Selective Guarded Execution Using Profiling on a Dynamically Scheduled Processor

Srinivas Mantripragada
Silicon Graphics Inc.
2011 N. Shoreline Blvd
Mountain View, CA 94043

Alexandru Nicolau
Dept. of Computer Science
University of California
Irvine, CA 92715

Abstract

Modern dynamically scheduled processors use branch prediction hardware to speculatively fetch and execute most likely executed paths in a program. Complex branch predictors have been proposed which attempt to identify these paths accurately such that the hardware can benefit from out-of-order (OOO) execution. Recent studies have shown that inspite of such complex prediction schemes, there still exist many frequently executed branches which are difficult to predict. Predicated execution has been proposed as an alternative technique to eliminate some of these branches in various forms ranging from a restrictive support to a full-blown support. We call the restrictive form of predicated execution as guarded execution.

In this paper, we propose a new algorithm which uses profiling and selectively performs if-conversion for architectures with guarded execution support. Branch profiling is used to gather the taken, non-taken and misprediction counts for every branch. This combined with block profiling is used to select paths which suffer from heavy mispredictions and are profitable to if-convert. Effects of three different selection criterias, namely size-based, predictability-based and profiled-based, on net cycle improvements, branch mispredictions and mis-speculated instructions are then studied. We also explain numerous adjustments that were made to the selection criterias to better reflect the OOO processor behavior.

1 Introduction

Conditional branches pose a serious problem in achieving parallelism from a program. They introduce *control dependencies* which imply that instructions that follow the branch cannot be executed before the branch condition is known. This form of dependence limits the parallelism extraction phases (static or dynamic schemes) to extract parallelism only between branch boundaries. Modern superscalar processors support dynamic scheduling with speculation as an alternative to alleviate this problem. Branch instructions are predicted using the prediction hardware and instructions from the predicted path are speculatively fetched and executed. If the prediction is correct, the hardware suffers no performance penalty. The speculated instructions can graduate earlier consuming resources that were otherwise idle. If the prediction is incorrect, the processor suffers from extra penalty required to bring the processor state to the correct dynamic instance of that branch.

Recent advancements in branch prediction technology has allowed for greater prediction accuracies by the hardware. The branch prediction outcomes is used by the hardware to speculate instructions from frequently executed path(s). Two-Level Adaptive [12], Gshare [10] and Hybrid predictor [8] are few examples of this approach. Inspite of recent advancements, there still exist many branches which are difficult to predict. With deeper pipelines these instructions can result in many wasted cycles incurred by the hardware when resolving branch mispredictions.

Predicated execution [1, 3, 9] is another architectural mechanism which can reduce the branch penalty. It helps eliminate branch instructions and converts control dependences into data dependences. The data dependences are usually in the form of additional source operands. The predicated instruction is executed conditionally depending on the value of this predicate operand. This eliminates any misprediction penalties that could have occurred with the prior branch instruction and increases the possibility of higher functional unit utilization between branch instructions. Previous studies have shown that selectively applying predicated execution on such ill-predicted branches can improve the overall branch prediction in the hardware [11, 4].

This paper proposes a new selective if-conversion algorithm using profiling on a dynamically scheduled processor supporting speculative execution with guarded execution support. The branch predictor used is the McFarling's combined (gshare + bimodal) branch predictor [15]. Profiling is done using this predictor to gather the dynamic execution characteristics for each branch instruction. This combined with block profiling is then used to select hammocks which are profitable to if-convert. Three different profitability selection criterias, namely size-based, predictability-based and profile-based are proposed and their effects on net cycle improvements, branch mispredictions and mis-speculated instructions are then studied. We also explain numerous adjustments that were made to these selection criterias to better reflect OOO processor behavior.

This paper is organized into 5 sections. Section 2 discusses previous work done in this direction. Section 3 describes the underlying guarded execution model and the profiling technique used. Section 4 explains the algorithm in greater detail and the different profitability selection criterias studied. Section 5 describes the simulation methodology used in our experiments, the benchmarks used and the results obtained from those experiments.

