PERFORMANCE FACTORS FOR SUPERSCALAR PROCESSORS

James E. Bennett
Michael J. Flynn

Technical Report No. CSL-TR-95-661

February 1995

The research described herein has been supported by NASA-Ames under grant NAG2-248, using equipment supplied by Silicon Graphics, Inc.

PERFORMANCE FACTORS FOR SUPERSCALAR PROCESSORS

by

James E. Bennett

Michael J. Flynn

Technical Report No. CSL-TR-95-661

February 1995

Computer Systems Laboratory

Departments of Electrical Engineering and Computer Science

Stanford University

Stanford, California 94305-4055

pubs@shasta.stanford.edu

Abstract

This paper introduces three performance factors for dynamically scheduled superscalar processors. These factors, availability, efficiency, and utility, are then used to explain the variations in performance that occur with different processor and memory system features. The processor features that are investigated are branch prediction depth and following multiple branch paths. The memory system features that are investigated are cache size, associativity, miss penalty, and memory bus bandwidth. Dynamic scheduling with appropriate levels of bus bandwidth and branch prediction is shown to be remarkably effective at achieving good performance over a range of differing application types and over a range of cache miss rates. These results were obtained using a new simulation environment, MXS, which directly executes the benchmarks.

Key Words and Phrases: Dynamic scheduling, Branch prediction, Memory latency, Non-blocking cache, Speculative execution.

Copyright © 1995
by
James E. Bennett
Michael J. Flynn

Contents

1 Introduction **1**

2 Experimental Procedure **1**

2.1 The processor model 1

2.2 The simulation process 2

2.3 The benchmarks 3

2.4 The experiments 4

3 Performance factors **5**

4 Results **6**

4.1 Branch prediction level 6

4.2 Branch prediction accuracy 11

4.3 Varying memory bandwidth 13

4.4 Differing cache designs 14

5 Discussion **18**

5.1 Related work 18

5.2 Branch prediction 19

5.3 Compiling for dynamically scheduled machines 19

6 Conclusion **19**

List of Figures

1 The Simulation Process 3

2 Performance vs. Speculation 6

3 *Uncompress*: Performance factors 7

4 *FFT*: Performance factors 8

5 *Spice*: Performance factors 9

6 *Xlisp*: Performance factors 10

7 Performance vs. Miss penalty 15

8 Performance vs. Miss rate 16

9 Performance vs. Miss rate 17

List of Tables

1 Operation Latencies 2

2 Benchmarks 4

3 Branch frequency by category 11

4 Branch prediction accuracy 12

5 IPC as memory bandwidth varies 13

6 IPC for various cache configurations 14

7 Fetch availability vs. Miss penalty 15

8 Miss ratios 16

1 Introduction

This paper investigates the ability of dynamic scheduling to achieve good performance in a superscalar processor, over a range of processor design and memory system design options. To help understand the conditions which give rise to good performance in a dynamically scheduled superscalar processor, we introduce three performance factors: availability, efficiency, and utility. These factors are then used to explain the variations in performance that occur with different processor and memory system features.

Because of cycle time constraints, the size of the primary cache is likely to remain limited[HP90] in future processors. The miss rate will accordingly remain significant. A secondary cache can be used to reduce the cost of primary cache misses, but in a processor capable of executing multiple instructions per cycle, these misses can still have a major impact. Dynamic scheduling has been proposed as an effective technique for tolerating memory latency due to cache misses[CCMH91, BP91, GGH92].

To investigate the factors that contribute to good performance in the presence of memory latency, we study the performance for a fixed primary cache organization, as the branch prediction depth varies. From the results of these runs, a particular processor configuration is chosen, and its ability to tolerate memory latency is tested by varying the memory system. The features of the memory system that are varied are the size, associativity, miss penalty, and bandwidth of the primary data cache. Dynamic scheduling with appropriate levels of bus bandwidth and branch prediction is shown to be remarkably effective at achieving good performance over a range of differing application types and over a range of cache miss rates.

This paper is organized as follows. In section 2, the experimental procedure is outlined, with descriptions of the processor model, the simulation environment and the benchmarks. The factors of performance are introduced in section 3. Section 4 describes the results of the simulation runs. Section 5 discusses these results and related work on techniques for tolerating memory latency. Finally, section 6 presents some conclusions and plans for future work.

2 Experimental Procedure

2.1 The processor model

The processor model is an eight issue superscalar processor. The model is parameterized in terms of the bandwidth and latency of its various components. The processor timing is based on a shallow pipeline design, where cache accesses can be completed in a single cycle (see table 1). The timing is similar to the MIPS R3000[KH92], except for floating point latencies. All operations are assumed to be fully pipelined.

Each cycle, up to eight instructions can be fetched, up to eight instructions can begin execution, and up to eight instructions can complete (write their results to the register file). Registers are

Operation	*Latency*
Load	2 cycles
Branch	2 cycles
Int. Multiply	12 cycles
Int. Divide	35 cycles
Other int. op	1 cycle
FP Add	2 cycles
FP Multiply	3 cycles
FP Divide	12 cycles

Table 1: Operation Latencies

renamed to eliminate output (WAW) and anti- (WAR) dependencies. Instructions begin execution as soon as their operands become available, up to the limit of eight.

The instruction window for this model is defined as the number of instructions that have been fetched and have not yet begun execution. This is the number of instructions that need to be examined when determining which instructions are ready to begin execution, and includes the instructions which are waiting in reservation stations[Tom67]. To allow sufficient scope for the dynamic scheduling, the instruction window is set to 32 for all of the tests in this paper.

The primary data cache varies in size and degree of associativity, with a constant line size of 16 bytes. The policy on write misses is to first read in the cache line, and then perform the write (write miss allocate). Data from the primary cache is written out to the secondary cache when a miss occurs on a line which has been modified (a dirty line). The time to read a cache line from the secondary cache into the primary cache varies from four to eight cycles, and the time to write out a dirty cache line is the same. The number of outstanding requests allowed to the secondary cache varies.

The number of unresolved branches allowed varies from 0 to 8 branches.

Branch prediction is implemented as a 256 entry table of 3-bit saturating counters. This is a minor variation of the traditional 2-bit branch prediction scheme[Smi81]. Bits 2–9 of the branch address are used to index a table of 3-bit counters, and if the value found there is greater than 3, the branch is predicted taken, otherwise it is predicted not taken. When the branch is resolved, if the branch was taken, the counter is incremented, otherwise it is decremented. The value of the counter saturates at 0 and 7 (no wraparound occurs). There is also a 256 entry table of branch target addresses for predicting the destination address of indirect branches.

2.2 The simulation process

All the results in this study were obtained using the MXS simulation environment[BF94]. Figure 1 shows the simulation framework. The benchmark is first compiled and then it is executed by the simulator. During execution, the simulator maintains counts of various statistics of interest, such as number of loads, stores, cache misses, etc. After the execution is completed, the statistics are

