

A Comparative Performance Evaluation of Various State Maintenance Mechanisms

Michael Butler and Yale Patt
Department of Electrical Engineering and Computer Science
The University of Michigan
Ann Arbor, Michigan 48109-2122

Abstract

Speculative execution and dynamic scheduling are two promising techniques for achieving high performance in superscalar processors. These techniques require a mechanism for maintaining all architecturally visible machine state. In this study we examine the performance implications of three common state maintenance mechanisms: the reorder buffer, the history buffer, and checkpointing. We model the execution of the four integer benchmarks from the SPEC89 suite for a variety of maintenance techniques. We report the results of these measurements and their implications with respect to the design of high performance superscalar processors.

1 Introduction

Speculative execution and dynamic scheduling are two promising techniques for achieving high performance single-instruction-stream execution. These techniques however, involve modifying architectural machine state before it is known if such modifications are dictated by the correct sequential execution of the program. Implementation of these techniques requires management of the machine state such that correct execution can be guaranteed. We term any such mechanism a state maintenance mechanism.

State maintenance mechanisms provide for correct execution at the cost of various hardware support structures as well as possible performance degradation. To assess the performance implications of various state maintenance mechanisms, we model the execution of four integer benchmarks from the SPEC89 suite. We have measured the performance of one machine configuration under a variety of state maintenance assumptions.

Three basic approaches to state maintenance are modeled in this study: Reorder Buffer [1], History Buffer [1], and Checkpointing [2]. Performance differ-

ences among the three techniques arise from the time taken for each mechanism to recover from a branch misprediction. Checkpointing incurs the lowest misprediction penalty and thus achieves the highest average parallelism (instructions per cycle, IPC). The reorder buffer technique incurs, on average, an additional 1.6 to 2.7 cycle misprediction penalty (over checkpointing), for a decrease in average parallelism of between 4.0 and 7.7 percent. A moderately aggressive history buffer approach suffers an average of 1.6 to 1.9 additional cycles per misprediction, for a decrease in parallelism of 4.8 to 9.1 percent.

This paper is organized into five sections. Section 2 discusses the microarchitectural model of execution and the state maintenance mechanisms that we have simulated. Section 3 describes our experiments: the simulator, the benchmarks, and the machine configurations tested. Section 4 reports the results of our simulations and discusses the influence of state maintenance on performance. Section 5 offers some concluding remarks.

2 Model of Execution

The microarchitecture modeled in this study is a dynamically scheduled, speculative execution engine designed to exploit instruction level parallelism. We call this model the High Performance Substrate (HPS) [4].

Execution in HPS flows as follows: Each cycle multiple instructions are issued, and, using the information in the Register Alias Table, the instructions are merged into node tables, much like the Tomasulo algorithm merges operations into the reservation stations of the IBM 360/91 [8]. Associated with each instruction (node) are the source operands for that instruction (or identifiers for obtaining the operands), and destination information. Each node is stored in a node table independent of and decoupled from all other nodes currently awaiting dependencies in the datapath

until all its operands are available, at which point the node is eligible for firing. Each cycle ready nodes are fired, i.e. shipped to pipelined function units for execution. Each cycle, function units complete execution of nodes and distribute the results to nodes waiting for these results, which then may become firable.

For memory operations, a node is firable only if all of its operands are available, it is not dependent on any previous memory operations (flow dependence), and there are no previous memory operations to unknown addresses that may interfere with this operation. This dynamic memory disambiguation requires that, in the case of load operations, no previous stores are to unknown addresses. Likewise, for store operations, any previous loads or stores to unknown addresses will stall the store operation.

A separate node table exists in front of each function unit. Instructions are routed at issue time to a node table that feeds a function unit capable of performing the operation. Since each function unit has its own independent node table, scheduling is limited only to nodes residing in that node table thus easing the hardware requirements for scheduling and routing between node tables and function units.

2.1 State Maintenance

Out-of-order execution makes the support of precise interrupts more difficult. In fact, early machines that implemented out-of-order execution (completion) did not support precise interrupts (e.g. CRAY I, IBM 360/91 floating point unit). The difficulty lies in restoring the machine to a state such that all instructions which precede the offending instruction in the dynamic I-stream have updated the machine state, while none of those following it have.

Another difficulty arises from the employment of speculative execution. Dynamic branch prediction allows the machine to speculatively bring new instructions into the datapath before it has been confirmed that they are to be executed. While this allows for a larger "window" of instructions in which to find useful work, speculative execution must be supported by a mechanism to remove instructions that were erroneously issued.

Since exceptions are rare, the performance implications of a recovery technique for exceptions are minimal. Branch mispredictions, on the other hand, occur far more frequently and significantly impact the performance of wide issue machines. This study focuses on the branch misprediction recovery time and its impact on overall performance.

