# An Investigation of the Performance of Various Dynamic Scheduling Techniques

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

## Abstract

*An important design decision in the implementation of a superscalar processor is the amount of hardware to allocate to the instruction scheduling mechanism. Dynamic scheduling provides, at the cost of (possibly substantial) additional hardware, the potential to improve upon a static schedule by incorporating run-time information in the scheduling decision. In this study we model several different scheduling techniques and machine configurations. We measure the performance of these models on seven integer and floating point benchmarks taken from the SPEC89 suite. We report the results of these measurements and their implications with respect to the design of high performance superscalar processors.*

## 1 Introduction

Dynamic scheduling refers to a class of instruction scheduling techniques performed by hardware at run time. The goal of dynamic scheduling is to improve upon an existing static schedule by incorporating run-time information into the scheduling decision. This capability, however, comes at the cost of additional hardware support. The amount of hardware required is a function of the scheduling technique employed.

To assess the performance implications of various dynamic scheduling techniques, we model the execution of four integer benchmarks and three floating point benchmarks from the SPEC89 suite. We have measured the performance of one machine configuration under a variety of scheduling assumptions.

Our results show that most dynamic scheduling techniques deliver similar performance - within 5 percent of each other for integer benchmarks and within 20 percent for floating point benchmarks. This is due to the typically low number of ready nodes waiting for execution. Furthermore, we have found that a unified node table does little to improve integer benchmark performance, but improves floating point benchmark performance by around 25 percent.

This paper is organized in five sections. Section 2 discusses the microarchitectural model of execution and the scheduling techniques 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 scheduling on performance. Section 5 offers some concluding remarks and discusses the future work we have planned.

## 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 firable nodes are scheduled, i.e. shipped to pipelined functional units for execution. Each cycle, functional units complete execution of nodes and distribute the results to nodes waiting for these results, which then may become firable.



0-8186-3175-9/92 $3.00 © 1992 IEEE

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.

There are two basic node table configurations that we studied in this experiment: split and unified. In the split configuration, a separate node table exists in front of each functional unit. Instructions are routed at issue time to a node table that feeds a functional unit capable of performing the operation. Since each functional 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 functional units. This simplified scheduling comes at the expense of increased issue logic and routing complexity. The unified configuration, on the other hand, issues all instructions into a common node table that then feeds all functional units. The issue logic and routing demands are reduced at the expense of putting off the problem until nodes are fired and sent to functional units. Scheduling across a common node table may also involve more logic and/or more scheduling time since a larger number of instructions must be examined and routed.

Several different dynamic scheduling techniques were modeled in this study for each of the two node table configurations. Each cycle, hardware must select, or "schedule", a ready node to be executed by each functional unit. This involves using some heuristic to select a node to fire from the pool of available ready nodes. If no ready nodes are available the functional units remain idle. The eight different techniques modeled in this experiment are as follows:

- Branch Path - Nodes that are along the path to resolve a branch are given higher priority. In the case of ties, the oldest node is selected.
- Chain Length - Nodes are given priority according to the maximum length of dependent chains of nodes (currently in the machine). In the case of ties, the oldest node is selected.
- Fixed Priority - Nodes are selected based on a fixed priority scheme based on physical location in the node table. Since nodes are placed into node table entries in a rotating manner by the issue logic, this scheduling technique may allow simpler hardware while slightly favoring older nodes.
- Head Only - Only the oldest node (at the "head" of the table) is checked for possible scheduling. This represents a low cost scheduling technique that for split node tables, allows slip to occur between functional units, but within functional units instructions fire in original program order. For a "unified" node table configuration, this scheduling technique results in no instruction reordering – i.e. instructions execute in the original (static) program order.
- Max Dependents - Nodes are given priority based on the number of dependent instructions currently residing in the node tables (i.e. the number of instructions waiting for the result that will be produced).
- Oldest First - Nodes are given priority based on original program order. A given node will have a higher priority than any of the nodes that follow it in the sequential execution order.
- Random Rotate- A random starting point in the node table is selected and scheduling proceeds in a rotating fashion until a ready node is found. This is very similar to "Fixed Priority" except that the starting point in randomized.
- Random - Nodes are chosen at random from the pool of ready nodes with no regard to their original order in the i stream.