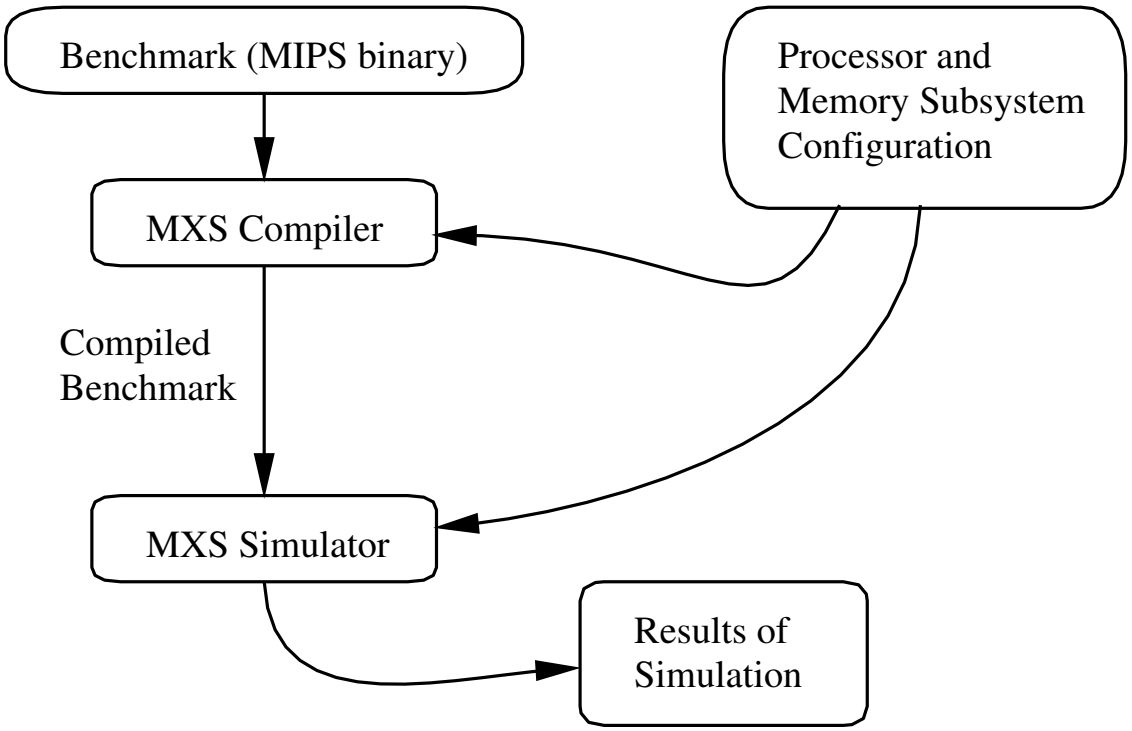

Figure 1: The Simulation Process

written out to a results file. Both the compiler and the simulator are configured by means of a parameter file which describes the processor and memory subsystem being modeled.

When using trace based simulation the addresses of speculatively generated loads and stores are not available. The execution based simulation technique used in this study correctly generates all memory references, both speculative and non-speculative.

For all these studies the compiler performed a simple translation from the MIPS binary to the format that the simulator expects. No rescheduling was performed, and MIPS instructions were mapped directly into instructions supported by the simulator (that is, no expansion in the number of instructions occurred). This is the performance that would be obtained when using a dynamically scheduled processor to run existing binaries. Increased performance beyond that reported here might be obtained by recompilation. Compiler issues are discussed in section 5.3.

The MIPS instructions set is similar to a number of other RISC instruction sets. The instruction per cycle (IPC) numbers presented here should be applicable to any of the machines in this class.

2.3 The benchmarks

The following benchmarks were used for evaluating the performance of the different processor models: Compress, Uncompress, Espresso, FFT, Linpacks, Sc, Spice, Wave, and Xlisp. They were chosen to represent a variety of distinct workloads and reference patterns. Some summary statistics on these benchmarks are presented in table 2. Length is the number of instructions executed in one

Benchmark	*Length*	*FP*	*Branch*	*Memory*	*Miss rate*
compress	2.39 M	0.0%	11.2%	33.7%	10.1%
uncompress	1.80 M	0.0%	13.3%	33.9%	3.5%
espresso	41.64 M	0.0%	14.3%	22.8%	1.4%
sc	46.30 M	0.0%	17.6%	31.3%	4.5%
xlisp	14.61 M	0.0%	12.6%	38.8%	2.4%
fft	3.39 M	29.9%	3.0%	38.8%	16.2%
linpacks	68.85 M	28.7%	6.5%	45.4%	7.2%
spice	99.21 M	7.2%	8.8%	34.9%	11.2%
wave	34.30 M	22.7%	5.0%	34.2%	9.8%

Table 2: Benchmarks

run of the benchmark, and the miss rate is the miss rate with an 8K 4-way set associative cache.

Each of these benchmarks was compiled with the default compilers (C and Fortran) provided with IRIX[1] version 5.1. The -O and -non_shared options were set. The default optimizations enabled by -O include loop unrolling, which helps to explain the low frequency of branches in the floating point benchmarks. The -non_shared option forces the libraries to be linked in with the application binary, which is needed for the simulator to be able to execute the library code.

These benchmarks were taken mostly from the SPEC 92 benchmark suite, but the inputs were reduced so that the simulator could run the benchmark to completion. Correspondingly reduced cache sizes were used to obtain a realistic range of cache miss rates. The non-SPEC benchmarks are FFT and Linpacks. FFT performs a one dimensional fast Fourier transform on an array of single precision floating point data of length 1024. Linpacks is the single precision version of the standard linear algebra benchmark.

2.4 The experiments

In the first set of experiments, the number of outstanding branches allowed was varied. Then a set of tests was run to determine the effectiveness of fetching along multiple alternative paths. From the results of these runs, a particular processor configuration was chosen, and its ability to tolerate memory latency was tested by varying the memory system.

To bracket the performance gains attributable to latency tolerance, the performance of a machine with no miss penalty is calculated, and also the performance of a machine that stalls on cache misses. The performance is then determined for different assumptions about the bandwidth of the interface between the primary and secondary caches. This portion of the experiment is modeled after the study that Farkas and Jouppi did on scalar processors[FJ94]. Then these results are tested by varying the cache size, degree of associativity, etc., in order to determine the effectiveness of this scheme over a range of cache sizes and designs.

The figure of merit used to compare different models is IPC, the number of instructions per cycle.

[1]IRIX is a trademark of Silicon Graphics, Inc.

This figure is computed by taking the number of instructions executed when running the benchmark on a *non-speculative* model, and dividing it by the number of cycles required to execute the benchmark with the given model. No-op instructions are not included in the instruction count.

3 Performance factors

In order to gain more insight into this behaviour, the following three performance factors were defined: fetch efficiency, fetch utility, and fetch availability. These factors are defined from the point of view of the instruction fetch mechanism. The instruction window is a buffer between the instruction fetch unit and the issue/execute mechanism. When the fetch unit is 100% efficient, then it is successfully filling all of the available slots in the instruction window each cycle.

Utility measures how many of the instructions that were fetched were truly needed in the computation. Loss of utility occurs when there are lots of no-ops in the instruction stream, or when instructions are fetched along an incorrectly speculated path.

Availability measures how much room there is left in the instruction window each cycle. If the fetch unit is more efficient than the issue/execute mechanism, then the instruction window will fill up. This results in a loss of availablility; the fetch unit could fetch more instructions, but there aren't enough available slots in the instruction window.

More precisely, the factors are defined as follows:

The total number of fetch slots in a given run is the fetch bandwidth times the number of cycles in the run. For example, if the fetch bandwidth is 8, and a benchmark runs for 1,000 cycles, then there are 8,000 fetch slots in the run (throughout this paper, the fetch bandwidth is fixed at 8). Out of these 8 fetch slots each cycle, some cannot be filled because the instruction window is full. The remaining fetch slots are known as the *available fetch slots*. Fetch availability is defined as the ratio of available fetch slots to the total number of fetch slots.

Even when it has an available fetch slot, the fetch unit isn't necessarily able to fill it. For example, the fetch unit might be stalled waiting for a branch to be resolved. *Fetch efficiency* is defined as the ratio of the number of instructions fetched to the number of available fetch slots.

Of the fetched instructions, not all of them are necessary to the execution of the program. They might have been fetched speculatively, and never executed. *Fetch utility* measures how appropriate or useful the fetched instructions were. It is defined as the ratio of the number of instructions executed in a *non-speculative* model to the number of instructions fetched in a given run.