Several techniques have been proposed in the liter-

ature to provide support for machine state recovery of speculative, out-of-order execution engines.

2.2 Reorder Buffer

Smith and Pleszkun [1] proposed a state maintenance mechanism that involves updating the architecturally visible register file strictly in-order, while allowing execution to proceed out-of-order. This is accomplished by maintaining a FIFO queue of instructions in the order in which they were issued. When an instruction completes execution, it places the result in the appropriate slot in the FIFO queue rather than in the register file. The queue, in turn, updates the register file in-order. Exceptions are handled when the offending instruction reaches the head of the queue. If the instruction has created an exception, by the time it reaches the head of the queue, all previous instructions have updated the register file, while no subsequent instructions have. Thus exceptions are precise.

This approach can also be applied to misprediction recovery. When a mispredicted branch reaches the head of the buffer, the rest of the buffer is flushed and issue proceeds along the correct path. This behavior is modeled for the reorder buffer in this study.

It should be noted that another reorder buffer-based technique can be used that allows for misprediction recovery to proceed immediately upon branch resolution. This technique will achieve the same performance as the checkpointing approach modeled in this study. The difficulty with this approach is that it requires a wide content addressable search with a rotating priority mechanism. The rotating priority mechanism is necessary since we want only the most recent result for a given register. Since several slots may contain results for the same architectural register, a simple CAM match is not sufficient. Furthermore, since the reorder buffer acts as a queue, the priority mechanism must allow for a rotating "start" position, or the FIFO must be implemented as a true hardware queue using shift registers.

2.3 History Buffer

Another technique proposed by Smith and Pleszkun is the history buffer. With this technique, instructions are free to update the register file as they complete execution, however, the previous value of the register is maintained in a LIFO queue. This LIFO containing the "history" of the register file is arranged, as in the reorder buffer, with a slot for each instruction in the order in which they were issued. The head of the queue contains the oldest instruction, and when it completes

it can be removed (“retired”) from the queue (i.e. the contents of the queue entry are discarded). When an instruction which has caused an exception reaches the top of the queue, the register file is reconstructed by copying the history buffer back into the register file beginning with the tail. Since we write back into the register file from tail to head, multiple writes to the same architectural register are correctly resolved.

This same approach can be taken to support branch misprediction recovery by “undoing” only those instructions issued since the offending branch. The performance penalty associated with this method arises from the cycles spent writing previous values back into the register file.

2.4 Checkpointing

Checkpointing [2] is an alternative state maintenance mechanism that protects vulnerable machine state at certain key points in the instruction stream. A “checkpoint” is a point in the dynamic instruction stream at which the machine state is preserved in some way so as to allow efficient restoration of the architecturally visible machine state. If checkpoints are established at strategic points in the instruction stream, such as at each branch, then the machine can quickly recover from branch mispredictions. Furthermore, as with the other recovery mechanisms, checkpointing can be used to support precise interrupts by backing the machine to the nearest previous checkpoint and then single stepping instruction issue until the offending instruction is reached again. This involves performing extra work, but since exceptions are rare, this is not likely to degrade performance noticeably.

The implementation of checkpointing involves tagging checkpointed entities (e.g. physical registers) with a bit field which indicates the checkpoints in which the entry “exists.” This bit field has a single bit for each checkpoint that the machine is capable of supporting. If the bit corresponding to the current checkpoint is set, then this physical register is the most recent reference to the associated architectural register. By manipulating the contents of this bit field, the register instance can be propagated to subsequent checkpoints or can be overwritten (by clearing the current bit). Backing up the machine requires returning to a previous set of physical-to-architectural mappings. This is accomplished by treating a previous checkpoint as the current one. Similarly, retiring a checkpoint that is no longer needed simply requires clearing the corresponding bit in these bit fields. A more detailed explanation of the new checkpointing as applied to the register file can be found in [7].

2.5 Response and Recovery Time

There are essentially two components of the performance impact of recovery mechanisms. The first is the time from the determination of the misprediction to the time when recovery begins. We call this time the “response time” for the misprediction. The other relevant time is the number of cycles actually spent performing the recovery. We call this the “recovery time.”

The performance of the recovery models differ in the values for these two quantities. Checkpointing is the most aggressive in terms of responding to and recovering from mispredictions. As explained above, a reorder buffer delays response until the branch reaches the head of the queue, but then immediately recovers. A history buffer, on the other hand, responds immediately but requires zero or more cycles to recover.

It is important to point out that the performance implications of these techniques are investigated only as they affect the register file. Use of these techniques to protect other structures is not considered here. The operation of the base machine in terms of issue constraints, window size, etc. does not vary - only the penalties associated with state recovery.