2 Previous Work

There have been many previous studies which attempt to combine the merits of guarded and speculative execution. Pnevmatikatos and Sohi [7] studied the effects of applying guarded execution on a superscalar architecture supporting OOO execution. They showed that by predicating as many branches as possible, the overall branch prediction on the hardware and succeedingly ILP can improve dramatically. Mahlke et al. [5] studied the effects of predicated execution on a VLIW (or statically scheduled) machine. Static estimates or profiling data derived from simple dynamic branch predictor are used to determine hard-to-predict branches which are then selectively if-converted [4]. Their experiments were done on a VLIW machine which doesn't support OOO execution, dynamic renaming and other features prevalant in modern superscalar processors. Tyson [11] studied the effects of predicated execution by predicating all short forward branches which were hard-to-predict. He, then approximated the reduction in branch misprediction penalty as a function of the processor issue and pipeline depth. Chang et al. [2] studied the effects of eliminating the hard-to-predict branches manually using the Two-Level Adaptive Branch Predictor and reported the results both in the number of branch mispredictions reduced and the net cycles saved.

In this paper, we extend previous work and make three new contributions. First, we present a new algorithm which selectively employs if-conversion on a dynamically scheduled processor with speculative execution and guarded execution support. Branch profiling is done to gather the hard-to-predict branches using the combined predictor. Most of previous work [7, 11, 2] decision to eliminate hard-to-predict branches has been at the granularity of the single branch instruction. In addition to the branch predictability criteria, we feel that there are other factors, eg. schedule estimates, block frequencies, block size, misprediction penalties etc. which are equally important. The proposed algorithm attempts to combine these criterias along with the branch predictability factor to selectively if-convert hammock branches. Second, it attempts to provide a deeper understanding of the effects of guarded execution on dynamic speculation. Three different profitability selection criterias, namely size-based, predictability-based and profile-based methods are proposed and their effects on branch mispredictions, mis-speculated instructions and net cycle improvements are then studied. Finally, we propose several adjustments to these selection criterias to better reflect OOO processor behavior. We feel that static estimates are not sufficient enough to decide profitability decisions.

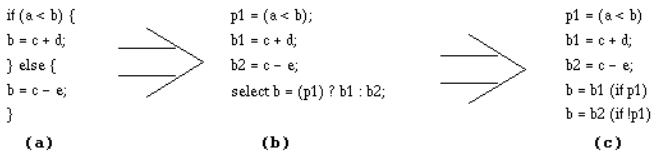

Figure 1: **Guarded Execution Example and Insertion of Select Instructions.**

3 Guarded Execution

3.1 Model Assumed

Our guarded execution model benefits from the usage of select instructions. A select instruction is of the form:

$$result = select(condition)?operand1 : operand2 \quad (1)$$

The condition is the predicate operand and is the result of the compare instruction. If the condition is true, result gets the value of operand1 else gets the value of operand2. The advantage of this representation is that multiple definitions of the same variable (or register) get a unique value; the select instruction is then used to pick one of these several values. The scheme nicely fits with the dynamic renaming scheme simulated by the machine [14] where each logical register is renamed to produce a unique value eliminating anti- and output- dependences. Figure 1 shows an example where the two definitions of the variable b under the if- and else- clause are renamed to unique temporaries, *b1, b2*; the select instruction then picks one from the two based on the predicate condition *p1*. The select instruction is then finally transformed to the machine instructions (in the form of conditional moves) as shown in Figure 1(c). The disadvantage of this approach is that the instructions which get converted to predicate form need to be first guaranteed to be safe speculatable, i.e cause no traps, divide-by-zero errors etc. This can sometimes limits the application of the technique.

3.2 Branch and Block Profiling

Profiling is initially done to gather program characteristics. The *pixie* tool is used to gather basic block counts. Block frequency is computed for each block by dividing its block count with that of its immediate dominator block. The procedure begin and the loop entry block frequency is always normalized to one, realistically the loop-entry block gets executed more often than the procedure entry block. This shouldn't affect the selectability criteria because most of the hammock regions which get selected never cross loop-boundaries, i.e don't have internal loop cycles.