The performance can be calculated from the three factors, as follows:

$$IPC = 8 * Availability * Efficiency * Utility$$

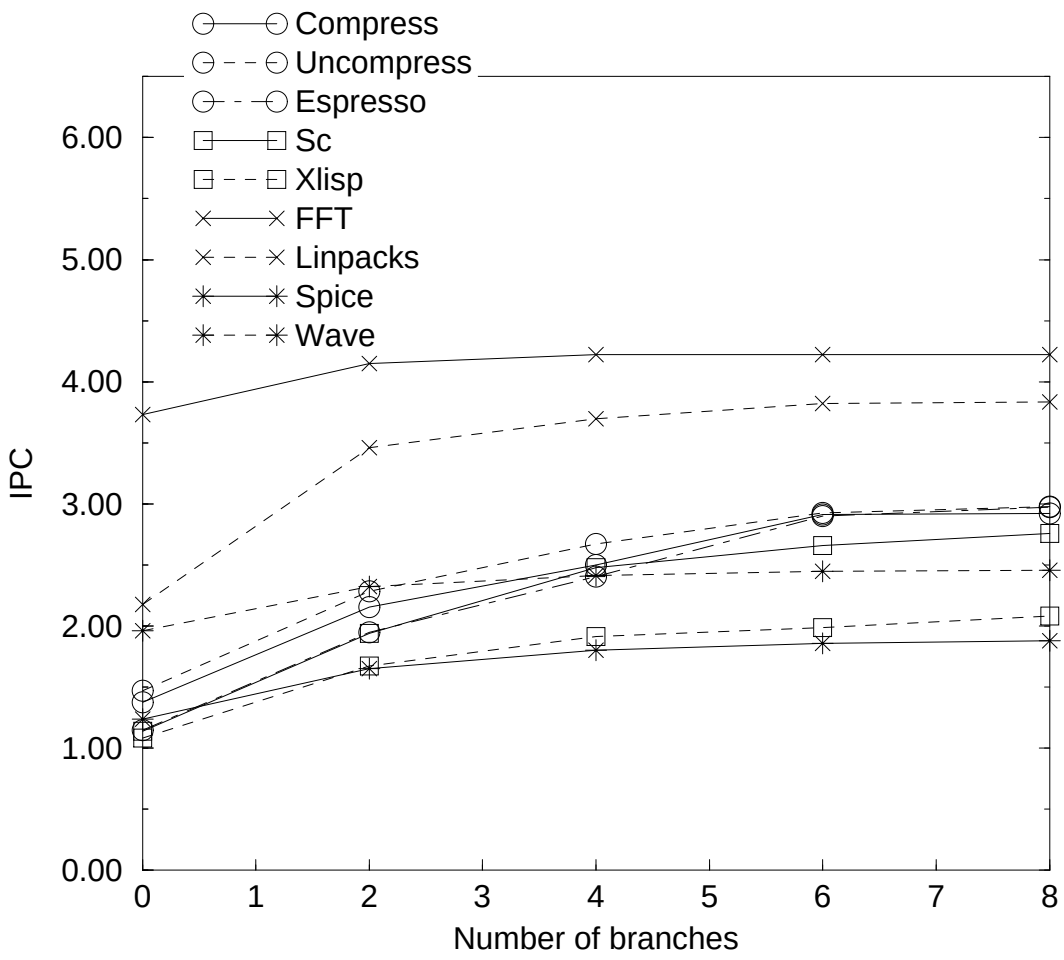

Figure 2: Performance vs. Speculation

4 Results

4.1 Branch prediction level

In figure 2, the performance of the processor model on the various benchmarks is shown for differing levels of speculation. The performance appears to flatten out at about 6 unresolved branches for the integer benchmarks, and at 4 branches for the floating point benchmarks. This is consistent with the lower frequency of branches in the floating point benchmarks.

The performance factors for the Uncompress benchmark are shown in figure 3, along with their product (which is $IPC/8$), expressed as percentages. The pattern shown in this figure is typical of the benchmarks in this study. Fetch efficiency increases as the level of branch prediction increases, because the processor spends less time waiting for branches to be resolved. This is offset somewhat by decreasing fetch utility, as instructions are fetched which have a lower probability of being on the true path when more branches are predicted. The overall effect, however, is to increase performance.