3 Experiments

3.1 Benchmarks

The results presented in this paper are for four integer programs from the SPEC suite: eqntott, espresso, gcc, and li, compiled for and run under the M88000 instruction set architecture. The benchmarks were compiled using Diab Data C Rel. 2.4 compiler with all optimizations turned on. All benchmarks were run unchanged with the following exception: cpp is not called in eqntott, gcc was run without cpp and used the output of the preprocessor as the input file.

Table 1 shows instruction classes and their simulated execution latencies. Each instruction class is listed with its execution latency (in cycles), and a description of the instructions that belong to that class.

3.2 Machine Configurations

The machine configuration simulated is listed in Table 2. The machine is specified by its set of function units. The datapath contains eight pipelined function units each capable of servicing different classes of instructions. The first three function units are capable of performing load/store, bit field, and integer operations. The fourth function unit can perform bit

Instruction Class	Execution Latency	Description
FP Add	3	FP add, sub, and convert
Multiply	3	FP mul and INT mul
Divide	8	FP div and INT div
Mem Load	2	Memory loads
Mem Store	1	Memory stores
Branch	1	Control instructions
Bit Field	1	Shift, and bit testing
Integer	1	INT add, sub and logic OPs

Table 1: Instruction Classes and Latencies

Function Unit	Instruction Classes
1	Load/Store, Bit Field, Integer
2	Load/Store, Bit Field, Integer
3	Load/Store, Bit Field, Integer
4	Bit Field, Integer
5	Divide, Integer
6	Multiply, Integer
7	FP Add, Integer
8	Branch, Integer
Cache	Size
I Cache	16k
D Cache	16k

Table 2: Machine Configuration

field and integer operations. The fifth, sixth and seventh FUs perform divides, multiplies and adds respectively, as well as integer operations, and the eighth FU performs branches and integer operations. Thus, all function units are capable of handling simple integer operations as well as another class of instructions.

3.3 Simulation Process

In contrast to previous performance studies, which typically employ trace-driven simulation, this study employs “full execution” simulation. The simulator reads in the executable image of the benchmark and simulates execution of a dynamically scheduled machine. The actual contents of registers and memory are maintained and cycle by cycle simulation of execution of the benchmark is performed. The ability to pursue mispredicted paths and allow them to interfere with execution gives rise to a more accurate simulation. This increased accuracy comes at the expense of increased simulation time of roughly three times that of trace-driven simulation.

Some relevant characteristics of the machine mod-

eled in this study are:

- Separate Instruction and Data caches are explicitly modeled in the simulator.
- Dynamic memory disambiguation is performed. Previous loads from unknown addresses will stall all subsequent stores until the load address has been resolved, and previous stores to unknown addresses will stall subsequent loads and stores.
- All function units are fully pipelined (i.e. able to initiate a new operation each cycle) and are mutually independent. The function unit latencies we used are given in Table 1.
- When a trap is encountered, the machine being simulated must stop issue, wait for all instructions currently in the window to complete, and then execute the trap instruction.
- Branch Prediction - As branches are encountered, a prediction is made based on the history of that branch combined with dynamically gathered information [12]. The machine proceeds with execution along the predicted path. This prediction is compared to the real outcome as determined by execution of the branch instruction. In the event of a misprediction, appropriate actions are taken in order to perform recovery of the machine state.

There are several key parameters that define the recovery model:

- Delay until Head - This flag indicates that the “response” to the branch misprediction is delayed until the branch reaches the head of the queue (i.e. all previous instructions have retired).
- Delay for Issue Cycles - This parameter assigns the “recovery” penalty to be equal to the number of issue cycles since the offending branch was issued. “Issue cycles” are cycles in which one or more instructions were issued. Stalling issue for events such as an instruction cache miss, results in no “issue cycles.”
- Delay for Register Writes - This flag results in a “recovery” time equal to the number of cycles needed to write the required number of values back to the register file. The number of writes required is a function of the number of instructions (which write to registers) that have been issued since the offending branch. This number is modified by the options listed below.

- Remove Dupes - This option reduces the number of register writes that have to occur in order to accomplish complete recovery by eliminating duplicate writes to the same architectural register.
- Only Ready Values - This option further reduces register writes by only writing back instances where the value of the register has been produced and distributed already.

4 Simulation Results

The performance figures 1 through 8 plot the performance, given in instructions retired per cycle (IPC), for various recovery models and benchmarks. Each chart shows the running (cumulative) average IPC for each variation of the maintenance mechanism for the first hundred million instructions retired, except for Gcc compiling dbxout.i, which runs to completion in 46 million instructions. The curve labeled "History Buffer" is a machine that immediately responds and recovers according to the "Delay for Register Writes" option described above.