The branch predictor simulated by the profiler is the McFarling combined (gshare + bimodal) branch predictor. The prediction outcome of each branch instruction is gathered and compared with the actual branch direction. A table of counters is maintained indexed by the branch address which gets incremented whenever there is a misprediction sequence for that branch. The accumulated value determines the number of times the branch has been mispredicted for the entire duration of the program.

4 Algorithm

The algorithm is shown in Figure 2. Currently, the algorithm identifies hammocks (simple or complex) in the program and if-converts them. All the loops in the region are first detected and a loop-list is created. An initial local scheduling pass is then invoked which is a list-driven single-block scheduler. The local scheduling pass is invoked to gather initial block (and parallelism) estimates. The main driver then traverses the loop-list starting from the innermost loop. The *Hammock Selector* identifies all possible and legal hammocks in the loop and presents an order of processing them. The default order is to process the hammocks in bottom-up fashion starting from the innermost-ifs. This presents an order where an if-converted inner-if region can be retried again as part of an outer nested-if region.

```
FOREACH procedure (region) {
  Detect all loops and create a loop list L inside this region.
  Do an initial local scheduling pass.
  WHILE the loop list L is not empty {
    Detect all hammock subgraphs HS in L.
    Sort/Prune HS using the hammock_selector.
    WHILE the hammock_list is not empty {
    (The processing order is bottom-up starting from
    the innermost hammocks).
      IF profitable_to_predicate HS_i {
        Delete all existing blocks in HS_i
        Create a new block B_j in HS_i
        Insert all the instructions in B_j
        (unique temporaries are generated for instructions
        which have been speculated.)
        Insert appropriate select instructions.
        Update the unprocessed hammock_list with the
        new identifier B_j.
      }
    }
  }
}
```

Figure 2: **Hammock Selection Algorithm**

4.1 Hammock Selection and If-Conversion

Both legal and profitability constraints are taken into account during hammock selection. With respect to legal constraints certain hammocks are not processed. They include hammocks which are part of any exception labels, contain side-entrances or contain any instruction which cannot be converted into safe-speculatable form. Once the hammocks are selected, the equations shown below are computed and analyzed.

Assuming that *A,B and C* are basic blocks with p being the probability that A branches to B and *1 - p* being the probability that A branches to C. Assume that $[B]$ represents the size (i.e number of instructions) and $||B||$ represents the schedule estimate derived from the local scheduler for block B.

$$LHS\_size = [B] \quad (2)$$
$$RHS\_size = [C] \quad (3)$$
$$LHS\_cycles = p * |B| \quad (4)$$
$$RHS\_cycles = (1-p) * |C| \quad (5)$$
$$Branch\_cost = Branch\_fixed + Branch\_taken + mispredictions * mispredict\_penalty \quad (6)$$
$$Total\_cycles(CFG) = |A| + LHS\_cycles$$

$$+RHS\_cycles + Branch\_cost \quad (7)$$

$$Total\_cycles(predicated) = |A + B + C| \quad (8)$$

The LHS_size (annotated for left-hand side size) is the size of block B. The RHS_size (annotated for right-hand side size) is the size of block C. The LHS_cycles is computed as a weighted schedule estimate, i.e. probability p times the schedule estimate of block B. The RHS_cycles is computed as product of (1 - p) times the schedule estimate for block C. The branch_cost is the cummulative sum of the branch fixed cost, taken cost and the product of the number of mispredictions times the mispredict penalty. The total_cycles for the original CFG is the cummulative sum of the schedule estimates for block A, the LHS_cycles, the RHS_cycles and the branch_cost. The total_cycles for predicated CFG is the schedule estimate for the block with all the instructions originally from block A, B and C appended along with the inserted select instructions.

Once a hammock region is selected as a profitable candidate for if-conversion, the following steps are sequenced. (1) A new block is created with all its control predecessors and successors correctly built. (2) All the existing instructions (either in original form or in speculatable form) are appended into the new block. Unique temporaries are created for each instruction which has been speculated. Appropriate select instructions are inserted which act as gate-keepers and select one among the several reaching definitions of the same variable into the original destined register. The destination register updation part is controlled by the predicate operand. (3) All the original blocks are now deleted. (4) The unprocessed hammock_list is now updated with the new hammock identifier so as to allow this if-converted region to be processed again as part of an outer (or intersecting) hammock region.