While these techniques are not exhaustive, they do represent a wide range of possible heuristics. It should be noted that we do not feel that these "dynamic" techniques need to be, or even can be, realized completely in hardware. Several of the techniques, such as branch path, chain length, and maximum dependents require compiler support to provide priority information. In the simulations run in this paper, the true values for these priorities were calculated as if the hardware were capable of efficiently calculating it. This gives a slightly optimistic accuracy to priorities for these mechanisms since a compiler may not be able to accurately calculate these values across multiple dynamically–predicted branches.

# 3 Experiments

## 3.1 Benchmarks

The results presented in this paper are for seven integer and floating point programs from the SPEC suite: doduc, eqntott, espresso, gcc, li, fpppp, and tomcatv, compiled for and run under the M88000 instruction set architecture. The benchmarks were compiled using the Green Hills FORTRAN 1.8.5 compiler or the 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

| 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

| Functional 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 |
| **Node Table** | **Number of entries** |
| Split | 8 x 16 = 128 |
| Unified | 128 |

Table 2: Machine Configuration

without cpp and used the output of the preprocessor as the input file. Due to time limitations, each benchmark was simulated for the first ten million instructions.

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 functional units. The datapath contains eight pipelined functional units each capable of servicing different classes of instructions. The first three functional units are capable of performing load/store, bit field, and integer operations. The fourth functional unit can perform bit 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 functional units are capable of handling simple integer operations as well as another class of instructions.

## 3.3 Simulation Process

The simulation process works as follows:

An instruction level simulator for the MC88100 (ISIM) reads in the object code and simulates execution, producing an instruction trace. Our HPS simulator reads in a configuration file which describes the machine to be simulated and then begins processing the dynamic instruction stream produced by ISIM. The simulator performs a cycle by cycle simulation and gathers execution rate statistics.

Our simulator makes several simplifying assumptions:

- Register renaming is performed. Renaming eliminates anti and output dependencies and is critical for achieving high performance with the model of execution we simulated.
- Separate Instruction and Data caches are explicitly modeled in the simulator.
- Data cache bank conflicts are modeled. Only one request can be sent to a given bank per cycle.
- 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 functional units are fully pipelined (i.e. able to initiate a new operation each cycle) and are mutually independent.
- Instructions are removed from the window (i.e. "retired") in whole packet units after all instructions in that packet have completed execution. This assumption corresponds to our use of checkpointing [7] as a mechanism for supporting both branch prediction miss recovery and precise interrupts.
- 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.
- Context switches are modeled by draining the machine and flushing the cache every one million machine cycles. Assuming a processor speed of 100 MHz, one million cycles represents 10 milliseconds between context switches. Previous studies have noted that compute bound programs such as the SPEC benchmark applications studied here experience context switches almost entirely because of expiration of the quantum [9]. 10 milliseconds represents a reasonable quantum size [11].
- Branch Prediction - As branches are encountered in the dynamic instruction stream, a prediction is

made based on the history of that branch combined with dynamically gathered information[10]. This prediction is compared to the real outcome as determined by the trace. Depending on the simulator options settings, the simulator either stalls issue until the branch is resolved, or injects garbage instructions into the machine to reproduce the effect of proceeding along the mispredicted path.

There are several key parameters that define execution in the simulator.

- Window size - This parameter limits the total number of instructions that can exist in the machine at any one time. An instruction is considered in the machine, and thus occupying space in the window, from the time it is issued until it is retired. Instructions can be in one of four states: waiting for operands, ready but waiting for the assigned functional unit to become free, executing, or waiting for retirement. Window size is given in the number of issue packets.

- Functional Unit Configuration - The number and capabilities of all functional units are specified in the configuration file. Each functional unit is defined by the instruction classes it is capable of executing.

- Instruction Class Latency - These latencies describe the number of clock cycles required to execute a given type of instruction. The latencies we used are given in Table 1.

- Node Table Configuration - This flag indicates whether the node table entries should be treated as one common node table ("unified") or separate ("split") node tables, one for each functional unit. If the "unified" option is selected, the issue logic places instructions into the common node table without regard to instruction class and functional unit capability. These machines therefore achieve a higher issue "density" (average number of instructions issued).

- Dynamic Scheduling Policy - The scheduling discipline determines which nodes are selected from the pool of ready nodes.