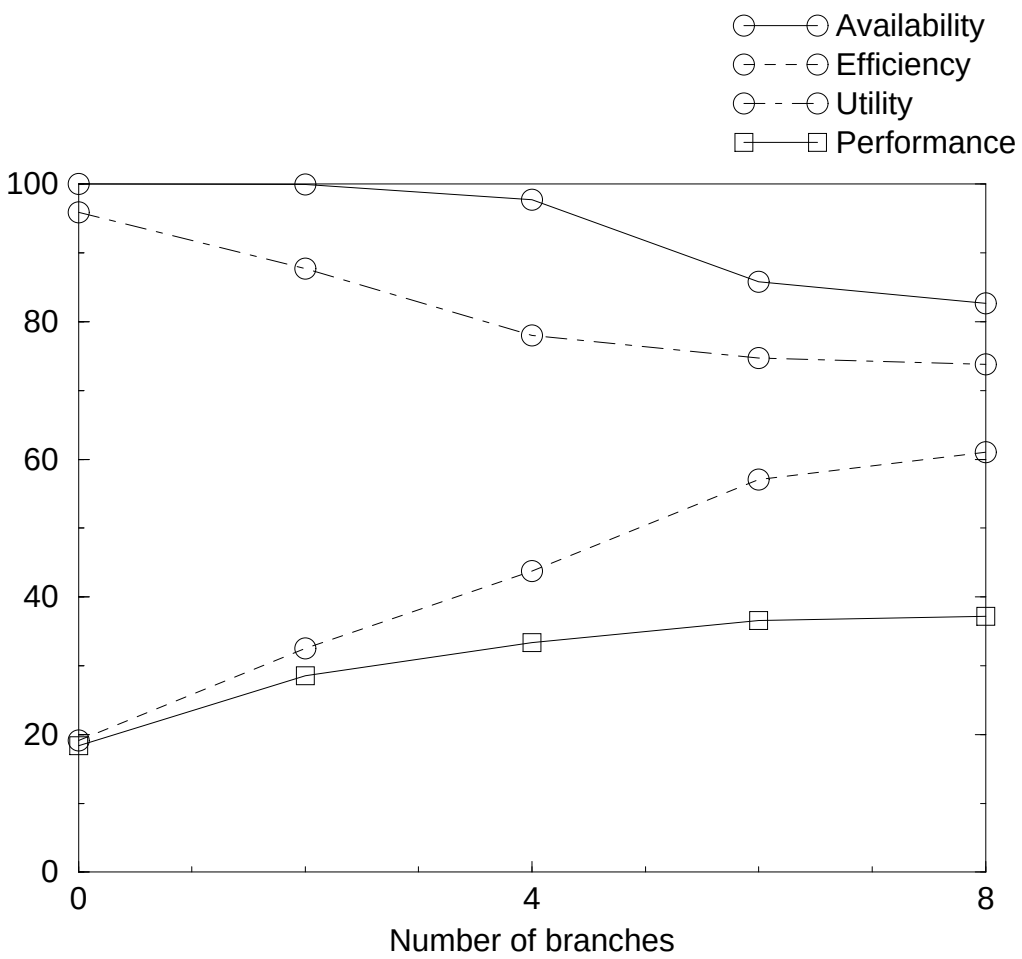

Figure 3: *Uncompress*: Performance factors

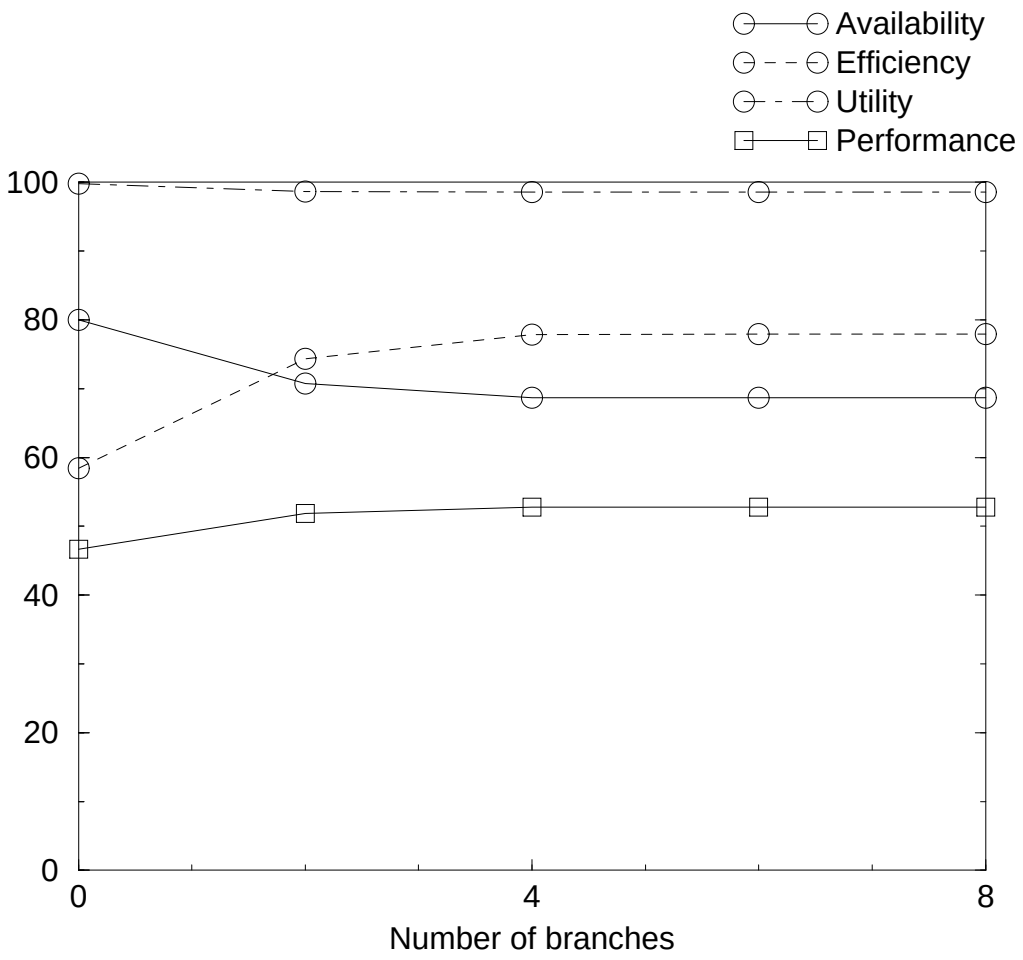

Figure 4: *FFT*: Performance factors

As more branches are predicted, the instruction window fills up more often, and this combined with the decreasing fetch utility, causes performance to flatten out after around six predicted branches. The improvement in fetch efficency also flattens somewhat as the level of branch prediction increases past six. Whenever a branch is encountered, fetching is stalled at least until a branch prediction is made, which means that the remaining fetch slots that cycle are lost. Thus the high frequency of branches in this benchmark places an upper bound on the fetch efficency.

The best performing benchmark overall was FFT. Figure 4 displays the performance factors for this benchmark. With increasing branch prediction level, efficiency increases, but availability decreases. This indicates that the fetch unit is more often successful in filling up the available slots in the instruction window, but that the window becomes full more often as more branches are predicted.

This benchmark is unusual in that the instruction window becomes full even when no branch prediction is performed. This indicates that the fetch unit is racing ahead of the computations, as in a decoupled architecture[Smi84]. This can happen whenever the branches are independent of the results of the computation, and the dependency chain from one branch to the next is shorter than the dependency chain through the computations.