4.1 The Basic Recovery Mechanisms

The performance of the three basic recovery mechanisms are shown in figures 1 through 4. The curves plot the cumulative averages of the IPC for the first execution region of the benchmarks. The relative performance of the different mechanisms is remarkably consistent across the benchmarks. The reorder buffer and the history buffer achieve roughly equivalent performance for the four benchmarks. Checkpointing outperforms the other techniques by a modest but consistent 4 to 9 percent.

Fluctuations in IPC that are common to all three models indicate characteristics of the region of execution that are independent of recovery efficiency. There are, however, regions of several of the benchmarks that favor one mechanism over another. In Eqntott for instance, a reorder buffer performs better than a history buffer during the early stages of execution (the first 1.5 million instructions) but then underperforms for the rest of the run.

To better quantify the performance differences in these models, table 3 gives the average delay per misprediction for the different benchmarks for the first ten million instructions. The checkpointing model has values of zero for both delays. The history buffer model is the column labeled "Imm: Inst." The first number is the additional time taken to "respond" to a misprediction. The second number is the "recovery" time. The

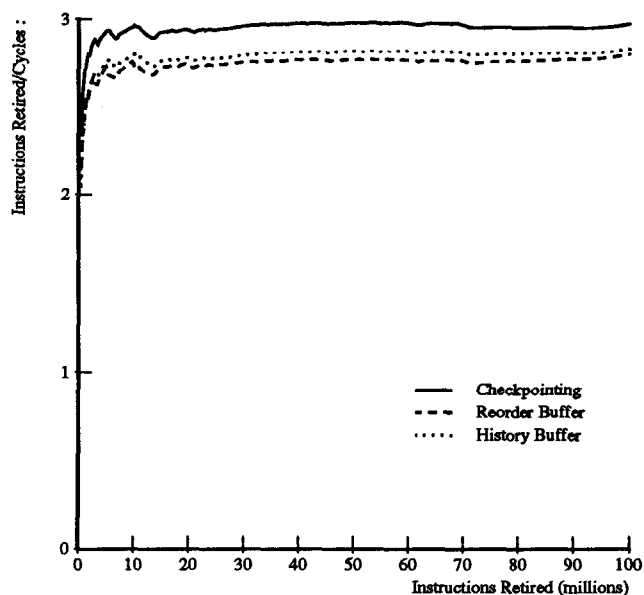


Figure 1: Eqntott

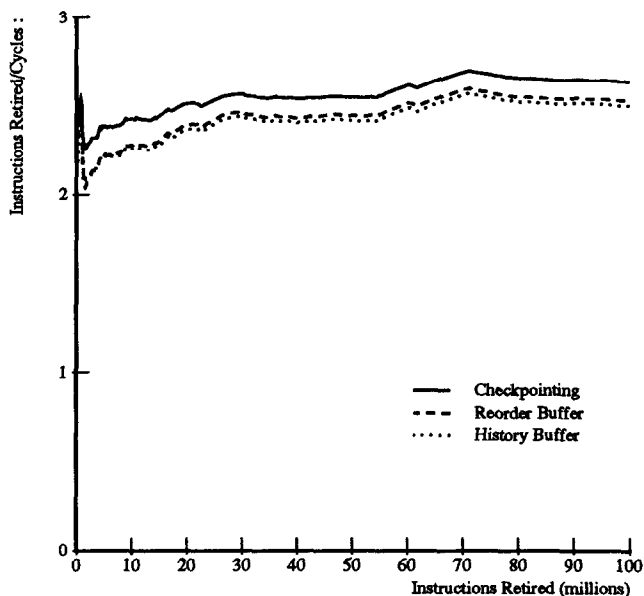


Figure 2: Espresso

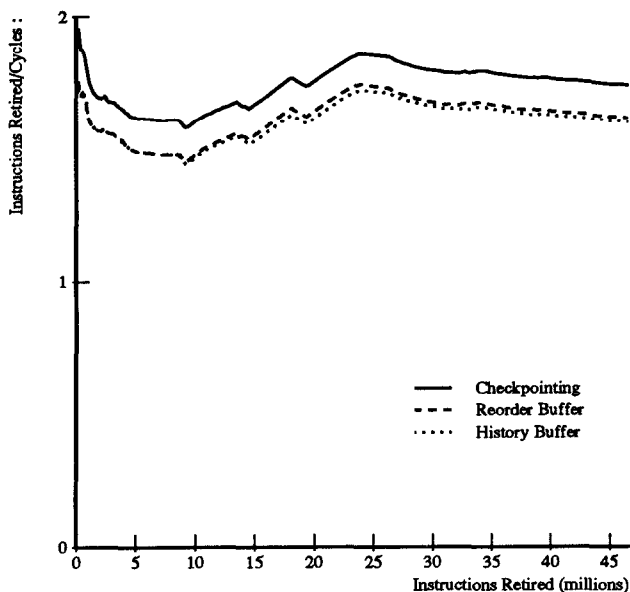


Figure 3: GCC

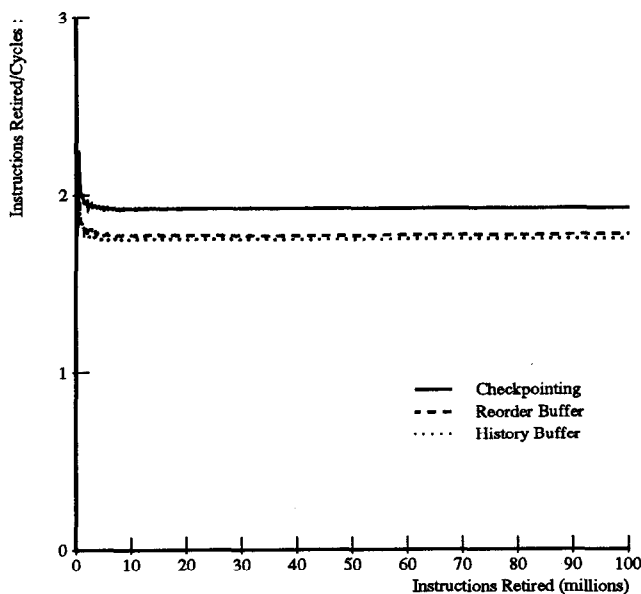


Figure 4: Xlisp

sum of the two numbers gives the average number of cycles spent per misprediction above and beyond the time taken to resolve the branch.

4.2 History Buffer Variations