### 3.3.1 Trace Driven Issues

Trace driven simulation is an important method of easily gathering performance statistics without becoming bogged down in the details of full simulation from an executable image. Traces however lack information that could potentially impact performance. In particular, when branches are mispredicted, the alternate (mispredicted) path is not known since the trace contains only the correct execution path. Actual traversal of the incorrect path can alter any machine by changing instruction and data caches as well as the BTB. With high branch prediction accuracy and large caches, these effects are likely to be small. Scheduling and execution however, may be more acutely affected by the injection of false paths. This effect is also likely to be minor for scheduling techniques such as "oldest first" since new nodes will not preempt older nodes that are ready (there is still a small effect due to contention for distribution buses when long latency operations collide with short latency operations in the same functional unit). Some scheduling techniques (e.g. random), however, are susceptible to interference from newly injected (mispredicted) instructions. If these instructions preempt instructions along the branch resolution path, performance will suffer.

In order to account for this interference effect and more accurately determine the performance of the various scheduling techniques, we have run simulations in which we inject "fake" instructions whenever the machine mispredicts a branch. These "fake" instructions are allowed to become ready (if they are not already ready at issue) and interfere through preemption with the nodes that are resolving the branch (and discovering the misprediction).

The "fake" instructions that are chosen to mimic the mispredicted path are those from the correct path. The assumption being that, to a first order approximation, both paths of the mispredicted branch contain similar numbers of ready nodes. In order to test the hypothesis that both paths look similar enough to make this assumption reasonable, we have performed a series of experiments constructing histograms of the number of ready nodes that follow branches in each direction and have found that while a small number of branches do have significantly different histograms, the majority of branches are very similar. We only have this information for branches that end up being both taken and not taken at some time throughout the course of execution. Branches that always go the same way however, are likely to be correctly predicted, and therefore need not be considered. "Fake" instructions do not affect machine state such as instruction and data caches, nor do they contribute to any performance statistics such as issue density and IPC. Except for scheduling interference, it is as if the machine had stalled issue from the time of misprediction to resolution of the branch.

# 4 Simulation Results

Figures 1 through 7 plot the performance, given in instructions executed per cycle (IPC), for each of the seven benchmarks. The chart shows the performance for each scheduling discipline for both split and unified

node table configurations. The dark bar shows the performance with no interference introduced by the mispredicted path (i.e. stalling issue until the branch is resolved). The white bar, labeled with the suffix "fe", is the performance when dummy instructions are thrown in and allowed to compete for resources.

## 4.1 Split vs. Unified Window

There are two primary mechanisms whereby a unified node table can outperform split node tables. The first is that with a unified node table, issue is not restricted to match incoming instructions with functional units on a one to one basis. In this way, the issue "density" and thus performance, can be higher for a unified window. Another potential performance advantage over split node tables arises when one (split) node table has no ready nodes while another node table has two or more nodes ready. These advantages come at the cost of more complicated (and costly) scheduling logic.

As can be seen from the performance graphs, however, these potential advantages do not significantly improve performance for the integer benchmarks. There is little difference between the performance of the split node tables and the unified node table. It turns out that the issue density does not increase significantly because issue still breaks when the first branch is reached regardless of node table configuration. Branches are overwhelmingly the single largest reason for issue breaks in the integer benchmarks.

The floating point benchmarks realize a noticeable performance increase from unified node tables. This arises largely from the increase in issue density. Fpppp goes from 3.8 instructions per issue to 7.4 out of a maximum issue rate of 8. With a single fp adder and single fp multiplier, issue often breaks because of consecutive fp operations, as well as strings of consecutive memory operations.

## 4.2 Scheduling Discipline

As can be seen from the performance charts, there is very little variation in performance for the different scheduling techniques. This behavior is consistent across the integer benchmarks with slightly more variation exhibited by the floating point benchmarks. The exception is the performance of "head" which consistently and substantially lags the others.

Aside from "head", which achieves as low as 44 percent of the performance of the best scheduling heuristic, the different scheduling techniques perform within 5 percent of each other on the integer benchmarks. The floating point benchmarks, led by Tomcatv, show differences of up to 20 percent, again excluding "head" which underperforms by as much as 80 percent.