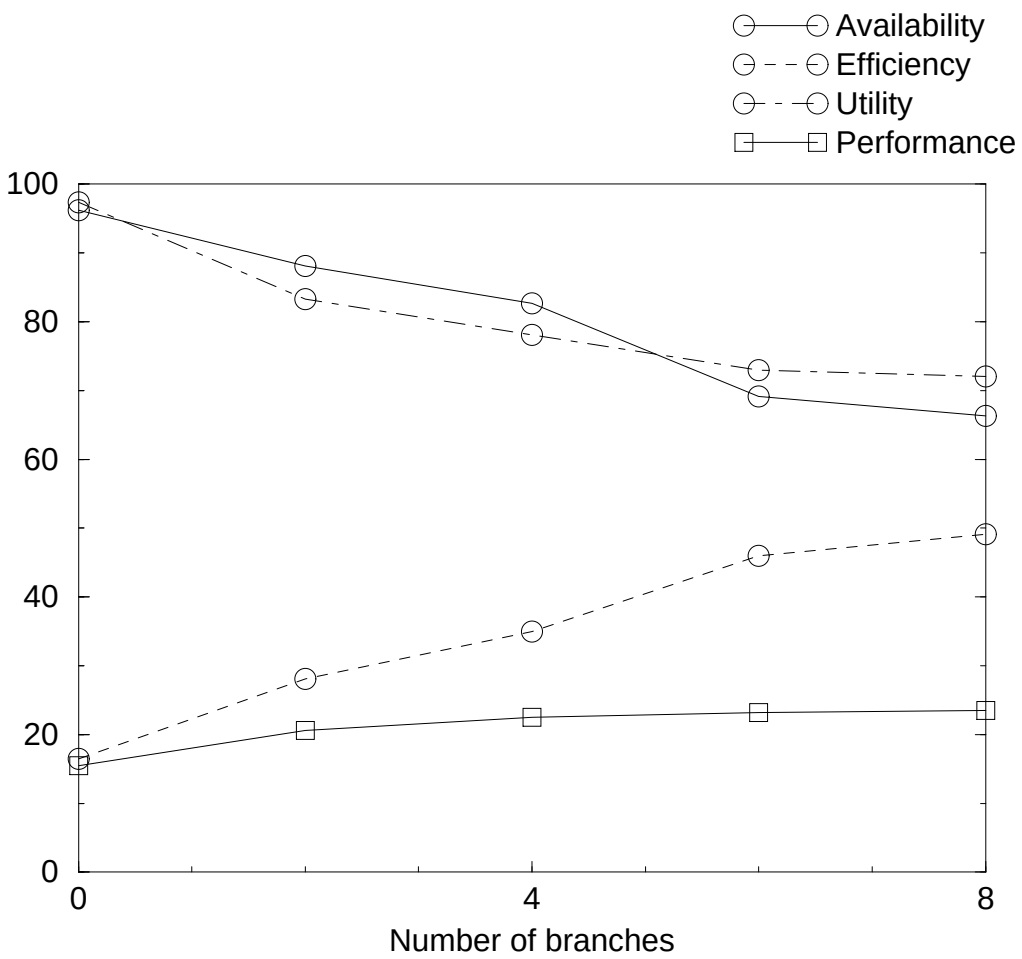

Figure 5: *Spice*: Performance factors

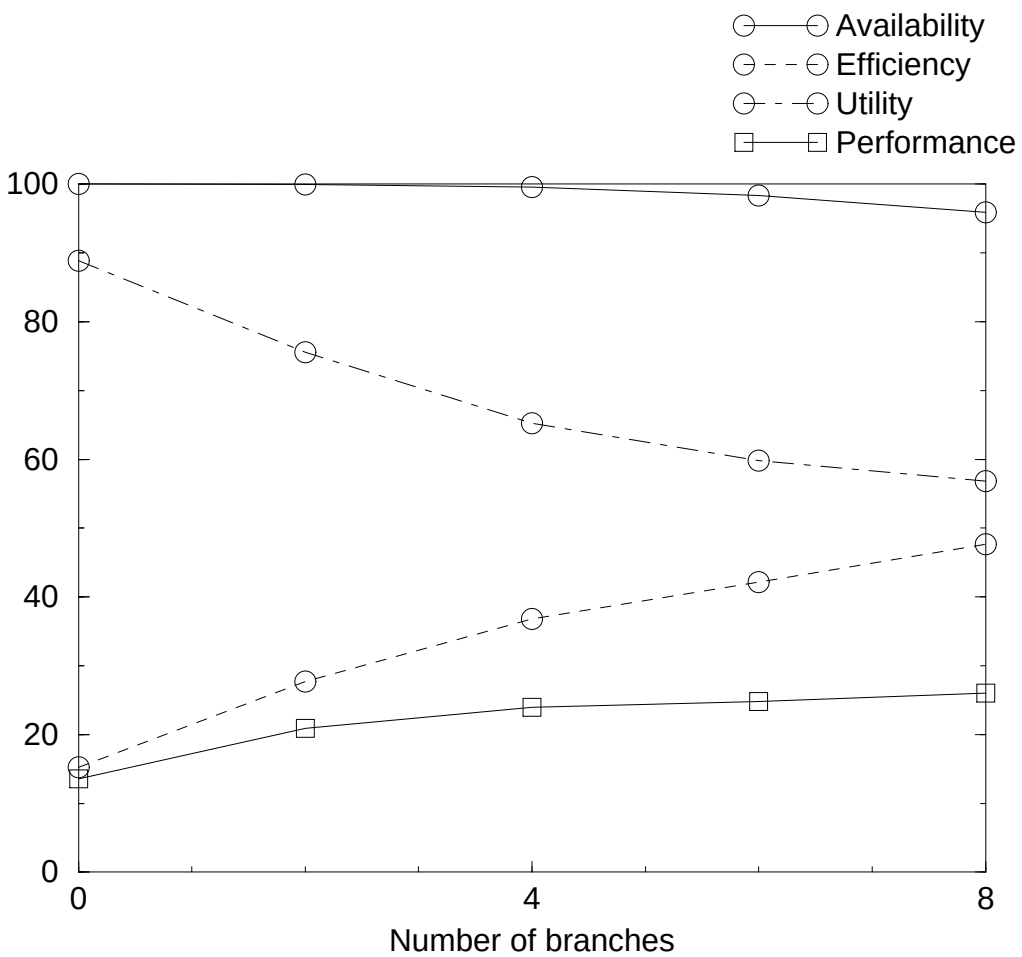

Figure 6: *Xlisp*: Performance factors

Benchmark	*Taken*	*W. Take*	*W. Fall*	*Fall-thru*	*Indirect*
compress	68.5%	1.7%	3.2%	26.4%	0.2%
uncompress	70.9%	1.3%	0.4%	16.2%	11.1%
espresso	61.2%	3.4%	2.9%	23.8%	8.7%
sc	55.9%	2.9%	2.7%	30.0%	8.6%
xlisp	41.3%	3.7%	2.2%	35.9%	16.9%
fft	70.7%	0.4%	0.2%	23.2%	5.5%
linpacks	69.4%	0.4%	0.1%	22.7%	7.3%
spice	49.0%	4.2%	2.1%	28.7%	16.1%
wave	58.6%	1.5%	1.8%	28.1%	10.0%

Table 3: Branch frequency by category

Spice and Xlisp have the lowest overall performance of all of the benchmarks, so it is interesting to look at their performance factors.

In the Spice benchmark (figure 5) we see that while the fetch efficiency continues to rise, especially between 4 and 6 predicted branches, it is accompanied by a corresponding decline in availability. This indicates the instructions aren't being issued as fast as they are being fetched, because there isn't enough parallelism within the instruction window. The fetch utility also declines steadily as the number of predicted branches increases. The end result is little or no performance increase.

In Xlisp (figure 6), on the other hand, availability remains high, and the flattening of performance is the result of the modest rate at which efficiency is increasing, and poor utility. The poor utility is due to the fact that branch prediction accuracy on this benchmark is worse than on the other benchmarks (see section 4.2). The low efficiency is caused by the relatively high branch frequency, and because branches are taking longer in this benchmark to be resolved. This second effect is demonstrated by the continued increase in efficiency as the level of branch prediction is increased from six to eight.

These two cases illustrate two different constraints on instruction level parallelism: Limits due to data dependencies in the case of Spice, and limits due to control dependencies in the case of Xlisp.

4.2 Branch prediction accuracy

Branch prediction affects the performance of the dynamically scheduled processor primarily by its impact on fetch utility. The better the branch predictor, the lower the probability that instructions will be incorrectly fetched, and the higher the fetch utility. The branch prediction mechanism in this model achieves about 90% accuracy, on average. This average can be broken down according to branch type and benchmark.

The 3-bit counter is used to classify branches into four categories. A branch is predicted taken, weakly taken, weakly not taken, or not taken, as the counter takes on the values 6 or 7, 4 or 5, 2 or 3, or 0 or 1, respectively. A fifth type of branch is the indirect branch, or branch to a location whose address is in a register. In the instruction set of the simulator indirect branches are always

Benchmark	*Taken*	*W. Take*	*W. Fall*	*Fall-thru*	*Indirect*	*Overall*
compress	97.3%	51.8%	64.1%	81.5%	89.3%	91.3%
uncompress	92.6%	68.2%	52.2%	90.0%	85.8%	90.9%
espresso	93.7%	60.7%	53.8%	91.9%	80.8%	89.8%
sc	92.1%	60.2%	55.2%	92.1%	80.7%	89.2%
xlisp	82.1%	61.8%	59.4%	89.2%	64.4%	80.4%
fft	96.3%	61.0%	60.4%	89.0%	98.9%	94.5%
linpacks	92.6%	29.3%	53.6%	99.5%	98.1%	94.3%
spice	88.3%	60.5%	47.7%	89.8%	84.3%	86.1%
wave	90.7%	59.8%	63.2%	94.1%	88.8%	90.5%

Table 4: Branch prediction accuracy

unconditional. The frequencies of the various kinds of branches is given in table 3.

The effectiveness of the branch prediction mechanism is given in table 4. The accuracy of the predictions is given for each of type of branch. In the case of indirect branches, the accuracy refers to how often branch target addresses in the table agree with the correct address. These numbers were taken from the run in which the branch prediction level was set to six. The values vary slightly from run to run, since at the time the prediction is made, different numbers of unresolved branches may be outstanding.

The 3-bit counter does a good job of classifying the branch instructions according to whether they are predictable or not. The weakly predicted branches have an overall prediction accuracy of around 60%, not much better than a coin flip. Fortunately there are relatively few branches in the weakly predicted categories, so the overall branch prediction accuracy is quite high.

The suggestion has been made that following multiple paths of execution is a good way to increase instruction level parallelism[Uht93]. However for a fixed level of instruction fetch bandwidth, it would seem that the optimal strategy is to fetch only along the most probable path.

Assuming for the moment that fetch availability and fetch efficiency are independent of the choice of which path to follow, then the path with higher probability will yield the better fetch utility and thus the better performance. If the fetch bandwidth is to be divided between two paths, then fetches along the path of lower probability are displacing fetches along the higher probability path, and so performance on the average is reduced.

However, fetching along multiple paths does have some second order effects. The instructions along distinct paths are independent of each other, so the parallelism in the instruction window is increased. This may be of some benefit if the instruction window is filling up frequently (i.e if availability is low), as it will increase the rate at which instructions are removed from the window by execution.

Thus if a branch has a roughly even chance of being taken, then it might make sense to follow both paths. Using the branch prediction bits, branches can be classified according to whether they are strongly or weakly predicted. However the vast majority of branches are strongly predicted, as shown in table 3, so any benefit from following multiple paths will be limited by lack of opportunity.

Benchmark	*Blocking*	*fc=1*	*fc=2*	*fc=3*	*fc=4*	*Unlim*	*No lat.*
compress	1.95	2.32	2.78	2.84	2.88	2.91	3.31
uncompress	2.66	2.86	2.93	2.93	2.93	2.93	3.17
espresso	2.77	2.88	2.90	2.90	2.90	2.90	2.97
sc	2.18	2.44	2.61	2.64	2.65	2.66	2.81
xlisp	1.81	1.94	1.97	1.98	1.98	1.99	2.11
fft	1.66	1.87	3.21	3.86	4.09	4.22	4.91
linpacks	1.95	2.84	3.78	3.82	3.82	3.82	3.89
spice	1.50	1.69	1.82	1.84	1.85	1.86	2.13
wave	1.66	1.96	2.27	2.38	2.42	2.45	2.63

Table 5: IPC as memory bandwidth varies

The benchmarks were run on models that varied in the number of paths followed from 1 to 6, to test out this line of reasoning. The net result was that increasing the number of paths followed, and using the strategy of following both paths only if the branch was weakly predicted, did cause a slight increase in performance. The largest increase was 1.3%, in the case of Espresso, with the model that could follow up to three paths.