4.2 Profitability Criterias

In this section, we describe the different profitability criterias that were used to decide whether to if-convert a hammock region.

4.2.1 Size Based

$$LHS\_size + RHS\_size <= threshold\_value_{size} \quad (9)$$

In this selection criteria, all hammocks which fit the above equation are considered as profitable candidates. A *threshold_value* of 8 instructions is used in the experiment. This threshold_value doesn't include any of the extra select instructions that may get added. The idea behind this selection method is to catch all hammock regions which are fairly small in size, and by if-converting them, can improve the fetch bandwith and parallelism between branch instructions. Usually, with a fetch of 4-5 instructions per cycle, the whole hammock can be fetched in a cycle or two. This can also reduce some of the wasted cycles the hardware may incur when speculatively fetching one of the arms and having to trash it later. Klauser et al. [19] and Tyson's restricted scheme [11] fall in this category.

4.2.2 Predictability Based

$$LHS\_cycles + RHS\_cycles <= Branch\_cost \quad (10)$$

In this selection criteria, all branches which have high misprediction penalties gathered from the intial profile run using the combined predictor are considered as candidates for if-conversion. The branch predictability criteria is extended to selectively if-convert hammock regions only when the combined LHS_cycles and RHS_cycles is still less than the cost to execute the branch (equation 10). As a result, not all branch instructions with high misprediction costs are selected. The criteria excludes cases where the branch may have high misprediction costs but the cost to consume the LHS_cycles and RHS_cycles is still high and doesn't justify if-converting the hammock. The converse situation can also be true where the branch may have minimal misprediction costs but the combined LHS_cycles and RHS_cycles is also less such that if-converting the hammock can still be profitable.

4.2.3 Size + Predictability + Profile Based

$$LHS\_size + RHS\_size <= threshold\_value \quad (11)$$

$$LHS\_cycles + RHS\_cycles <= Branch\_cost \quad (12)$$

$$Total\_cycles(predicated) <= Total\_cycles(CFG) \quad (13)$$

In this selection method, we combine the size-based and predictability-based criterias with the profile-based criteria. The profile-based method, in addition to the branch predictability uses the block counts to compute the edge probabilities. These are used to compute the weighted schedule estimates before and after predicating the hammock (equation 13). If the $Total\_cycles$ for the predicated region is less than or equal to the $Total\_cycles$ for the original CFG, the decision to if-convert the hammock is considered profitable. In this selection criteria, the decision is considered profitable if any of the above three equations hold true.

Hardware Feature	Specifications
L1 Icache	512 kB, 32 byte line, 4 way, LRU repl. policy, 1 cycle (hit)
L1 Dcache	128 kB, 32 byte line, 4 way, LRU repl. policy, 1 cycle (hit)
Branch Predictor	gshare + bimodal predictor (2048 entry)
Functional Units	3 alu, 2 memory, 3 fpu, 1 fp mul/div
Latency	alu (1), imul (3), ld/st (2), fp-div (11), fp-sqrt (22), rest of fp ops (3), branch misprediction penalty (7)

Table 1: **Machine specifications and latencies assumed.**

Benchmarks Data Set	129.compress train	126.gcc amptjp	134.perl train	099.go train	130.li train	147.vortex train
Instr (M)	35.82	1276.5	10.51	548.2	10.5	245.4
Branches (M)	6.2	308.2	2.1	80.2	2.1	55.8
Mispredictions (M)	0.64	29.7	0.09	16.1	3.1	1.7
Misprediction Rate	9.2%	11.2%	10.2%	22.5%	5.8%	1.2%
Cycles (M)	21.0	1395.1	9.7	631.9	118.6	149.0
Inst. Per Branch	5.8	5.0	4.9	6.8	4.9	4.4
Inst. Per Cycle	1.8	1.2	1.1	1.2	1.4	2.1

Table 2: **Benchmark characteristics and baseline machine performance.**

5 Performance Evaluation

The underlying architectural model is a base superscalar processor which allows OOO execution, dynamic speculation and performs writeback in in-order fashion. The detailed specifications of the machine parameters and latencies are specified in Table 1. The register files comprises of 64 registers each 64-bits wide, out-of-which only 32 registers are visible to the external user. The processor 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 processor allows an in-order fetch and dispatch of up to four instructions per cycle.