Figures 5 through 8 show the performance for several variations of the history buffer approach. For comparison purposes, the graphs also include the curve showing the performance of the checkpointing model. Each curve is labeled with a notation indicating the parameter settings for that particular model. The first word identifies the “response time.” “Head” indicates that response is delayed until the branch reaches the head of the queue. “Imm” indicates that the machine immediately responds. The rest of the notation describes the recovery time. “Cycle” indicates that the machine delays for the number of issue cycles since the branch was issued. “Inst” identifies recovery as a function of the number of instructions that have to be undone (see Delay for Register Writes, above) and can be further modified by “No Dupes” and “Ready Only” as above. Due to time limitations, only the first ten million instructions are simulated for these models, however the performance shown is consistent with test runs up to one hundred million instructions.

The history buffer variations show a much wider range of performance. The best history buffer mechanism outperforms the base history buffer by 24 to 30 percent. Once again, the performance of the different variations is consistent across the four benchmarks.

With the exception of history buffers that recover based on “issue cycles” since the offending branch was issued, the relevant data that affects recovery time are the number of register writes that need to be performed to accomplish recovery. Figures 9 through 11 show histograms of the number of values for the gcc benchmark that need to be written back to the register file for recovery using the different history buffer techniques. The reduction of the number of writes results in the performance increase in the more aggressive models.

5 Concluding Remarks

An efficient state maintenance mechanism allows for more aggressive exploitation of instruction level parallelism by providing support for speculative and out-of-order execution. Any mechanism however comes at the expense of hardware and design complexity. We have investigated the performance implications of a variety of mechanisms.