Because this was a small effect, and since following multiple paths would involve considerable expense to implement in a real machine, the remaining experiments assume that only a single path is followed.

4.3 Varying memory bandwidth

In [FJ94], the performance of a scalar processor was studied with varying organizations of a non-blocking cache. The terminology "fc=1", "fc=2", etc., was defined there to mean that the cache could support one outstanding fetch, two outstanding fetches, etc. This number is a measure of the bandwidth demanded by the primary cache.

In this set of simulations, the processor is run with a limit on the number of outstanding primary cache fetches. When the limit is reached, no more cache requests are allowed until one of the outstanding fetches completes. The processor model in these test runs supports six deep branch prediction, and only follows a single branch path. The results are shown in table 5. For reference, the results of the same processor on a perfect memory system (no latency) and on a blocking cache are shown.

For most benchmarks, two outstanding fetches is sufficient to approach the performance of the unlimited fetch bandwidth case. One exception to this is the FFT benchmark, which shows strong performance gains all the way up to four outstanding fetches.

In comparison with the scalar case, clearly there is an increased demand for bandwidth into the primary cache, even though the latencies are relatively small here. One good example is Compress, which shows an overall performance increase of 20% when going from one to two outstanding fetches. In contrast, in the scalar case (as reported in [FJ94]), there was minimal performance gain

Benchmark	*No lat.*	*16K 4-way 4 cycles*	*8K 4-way 4 cycles*	*8K Direct 4 cycles*	*8K 4-way 8 cycles*
compress	3.31	2.95	2.91	2.89	2.38
uncompress	3.17	3.08	2.93	2.85	2.72
espresso	2.97	2.96	2.90	2.80	2.83
sc	2.81	2.69	2.66	2.60	2.51
xlisp	2.11	2.02	1.99	1.94	1.84
fft	4.91	4.45	4.22	4.08	3.33
linpacks	3.89	3.85	3.82	3.80	3.48
spice	2.13	1.93	1.86	1.83	1.64
wave	2.63	2.56	2.45	2.48	2.22

Table 6: IPC for various cache configurations

in going from one to two outstanding fetches.

By comparing the performance of the blocking model with the performance of the "fc=2" model, it is clear that substantial amounts of memory latency have been hidden by the dynamic scheduling. The biggest gains occur (naturally) on the benchmarks with the highest miss rate. The performance of Compress, for example, increases by 53%. The biggest improvement occurs for FFT, which shows a 126% increase in performance.

4.4 Differing cache designs

To determine how robust the performance of this model is in different environments, the same processor model was run with a variety of different cache configurations. The results are shown in table 6. The cache configurations tested were a direct mapped 8K cache (8K Direct), a 16K 4-way set associative cache (16K 4-way), and an 8K 4-way set associative cache with double the miss penalty. The miss penalties in cycles are given beneath each cache type. The figures in the first column are the results for the 8K 4-way set associative cache used previously.

One question that is sometimes asked is whether the performance decreases linearly with increasing cache miss penalty. Looking at the results for no latency, and the two 8K 4-way caches with miss penalties of 4 and 8 cycles, the performance decline is roughly linear for most of the benchmarks (see figure 7). The benchmarks where the decline is clearly not linear are FFT, Linpacks, and Compress. The drop off in performance suggests that the ability of the processor model to tolerate cache misses has been saturated in some way by these benchmarks.

When the miss penalty increases, this causes instructions to reside longer in the instruction window, waiting on data. Thus the pressure on the instruction window increases as the miss penalty increases. In table 7, we see that the same benchmarks (FFT, Linpacks, and Compress) have the biggest decrease in fetch availability as the miss penalty increases.

The variation in performance for different cache sizes and configurations is mostly explained by the variation in miss rates for the different caches. The miss ratios for the different benchmarks are

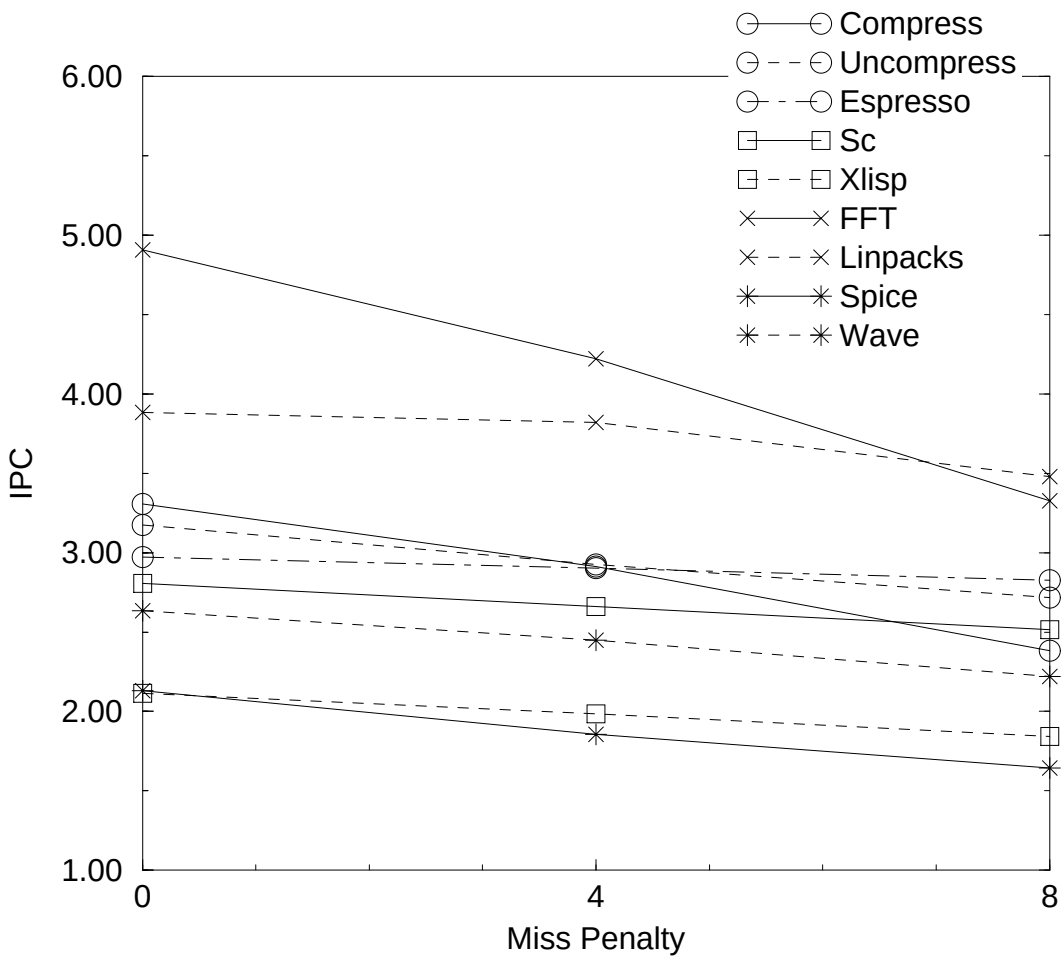

Figure 7: Performance vs. Miss penalty

Benchmark	*No latency*	*4 cycles*	*8 cycles*
compress	89.6%	85.2%	72.0%
uncompress	92.2%	85.8%	80.3%
espresso	94.2%	93.6%	92.7%
sc	94.6%	94.4%	94.0%
xlisp	99.0%	98.4%	97.0%
fft	80.8%	68.7%	54.2%
linpacks	67.0%	65.8%	59.3%
spice	73.3%	69.2%	65.3%
wave	75.3%	74.7%	73.2%

Table 7: Fetch availability vs. Miss penalty

Benchmark	*8K Direct*	*8K 4-way*	*16K 4-way*
compress	10.2%	10.0%	8.4%
uncompress	4.5%	3.4%	1.4%
espresso	3.2%	1.3%	0.3%
sc	5.9%	4.4%	3.0%
xlisp	3.2%	2.1%	1.4%
fft	14.9%	16.2%	6.5%
linpacks	7.7%	7.2%	4.8%
spice	11.6%	10.2%	7.6%
wave	10.3%	9.6%	5.2%