One reason for the similar performance is the low average number of ready nodes in the node tables. Figures 8 and 9 show histograms of the number of ready nodes in each (split) node table and in the unified node table for gcc, the poorest performing benchmark, running under oldest first scheduling. As can be seen, the distribution of ready nodes is heavily skewed to zero and one, so there is little opportunity for different scheduling techniques to give rise to different schedules. This should not be taken to mean however that there is no need to allow out of order execution. There tends to be few things ready each cycle, but they do not usually come ready in the same order as statically scheduled. This can be seen in the performance of the "head" scheduling discipline.

It is important to keep in mind when comparing the performance of the different techniques that "Branch Path", "Chain Length", and "Maximum Dependents" resort to oldest first scheduling to break ties. In some cases, such as "Branch Path" scheduling and the branch–sparse Fpppp benchmark, these techniques degenerate into "oldest first."

One interesting point to note is that the efficiency of "oldest first" scheduling (and those that preserve an oldest first bias) is relatively insensitive to mispredicted path interference and thus branch prediction accuracy.

## 4.3 Analysis

Dynamic scheduling is a hardware intensive approach to achieve higher performance. While it has a necessarily smaller scope of instructions to schedule than static scheduling, it has the benefit of dynamic information which can affect the schedule. In particular there are two aspects of high performance execution that give rise to scheduling alterations. The first is speculative execution that arises from (potentially more accurate) dynamic branch prediction. Dynamic scheduling allows the instructions from before the branch and those from the predicted path (even across multiple predictions) to be scheduled together as a unit.

Another important aspect of high performance execution that can alter a static schedule is the effect of non–deterministic latencies. Data cache misses often result in long latency fetches from memory that disrupt the schedule. As memories get longer relative to CPU cycle time, and as multiprocessing systems become more prevalent, this effect is likely to become more dramatic.[1]

The scheduling technique which gives rise to the highest performance varies for the different benchmarks. In general, "Branch Path", "Maximum Dependents", and "oldest first" perform among the best and tolerate mispredicted paths well. The hardware required to schedule according to these priority schemes, however, may not be justified when compared to "random" scheduling.