We use the SPECint95 benchmark suite for our performance analysis. The benchmark programs were compiled using the SGI MipsPro compiler to generate a MIPS binary. An initial profiling run is made to gather the block frequencies and the misprediction counts of each branch instruction in the program. The proposed algorithm and different selection criterias are then used to select profitable candidates. The resultant binary is then fed to the SimpleScalar tool [14]. The resultant performance summaries are generated only when the underlying benchmark runs to completion successfully.

Table 2 shows the execution characteristics of the benchmarks and the baseline machine performance. The training input is used to gather the initial program profile and the benchmarks are executed using either the training or their regular inputs. 129.compress, 126.gcc and 134.perl show an average of 10% misprediction rate. 099.go is the worst among them with a heavy misprediction rate (about 23%) and 147.vortex is the best among them with only a 1.2% misprediction rate. The misprediction rate is measured as a ratio of the total number of mispredictions occured during the program execution with the number of prediction table lookups. An average of 5 instructions was observed to execute between any two consecutive branch instructions. A 1.1-2.1 IPC range is observed for the benchmarks with 147.vortex being the best and 134.perl being the worst.

5.1 Evaluation Methodology

Table 3 summarizes the results obtained for the three different selection criterias. The baseline measurement was done using the *-O3* options for all SPECint95 benchmarks. The baseline compiler includes a powerful set of optimizations including an inter-procedural analyzer, global optimizer, modulo scheduling [17] and global scheduling [18]. The net percentage cycle improvements and the percentage reductions in branch mispredictions over the baseline version (as shown in Table 2) is shown for the three different selection criterias.

Benchmarks	Size-Based		Predicatability-Based		(Size + Predictability + Profile)-Based	
	Decrease in Branch Mis-predictions (%)	Cycle Speedup (%)	Decrease in Branch Mis-predictions (%)	Cycle Speedup (%)	Decrease in Branch Mis-predictions (%)	Cycle Speedup (%)
129.compress	19.2	4.6	30.3	9.3	55.2	15.2
126.gcc	4.3	2.1	8.2	5.6	8.2	5.6
134.perl	8.9	6.1	11.8	7.3	11.8	7.3
099.go	11.2	5.2	11.2	5.2	22.2	12.4
130.li	2.1	1.4	2.1	1.4	2.1	1.4
147.vortex	0.3	0.01	0.5	0.02	1.2	0.5

Table 3: **Performance Improvements for SPECint95 benchmarks using the three different selection criterias. The two subcolumns under each of the selection criteria show the net improvement (in percentages) over corresponding results obtained using the baseline version (shown in Table 2).**

5.1.1 Effect on Branch Mispredictions and Cycle Improvements

The size-based method, due to its limited nature and not being fully cognizant about profitability ratios, did decently well. The selection criteria was able to reduce about 19.2% of the branch mispredictions on compress benchmark and about 11.2% of the mispredictions on 099.go benchmark. The selection method wasn't quite effective for other cases. In case of the 129.compress benchmark, most of the branch hammocks selected contribute a significant portion of the benchmarks total number of mispredictions. The selection method wasn't quite effective for the 126.gcc benchmark because most of the problematic branches had calls or other unsafe instructions which can't be safely speculated hence couldn't be if-converted. The net percentage cycle improvement over the baseline number using this selection method varied between 1%-6%.

The predictability-based method overall faired better than the size-based method. Most of the candidates selected were fairly medium in size (with a maximum of upto 35 instructions for 134.perl benchmark). 129.compress benefited the most with an overall reduction in branch mispredictions upto 30%. This selection method was able to target branches which were relatively difficult to predict with high misprediction costs and eliminating them was more profitable than the extra annulled operations. The overall cycle improvement over the baseline number using this selection method ranged between 1% - 9%.

