.

We feel that this study opens up a new area of research

Table 3: Reservation Station Usage Summary

Benchmarks	$2-bitBP^1$			$ProposedApproach^2$			$PerfectBP^3$		
	BR^4	$LDST^5$	ALU^6	BR	LDST	ALU	BR	LDST	ALU
Compress	13.91	1.52	0	44.47	3.15	0	64.8	4.11	0
Espresso	9.05	0	0	57.9	0.82	2.4e-04	64.8	0.04	0
Xlisp	13.67	7.59e-07	0	48.2	3.2e-05	0	67.6	5.6e-04	0
Grep	13.75	0.02	0	53.28	0.03	0	69.21	0.05	0

1= using 2-bit branch prediction scheme
2= using the combined approach in addition to 2-bit branch prediction
3= using perfect branch prediction without the combined approach
4= % times branch reservation unit is full (ratio to the final commit cycle)
5= % times load/store reservation unit is full (ratio to the final commit cycle)
6= % times alu reservation unit is full (ratio to the final commit cycle)

Table 4: Functional Unit Usage Summary and IPC

Benchmarks	$2-bitBP^1$				$ProposedApproach^2$				$PerfectBP^3$			
	ALU^4	$LDST^5$	SFT^6	IPC^7	ALU	LDST	SFT	IPC	ALU	LDST	SFT	IPC
Compress	0.74	5.44	0.93	0.63	2.76	8.86	1.53	1.16	4.76	11.91	1.98	1.51
Espresso	6.64	2.05	0.34	0.68	5.58	20.15	0.70	1.36	6.64	22.8	0.78	1.53
Xlisp	0.12	7.39	5.73e-05	0.61	0.37	17.2	5.9e-05	0.98	0.65	23.1	9.79e-05	1.33
Grep	0.41	4.97	0.76	0.64	2.25	8.98	1.02	1.25	3.85	10.44	1.19	1.49

1= using 2-bit branch prediction scheme
2= using the combined approach in addition to 2-bit branch prediction
3= using perfect branch prediction without the combined approach
4= % times alu unit is full (ratio to the final commit cycle)
5= % times load/store unit is full (ratio to the final commit cycle)
6= % times shifter unit is full (ratio to the final commit cycle)
7= instructions per cycle (excluding annulled)

focus and the work presents preliminary results in that direction.

8. Acknowledgments

The authors appreciate the comments of anonymous reviewers which improved the quality of the paper. This work was supported in part by NSF grant CCR8704367 and ONR grant N0001486K0215.

References

[1] V. H. Allan, J. Janardhan, R. Lee, and M. Srinivas. Enhanced Region Scheduling on a Program Dependence Graph. In *Proceedings of the 25th International Symposium and Workshop on Microarchitecture (MICRO-25)*, Portland, OR, December 1-4, 1992.

[2] J. Dehnert, P. Hsu, and J. Bratt. Overlapped loop support in the Cydra5. In *Proceedings of the 16th Annual International Symposium on Computer Microarchitecture*, 1989.

[3] K. Ebcioğlu and T. Nakatani. A New Compilation Technique for Parallelizing Loops with Unpredictable Branches on a VLIW Architecture. In D. Gelernter, editor, *Languages and Compilers for Parallel Computing*, pages 213–229. MIT Press, Cambridge, MA, 1990.

[4] P. Hsu and E. Davidson. Highly concurrent scalar processing. In *Proceedings of the 13th Annual International Symposium on Computer Microarchitecture*, 1986.

[5] S. A. Mahlke, R. E. Hank, R. A. Bringmann, J. Gyllenhaal, D. Gallagher, and W.W.Hwu. Characterizing rge impact of predicated execution on branch prediction. In *Proceedings of the 25'th Annual International Symposium on Microarchitecture*, December 1994.

[6] S. A. Mahlke, D. C. lin, W. Y. Chen, R. E. Hank, and R. A. Bringmann. Effective compiler support for predicated execution using the hyperblock. In *Proceedings of the 25'th Annual International Symposium on Microarchitecture*, December 1992.

[7] A. Nicolau and J. A. Fisher. Measuring the available parallelism for very long instruction word architectures. *IEEE-TC*, C-33:1088–1098, Sep 1984.

[8] A. Nicolau, R. Potasman, and H. Wang. Register Allocation, Renaming and Their Impact on Fine-Grain Parallelism. In *Proceedings of the 4'th International Workshop on Languages and Compilers for Parallel Processing*, Santa Clara, CA, August 1991.

[9] D. Pnevmatikatos and G. Sohi. Guarded Execution and dynamic branch prediction in dynamic ILP processors. In *Proceedings of the 16th Annual International Symposium on Computer Microarchitecture*, 1994.

[10] B. R. Rau, M. S. Schlansker, and P. Tirumalai. Code Generation Schema for Modulo Scheduled Loops. In *Proceedings of Micro-25, The 25th Annual International Symposium on Microarchitecture*, December 1992.

[11] B. R. Rau, D. W. L. Yen, W. Yen, and R. A. Towle. The Cydra 5 Departmental Supercomputer: Design Philosophies, Decisions, and Trade-offs. *The IEEE* Computer, pages 12–25, January 1989.

[12] K. Shimura and H. Nishimoto. Introduction to Paratool Version 3.0. Technical Report Processor Laboratory Technical Report, Fujitsu Laboratories Ltd., 1994.

[13] M. Srinivas, A. Nicolau, and V. H. Allan. An Approach to Combine Predicated/Speculative Execution for Programs with Unpredictable Branches. In *Proceedings of the IFIP Transactions, Parallel Architectures and Compliation Techniques (PACT-94)*, Montreal, Canada, August 24-26, 1994.

[14] S. Turner, M. Zagha, B. Larson, and M. Itzkowitz. Performance Analysis Using the MIPS R10000 Performance Counters. In *International Conference on High Performance Computing and Communications*, Pittsburgh, PA, November 17-22, 1996.

[15] N. J. Warter, S. A. Mahlke, W. W. Hwu, and B. R. Rau. Reverse If-Conversion. In *Conference Record of SIGPLAN Programing Language and Design Implementation*, June 1993.

[16] C. Y. Yeager. The MIPS R10000 Superscalar Microprocessor. *IEEE Micro*, 5(2), April 1996.