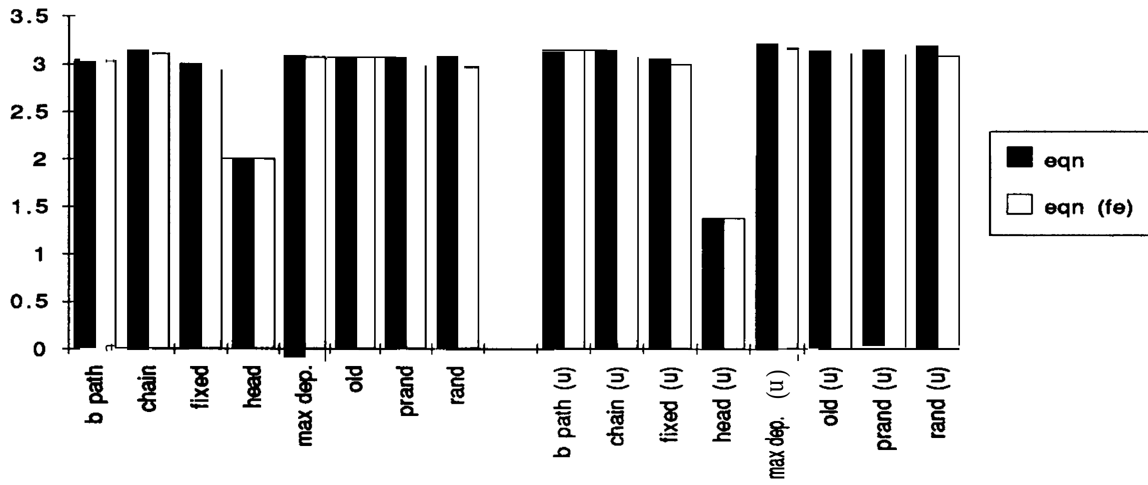


Figure 1: Eqntott

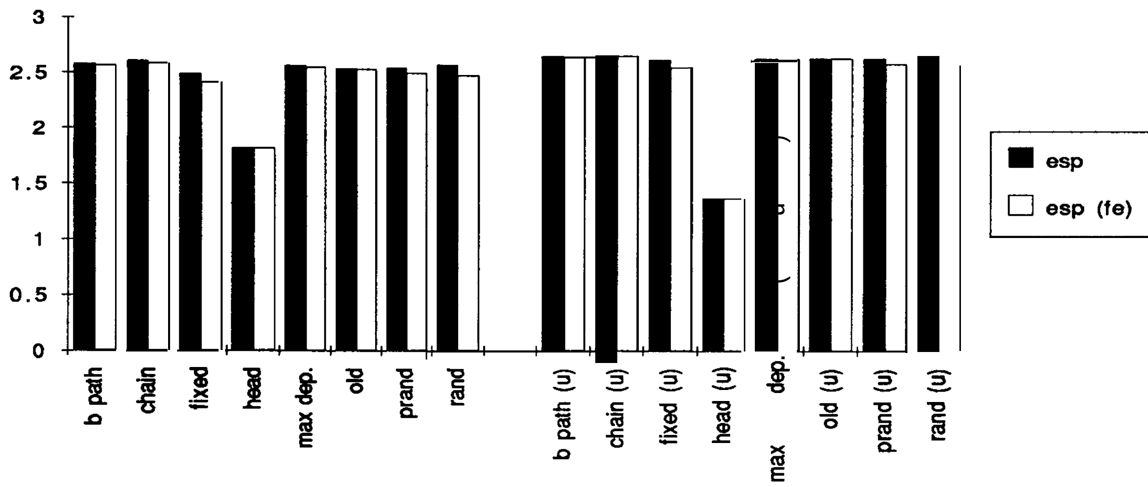


Figure 2: Espresso

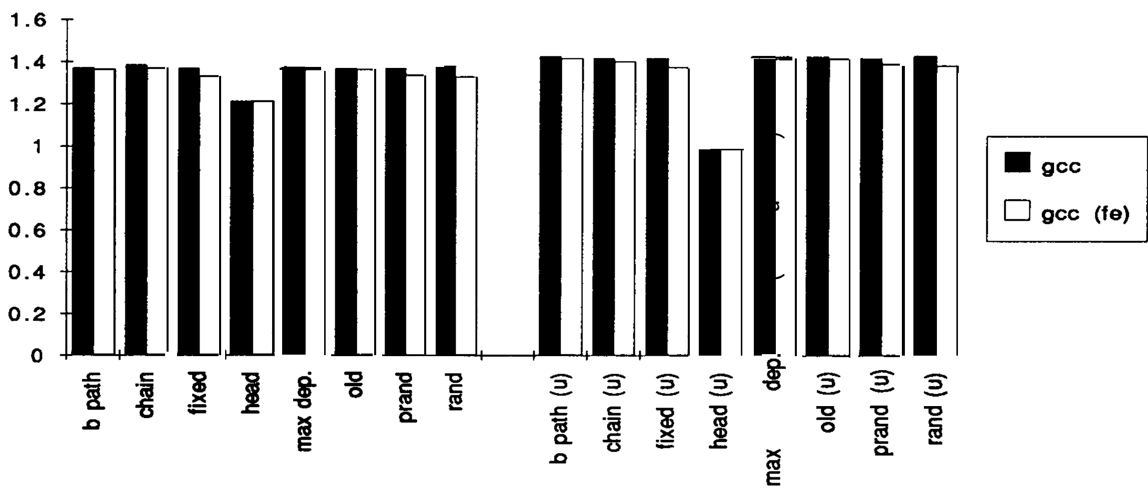


Figure 3: GCC

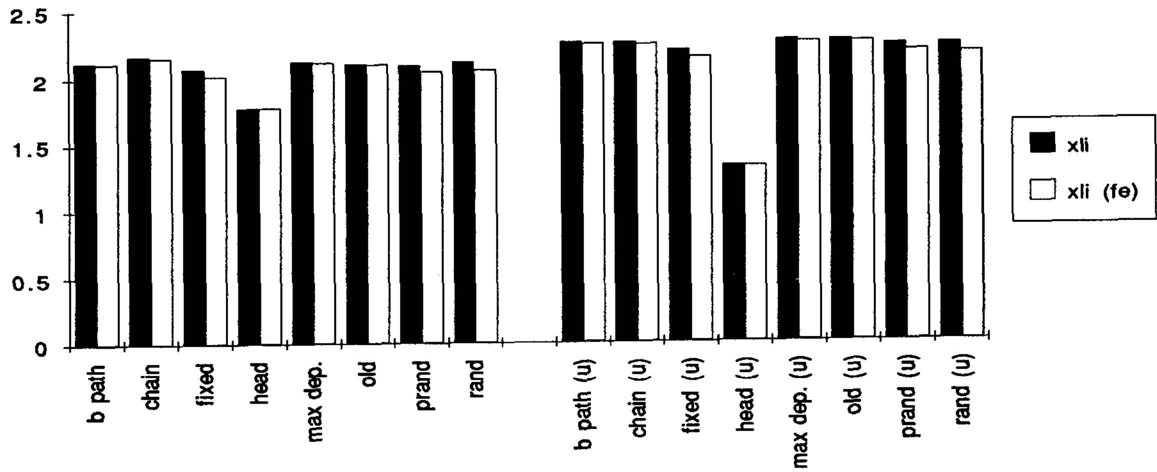


Figure 4: Xlisp

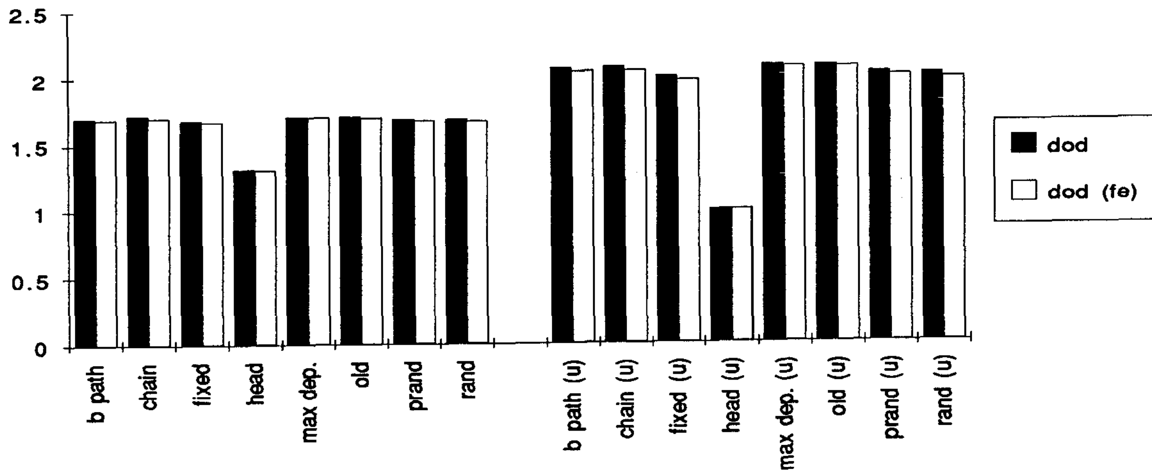


Figure 5: Doduc

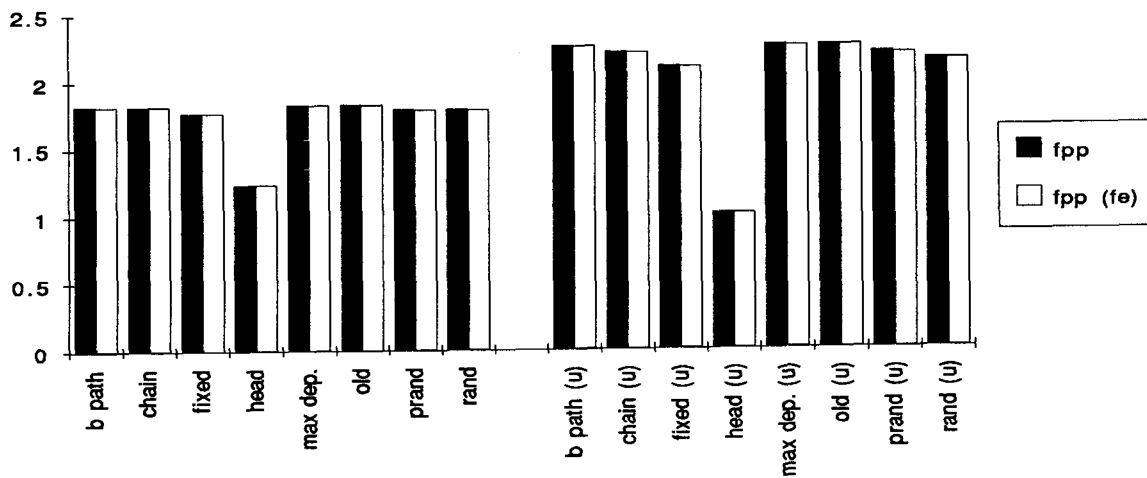


Figure 6: Fpppp

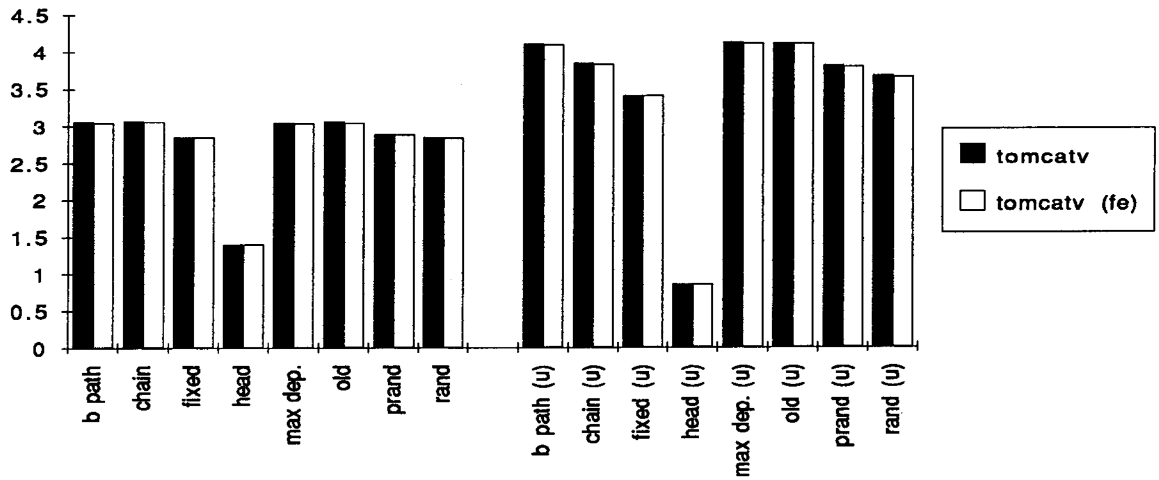


Figure 7: Tomcatv

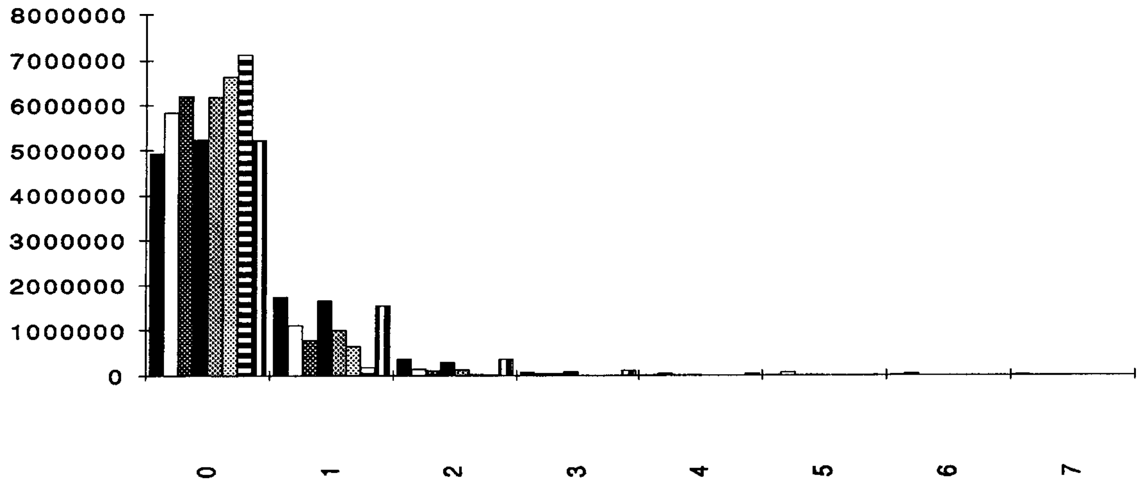


Figure 8: Number of Ready Nodes in each of the Split Node Tables

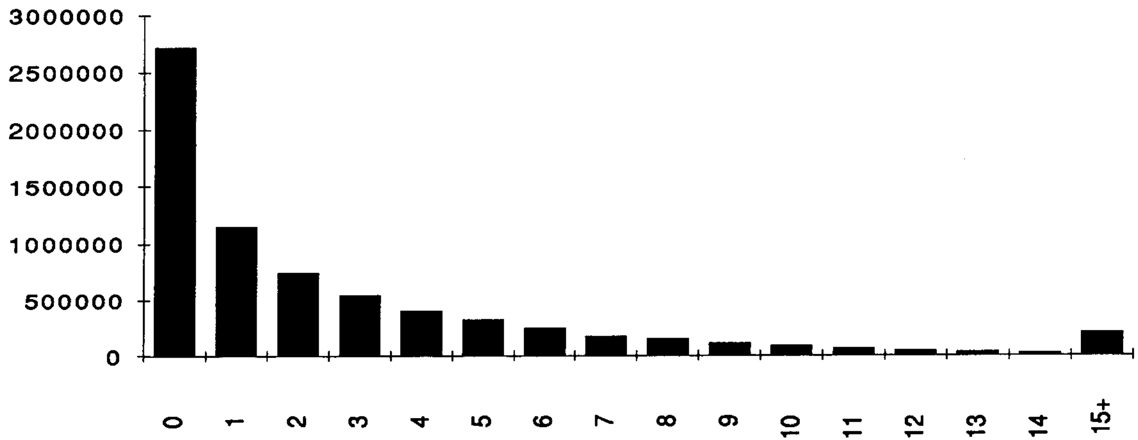


Figure 9: Number of Ready Nodes in Unified Node Table

# 5 Concluding Remarks

Microarchitectures that support dynamic scheduling demonstrate an ability to tolerate variable latencies such as data loads from a cache as well as adjust schedules to incorporate speculative execution. This ability comes at the