The size + predictability + profile-based method did the overall best, both in reducing the total number of branch mispredictions and in the net cycle improvement. 099.go benchmark improved a lot using this selection method with an overall 22% reduction in branch mispredictions and with a cycle speedup over the baseline version upto 12%. This selection method was able to compare the parellelism present in the if- and else- paths with the branch execution costs and able to if-convert if the costs associated with the latter is much worser. The overall cycle improvement over the baseline number using this selection method ranged between 1% - 15%.

130.li and 147.vortex benchmarks didn't improve using either of the three different selection criterias. The overall % reductions in branch mispredictions and cycle speedups were minimal.

Benchmarks	Total no. of mis-speculated instrs.	
	Baseline (M)	Proposed (M)
129.compress	6.4	4.7
126.gcc	240.0	209.2
134.perl	1.5	1.31
099.go	134.9	119.6
130.li	41.4	37.6

Table 4: **Reduction on total number of mis-speculated instructions over the results obtained using the baseline model without if-conversion.**

5.1.2 Reduction in Mis-speculated instructions

A mis-speculation happens in a dynamic execution instance whenever there is a branch misprediction detected. Instructions fetched and/or executed from the wrong path need to be undone. Execessive misspeculation sequences can deteriorate the program performance since many cycles will be wasted in invalidating all buffers and bringing the processor state to the correct branch. All the instructions required to recover the processor state to the point of the mispredicted branch are counted as misspeculated instructions. As noticed in Table 4 under the baseline model, 126.gcc and 099.go suffer from execessive misspeculations. These speculative instructions don't contribute to the useful program parallelism. Using the proposed selection algorithm not only reduced the overall misprediction rate in the hardware (as shown in Table 3) but also reduced the mis-speculation rate (as shown in Table 4). The hardware was able to reduce the scenarios where the speculated instructions had to be annulled later. This can be attributed due to several reasons, (1) overall improvement in branch prediction, and (2) elimination of difficult to predict branches. Most noticable % decrease was observed in 129.compress benchmark where the number of misspeculations decreased from 6.4M to 4.7M instructions.

5.1.3 OOO Extensions

We made numerous adjustments to schedule estimates to better reflect the OOO behavior of the processor. This allowed the if-conversion algorithm to benefit from accurate estimates and make it more aggressive in selecting profitable candidates.

First, among them, was to relax the anti- and output- dependences. The scheduler relies on the accurate dependence graph to compute its schedule estimate. The if-conversion algorithm uses this schedule estimates to drive its profitability criteria. All *anti-* and *output* dependences are treated as arcs with zero latencies. This allows anti- and output- dependent operations to be scheduled in the same cycle. The estimate is more aggressive and this emulates closely to the hardware dynamic renaming scheme that the architecture relies on when performing OOO execution.

Second, an attempt is made to reduce the number of load-store spill instructions. The local scheduling phase is altered to minimize spill instructions at the expense of a possible increase in schedule length. The assumption is that the hardware can overcome this limitation by doing OOO execution. However, if the program is bottlenecked with memory (including spill) instructions, there is not much that the hardware can do, other than benefit from bypassing using limited buffers. The detailed explanation of the register-sensitive adjustments is beyond the scope of this paper.

Benchmarks	Additional Percentage Improvement
129.compress	0.6
126.gcc	3.4
134.perl	1.6
099.go	4.8
130.li	1.0

Table 5: **Additional performance improvements over the results obtained when the OOO adjustments are not made.**

Third, an attempt is made to schedule the compare instructions which set the branch predicates as early as possible. The intent is that this will help in the early resolvement of the condition and reduce any misspeculations. This adjustment directly doesn;t affect the applicability of the algorithm but improves on the scheduling performance by reducing any ill-effects involved with dynamic speculation.

Table 5 shows the additional performance improvements over the results obtained when the OOO adjustments were added. This results in an additional 1% - 5% improvement over results obtained when the OOO adjustments were not made using the size + predictability + profile based selection criteria.