Checkpointing provides the most aggressive ap-

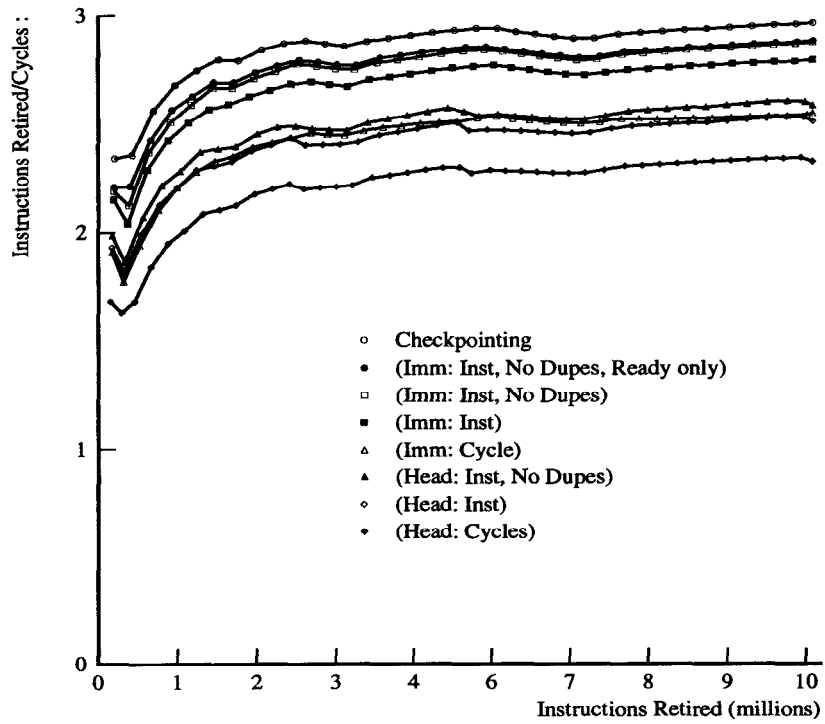


Figure 5: Eqntott - History Buffer Variations

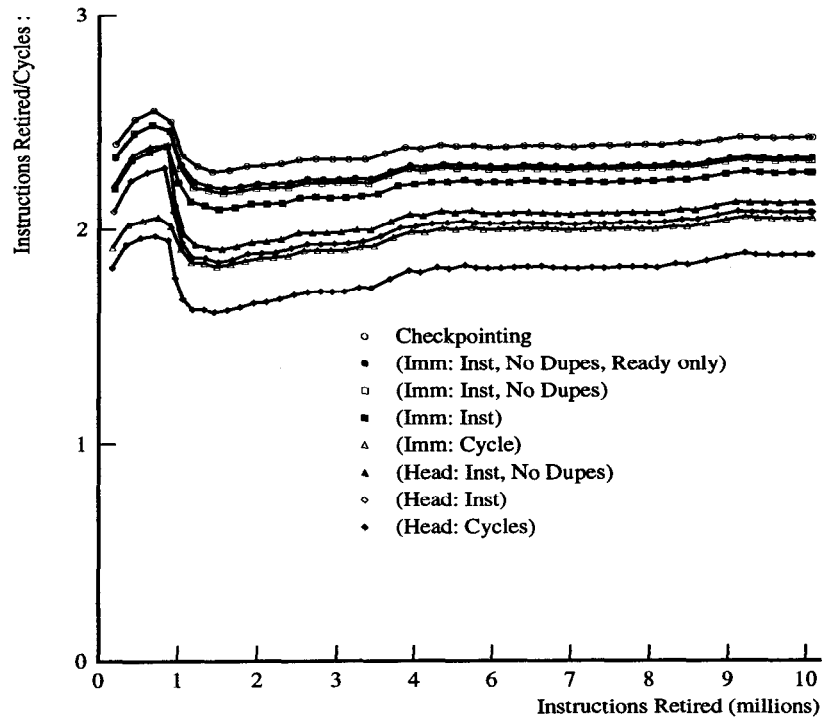


Figure 6: Espresso - History Buffer Variations

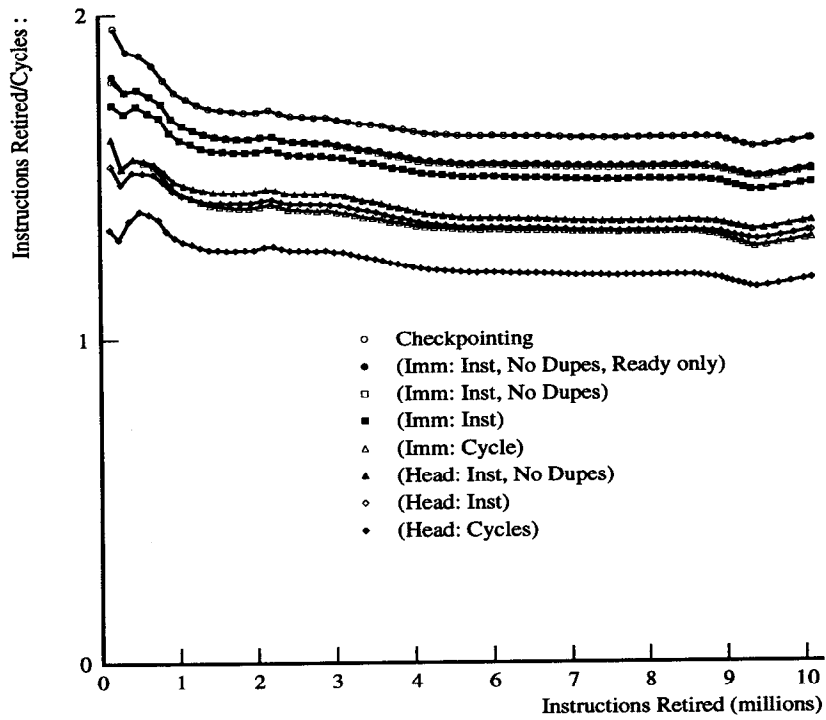


Figure 7: GCC - History Buffer Variations

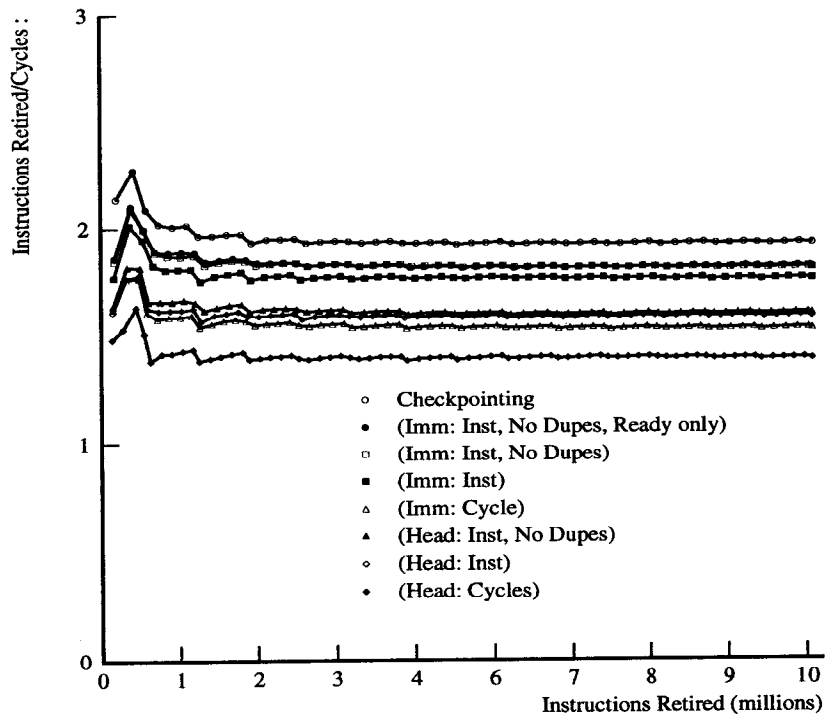


Figure 8: Xlisp - History Buffer Variations

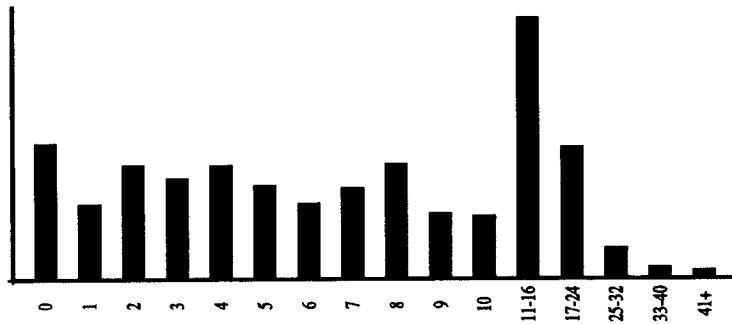


Figure 9: GCC - Register Writes for HB Recovery - All Instructions

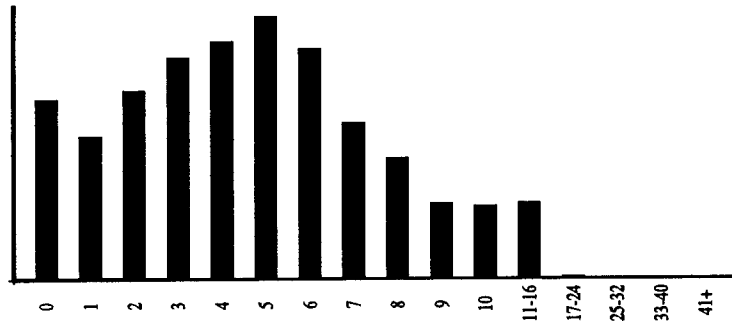


Figure 10: GCC - Register Writes for HB Recovery - Remove Dups

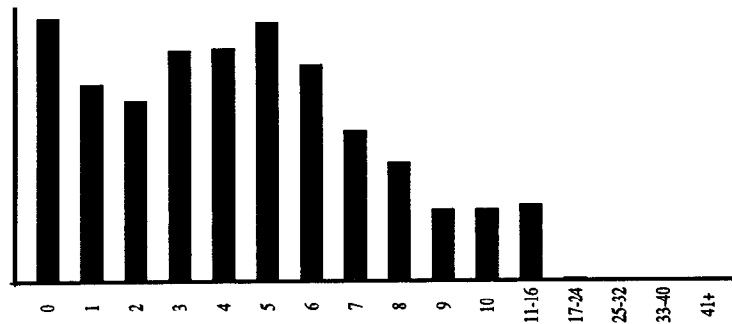


Figure 11: GCC - Register Writes for HB Recovery - Only Ready Values

Benchmark	ROB	(Head: Cycles)	(Imm: Cycles)	(Imm: Inst)	(No Dups)	(Ready Only)
Eqntott	2.99 + 0	3.00 + 5.32	0 + 5.33	0 + 2.07	0 + 1.01	0 + 0.95
Espresso	1.71 + 0	1.71 + 3.90	0 + 4.09	0 + 1.52	0 + 0.98	0 + 0.84
Gcc	1.99 + 0	1.97 + 4.09	0 + 4.37	0 + 1.62	0 + 1.09	0 + 1.04
Xlisp	1.55 + 0	1.55 + 4.21	0 + 4.39	0 + 1.63	0 + 1.11	0 + 1.02

Table 3: Average additional delay per mispredicted branch (response + recovery)

proach to branch misprediction recovery. This gives rise to a modest but consistent performance improvement over other mechanisms. The performance of a reorder buffer based machine lags by 4 to 8 percent, while a moderately aggressive history buffer based machine lags by a similar 5 to 9 percent. Variations of the history buffer approach result in a wider range of performance differences of 24 to 30 percent.