Table 8: Miss ratios

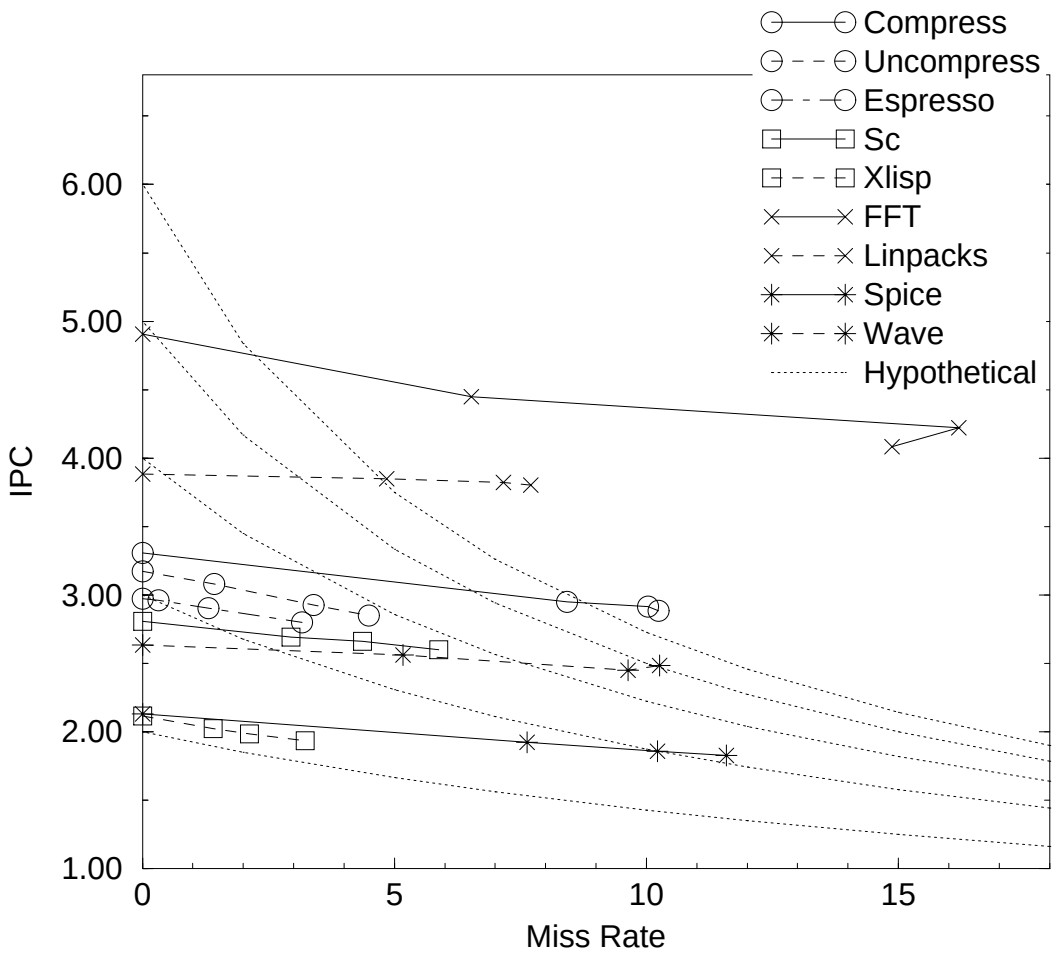

Figure 8: Performance vs. Miss rate

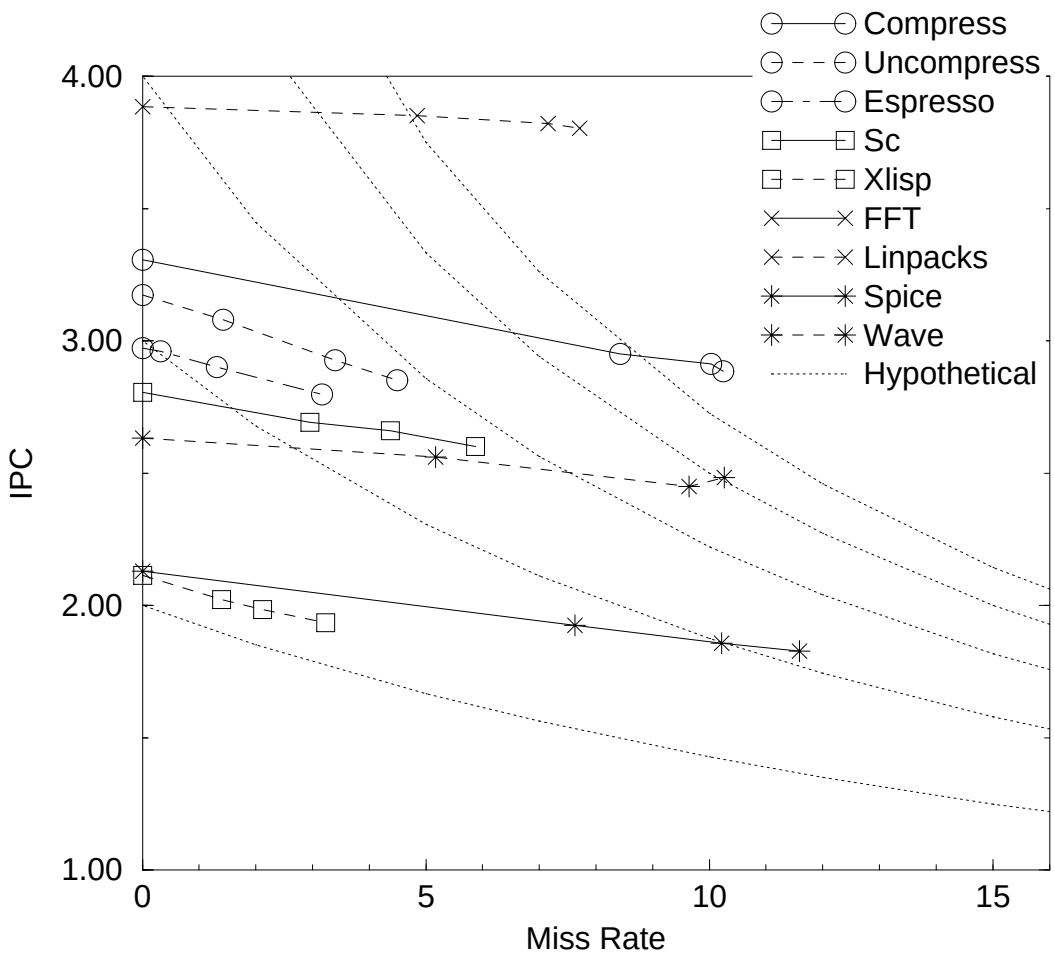

Figure 9: Performance vs. Miss rate

given in table 8, and figure 8 plots performance vs. miss rate for each of the benchmarks.

The dotted lines in figures 8 and 9, labeled "Hypothetical", represent the performance that would be seen if the processor were to block on a cache miss. These curves were calculated assuming that one-third of all operations were loads and stores, and that one-half of the cache misses were dirty misses. These were taken from the observed ratios in the benchmarks.

The amount of latency being tolerated by the benchmark through the use of dynamic scheduling is shown by these lines. At the Y axis, where the miss rate is 0, choose a nearby hypothetical line for each benchmark. Then follow the two lines to the left. As the gap between these two lines grows, the benchmark is tolerating increasing amounts of memory latency.

The direct relationship between performance and miss rate for most benchmarks is clear from the graph. The FFT benchmark once again reveals some unusual behaviour. Although overall miss rate is smaller for the FFT in the direct mapped cache, more of the misses are dirty misses, so the total miss penalty is still greater than in the set associative case.

A close up of the graph, excluding FFT, is shown in figure 9. The leftmost point of each line segment is the perfect memory case (no misses). Moving to the right along the line segment, the data points are for a 16K 4-way set associative cache, an 8K 4-way set associative cache, and an 8K direct mapped cache.