cost of varying amounts of hardware support. We have found that for the scheduling techniques modeled, performance is largely independent of scheduling discipline. The variation between scheduling techniques ranges from 3 to 5 percent for the integer benchmarks and 2 to 20 percent for the floating point benchmarks. This suggests that the simpler and less hardware–intensive techniques are most attractive. Furthermore, the integer benchmarks show little performance is gained from going to a unified node table. Floating point benchmark performance, however, due to the increased issue density, improves by approximately 25 percent over a split configuration.

The results presented here are for a wide issue (eight) machine with a reasonably large window. This aggressive machine was chosen to allow enough "room" for instruction scheduling to play out 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 scheduling on larger 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 Mark Campbell, including the gift of an NCR Tower multiprocessor which we have used in some of 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 Steve Melvin and the rest of our research group at Michigan.

# References

[1] K. Gharachorloo, A. Gupta, and J. Hennessy, "Hiding Memory Latency using Dynamic Scheduling in Shared-Memory Multiprocessors ," *Proceedings of the 19th International Symposium on Computer Architecture*, (May 1992), pp.22-33.

[2] 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.

[3] D. Wall, "Limits of Instruction-Level Parallelism," *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, (April 1991), pp.176-188.

[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, M. Shebanow, and Steve Melvin, "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 Annunal Symposium on Computer Architecture*, (June 1986), pp.297-307.

[7] W.W. Hwu and Y.N. Patt, "Checkpoint Repair for Out-of-order Execution Machines," *IEEE Transactions on Computers*, (December 1987), pp.1496-1514.

[8] R.M. Tomasulo, "An Efficient Algorithm for Exploiting Multiple Arithmetic Units," *IBM Journal*, Vol. 11, (January 1967), pp. 25-33.

[9] J. Mogul, and A. Borg, "The Effect of Context Switches on Cache Performance," *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, (April 1991), pp.75-84.

[10] Tse–Yu Yeh and Y.N. Patt, "Two–Level Adaptive Branch Prediction," Technical Report CSE-TR-117-91, Electrical Engineering and Computer Science department, University of Michigan, (November 1991).

[11] S. Leffler, M. McKusick, M. Karels, and J. Quarterman, *"The Design and Implementation of the 4.3BSD Unix Operating Systems,"* Addison-Wesley, 1989.