The results presented here are for a wide issue (eight) machine with a large window. This aggressive machine was chosen to allow enough "room" to isolate the recovery mechanism's impact without having other bottlenecks unnecessarily limit performance and thus cloud the results. We plan on exploring other machine configuration spaces to investigate the effect of state maintenance on different window sizes and machine widths.

Acknowledgement

This paper is one result of the HPS Architecture research that we are doing at Michigan. The support of Motorola, Intel, Hewlett Packard, Scientific and Engineering Software, and HaL is greatly appreciated. In addition, NCR has been continuously enthusiastic about our work. The support of Jim Pike, including the gift of two NCR Tower multiprocessors which we have used to perform our simulations, is also greatly appreciated. Finally, we acknowledge that we consider ourselves very fortunate in having the opportunity to discuss ideas about the HPS execution model with the rest of our research group at Michigan.

References

- [1] J. E. Smith, and A. R. Pleszkun, "Implementation of Precise Interrupts in Pipelined Processors" *Proceedings of the 12th Annual International Symposium on Computer Architecture*, (June 1985), pp. 36-44.
- [2] W.W. Hwu and Y.N. Patt, "Checkpoint Repair for Out-of-order Execution Machines.", *Proceedings of the 14th Annual International Symposium on Computer Architecture*, (June 1987), pp. 18-26.
- [3] Motorola Inc. *MC88100 RISC Microprocessor User's Manual*. Phoenix AZ.: Motorola Inc., 1991.
- [4] Y.N. Patt, W. Hwu, and M. Shebanow, "HPS, A New Microarchitecture: Rationale and Introduction.", *Proceedings of the 18th Annual Workshop on Microprogramming*, (December 1985), pp. 103-108.