6 Summary and Conclusions

In this paper, we present a new algorithm which uses profiling input and selectively performs if-conversion for architectures with dynamic speculation and guarded execution support. Branch profiling is done using the MacFarling's combined branch predictor. This coupled with block profiling is then used to select hammock regions which are profitable to if-convert. Three different profitability selection criterias, namely size-based, predictability-based and profile-based are proposed and their effects on net cycle improvements, branch mispredictions and misspeculated instructions studied. The profile-based method did the overall best. We observed that selective and profitable guarded execution not only reduces the misprediction rate but also the mis-speculation rate in the hardware. The performance of the underlying OOO processor improved further when the proposed OOO adjustments were added.

References

[1] J.C. Dehnert, P.Y.T. Hsu and J. P. Bratt, "Overlapped loop support in the Cydra5," *Proceedings*

of the 16th Annual International Symposium on Computer Microarchitecture, 1989

[2] P.Y. Chang, E. Hao, Y. Patt and P. Chang, "Using Predicated Execution to Improve the Performance of a Dynamically Scheduled Machine with Speculative Execution," *International Conference on Parallel Architectures and Compilation Techniques*, 1995

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

[4] S.A. Mahlke, R.E. Hank, R.A. Bringmann, J.C. Gyllenhall, D.M. Gallagher and W.W. Hwu, "Characterizing the Impact of Predicated Execution on Branch Prediction," *Proceedings of the 27th International Symposium and Workshop on Microarchitecture (MICRO-27)*, Nov 30 - Dec 2, 1994

[5] S.A. Mahlke, D.C. Lin, W.Y. Chen, R.E. Hank and R.A. Bringmann, "Effective Compiler Support for Predicated Execution Using the Hyperblock," *Proceedings of the 25th International Symposium and Workshop on Microarchitecture (MICRO-25)*, Portland, OR, December 1-4, 1992

[6] A. Nicolau and J.A. Fisher, "Measuring the available parallelism for very long instruction word architectures," *IEEE-TC, C-33, p 1088-1098* September, 1984

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

[8] P.Y. Chang, E. Hao, T.Y. Yeh and Y.N. Patt, "Branch classification: A new mechanism for improving branch predictor performance," *Proceedings of the 27th Annual International Symposium on Microarchitecture*, 1994

[9] B.R. Rau, D.W.L. Yen and W. Yen and R.A. Towle, "The Cydra 5 Departmental Supercomputer: Design Philosophies, Decisions, and Tradeoffs," *The IEEE Computer*, p 12-25, January, 1989

[10] S. McFarling, "Combining branch predictors," *Technical Report TN-36*, DEC Laboratories Ltd., June 1993

[11] G. Tyson, "The Effects of Predicated Execution on Branch Prediction," *Proceedings of the 27th International Symposium and Workshop on Microarchitecture (MICRO-27)*, San Jose, CA, Nov 30 - Dec 2, 1994

[12] T.Y. Yeh and Y.N. Patt, "Two-level adaptive branch prediction," *Proceedings of the 24th International Symposium and Workshop on Microarchitecture (MICRO-24)*, 1991

[13] E. Jacobson, E. Rotenberg and J.E. Smith, "Assigning Confidence to Conditional Branch Predictions," *Proceedings of the 29th International Symposium on Microarchitecture*, 1996

[14] D. Burger, T.M. Austin and S. Bennett, "Evaluating Future Microprocessors: The Simplescalar Tools Set," *Technical Report TR#1308, University of Wisconsin*, 1996

[15] S. McFarling, "Combining Branch Predictors," *WRL Technical Note TR-36, Digital Equipment Corporation*, June 1993.

[16] E. Rotenberg and Q. Jacobson and J. Smith, "A study of Control Intedependence in Superscalar Processors," *Fifth International Symposium on High Performance Computer Architecture (HPCA-5)*, 1999.

[17] J. Ruttenberg and G.R. Gao and A. Stoutchinin and W. Lichtenstein, "Software Pipelining Showdown: Optimial vs. Heuristic Methods in a Production Compiler," *ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 1996.

[18] M. Srinivas and S. Jain and J. Dehnert, "A New Framework for Integrated Global Local Scheduling," *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 1998

[19] A. Klauser, T. Austin, D. Grunwald and B. Calder, "Dynamic Hammock Predication for Non-predicated Instruction Set Architectures," *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 1998