In this graph, the lines are mostly straight, which means that the variation in performance is almost entirely explained by the miss rate. That is, the organization of the cache (direct mapped vs. set associative) does not appear to impact the performance of the dynamically scheduled processor, other than through the variation in miss rate.

5 Discussion

5.1 Related work

Other papers have studied the ability of dynamically scheduled machines to tolerate memory latency[CCMH91, BP91, GGH92]. The focus of [GGH92] was scientific applications running on multiprocessors with very long latencies. Static vs. dynamic scheduling is compared in [CCMH91] and the variation in performance with cache size and load latency are studied in [BP91].

While [BP91] and this paper both study scalar processors running a mix of applications, this paper examines the factors that allow dynamically scheduled machines to tolerate latency. Also [BP91] studied processors with a smaller instruction window and longer memory latencies. As a result, their instruction windows were more frequently saturated, leading to performance that was strongly influenced by memory latency.

5.2 Branch prediction

Many branch prediction methods have been proposed that are better than the simple scheme used here, for example [YP93]. The performance factors introduced in this paper can be used to analyze the likely impact of adopting better branch prediction mechanisms. The factor primarily affected by improved branch predition is fetch utility, so the biggest performance benefits will occur in those benchmarks where fetch utility is a significant factor. Xlisp, for example, should get a significant boost from better branch prediction.

However, fetch utility is not the most important factor limiting performance in most of the benchmarks. The fetch efficiency has the biggest impact on performance in general. A second consideration is whether the instruction window is becoming saturated, as indicated by a declining availability factor. When this happens, improving the fetch utility may not make much difference in performance, as it would increase the pressure on the instruction window. This implies that a simple branch prediction scheme, such as the one presented here, might be sufficient to achieve most of the attainable performance.

5.3 Compiling for dynamically scheduled machines

What sort of compiler optimizations are appropriate for dynamically scheduled machines? First, reducing the dynamic branch frequency is important to achieving good fetch efficiency. Techniques such as loop unrolling and superblock formation[HMB+93], for example, ought to work well.

Second, the fetch availability is determined by the rate at which the processor can execute the instructions in the instruction window. This, in turn, depends on the amount of parallelism that is present in the instruction window. Compiler optimizations that can increase the amount of parallelism available within such a fixed window size, such as loop transformations[WL91], should work well.

6 Conclusion

Dynamic scheduling approaches date back to the CDC 6600[Tho64] and the IBM 360/91[Tom67]. Various techniques for dynamic scheduling have been proposed in [WS84, PHS85, AKT86, Soh90, DT92]. Dynamic scheduling has even been proposed for VLIW processors[Rau93].

This paper inquires into the factors that affect the performance of such a machine. The factors of fetch availability, efficiency, and utility were introduced in order to provide some insight into the causes of variation in performance in a dynamically scheduled processor.

Depth of branch prediction has a significant impact on performance, as it directly improves the fetch efficiency. As fetch efficiency increases, the factors of availability and utility start to play a significant role in limiting performance. Executing along both paths of a branch provides only a small performance increase for machines of this class.

Dynamically scheduled machines are able to tolerate cache misses, but in order to do so they need additional bandwidth at the primary cache interface, in the form of support for multiple outstanding cache misses. For the latencies studied here, supporting two outstanding cache misses is quite effective for most of the benchmarks. However the FFT benchmark continues to improve significantly in performance as the number of outstanding cache misses supported is increased to four.

As the latency is increased from four cycles to eight cycles, the instruction window starts to saturate for some of the benchmarks. This leads to a sharper drop off in performance, as the full impact of the memory latency starts to make itself felt.

Finally, a dynamically scheduled machine can work with a direct mapped primary cache just as well as with a set-associative primary cache. The performance of the processor with different types of primary caches depends primarily on the miss rate, rather than on the cache organization.

In the future we plan to look at how the performance factors are affected by a more deeply pipelined processor model, in which cache accesses require two cycles instead of one. We also plan to look into the tradeoff between instruction window size and overall performance.

References

[AKT86] R. D. Acosta, J. Kjelstrup, and H. C. Torng. An instruction issuing approach to enhancing performance in multiple functional unit processors. *IEEE Transactions on Computers*, 35:815–828, September 1986.

[BF94] J. E. Bennett and M. Flynn. Two case studies in latency tolerant architectures. Technical Report CSL-TR-94-639, Stanford University, Computer Systems Laboratory, October 1994.

[BP91] M. Butler and Y. Patt. The effect of real data cache behavior on the performance of a microarchitecture that supports dynamic scheduling. In *Proc. of the 24th International Symposium on Microarchitecture*, pages 34–41, November 1991.

[CCMH91] P. Chang, W. Chen, S. Mahlke, and W. Hwu. Comparint static and dynamic code scheduling for multiple-instruction-issue processors. In *Proc. of the 24th International Symposium on Microarchitecture*, pages 25–33, November 1991.

[DT92] H. Dwyer and H. C. Torng. An out-of-order superscalar processor with speculative execution and fast, precise interrupts. In *Proc. of the 25th International Symposium on Microarchitecture*, pages 272–81, December 1992.

[FJ94] K. I. Farkas and N. P. Jouppi. Complexity/performance tradeoffs with non-blocking loads. In *Proc. of the 21st International Symposium on Computer Architecture*, pages 211–22, April 1994.

[GGH92] K. Gharachorloo, A. Gupta, and J. Hennessy. Hiding memory latency using dynamic scheduling in shared-memory multiprocessors. In *19th International Symposium on Computer Architecture*, pages 22–33, May 1992.

[HMB+93] R. Hank, S. Mahlke, R. Bringmann, J. Gyllenhaal, and W. Hwu. Superblock formation using static program analysis. In *Proc. of the 26th International Symposium on Microarchitecture*, pages 247–55, December 1993.

[HP90] J. Hennessy and D. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers, Inc., Palo Alto, CA, 1990.

[KH92] Gerry Kane and Joe Heinrich. *MIPS RISC Architecture*. Prentice-Hall, Inc., Englewood Cliffs, NJ, 1992.

[PHS85] Y. N. Patt, W. Hwu, and M. Shebanow. HPS, a new microarchitecture: Rationale and introduction. In *Proc. of the 18th Annual Workshop on Microprogramming*, pages 103–108, December 1985.

[Rau93] B. R. Rau. Dynamically scheduled VLIW processors. In *Proc. of the 26th International Symposium on Microarchitecture*, pages 80–92, December 1993.

[Smi81] J. E. Smith. A study of branch prediction strategies. In *Proc. Eighth Symposium on Computer Architecture*, pages 135–48, May 1981.

[Smi84] J. E. Smith. Decoupled access/execute computer architectures. *ACM Transactions on Computer Systems*, 2:289–308, November 1984.

[Soh90] G. S. Sohi. Instruction issue logic for high-performance, interruptible, multiple functional unit, pipelined computers. *IEEE Transactions on Computers*, 39:349–359, March 1990.

[Tho64] J. E. Thornton. Parallel operation in Control Data 6600. In *Proc. AFIPS Fall Joint Computer Conference*, number 26, part 2, pages 33–40, 1964.

[Tom67] R.M. Tomasulo. An efficient algorithm for exploiting multiple arithmetic units. *IBM Journal of Research and Development*, 11:25–33, 1967.

[Uht93] A. K. Uht. Extraction of massive instruction level parallelism. *Computer Architecture News*, 21:5–12, June 1993.

[WL91] M. Wolf and M. Lam. A loop transformation theory and an algorithm to maximize parallelism. *IEEE Transactions on Parallel and Distributed Systems*, 2:452–471, October 1991.

[WS84] S. Weiss and J. E. Smith. Instruction issue logic in pipelined supercomputers. *IEEE Transactions on Computers*, 33:1013–1022, November 1984.

[YP93] T-Y. Yeh and Y. Patt. A comparison of dynamic branch predictors that use two levels of branch history. In *Proc. of the 20th International Symposium on Computer Architecture*, pages 257–66, May 1993.