- [5] Y.N. Patt, W. Hwu, and M. Shebanow, "Critical Issues Regarding HPS, A High Performance Microarchitecture.", *Proceedings of the 18th Annual Workshop on Microprogramming*, (December 1985), pp. 109-116.
- [6] W. Hwu and Y.N. Patt, "HPSm, a High Performance Restricted Data Flow Architecture Having Minimal Functionality.", *Proceedings of the 13th Annual International Symposium on Computer Architecture*, (June 1986), pp. 297-307.
- [7] M. Butler and Y. Patt, "An Area-Efficient Register Alias Table for Implementing HPS," *Proceedings of the 1990 International Conference on Parallel Processing*, August 1990, pp. 611-612.
- [8] R.M. Tomasulo, "An Efficient Algorithm for Exploiting Multiple Arithmetic Units.", *IBM Journal*, Vol. 11, (January 1967), pp. 25-33.
- [9] W.M. Johnson, "Super-Scalar Processor Design", Technical Report No. CSL-TR-89-383, Stanford University, (June 1989).
- [10] G. S. Sohi and S. Vajapeyam, "Instruction Issue Logic for High-Performance Interruptable Pipelined Processors.", *Proceedings of the 14th Annual International Symposium on Computer Architecture*, (June 1987), pp. 27-34.
- [11] M. Butler, T. Yeh, Y. Patt, M. Alsup, M. Shebanow, and H. Scales, "Single Instruction Stream Parallelism Is Greater than Two," *Proceedings of the 18th International Symposium on Computer Architecture*, (May 1991), pp. 276-286.
- [12] T-Y Yeh and Y.N. Patt, "Two-Level Adaptive Branch Prediction", *The 24th ACM/IEEE International Symposium and Workshop on Microarchitecture*, (Nov. 1991), pp. 51-61.