Instruction Cache Fetch Policies for Speculative Execution

Dennis Lee and Jean-Loup Baer

Department of Computer Science
and Engineering, Box 352350
University of Washington
Seattle, WA 98195-2350
{dlee,baer}@cs.washington.edu

Brad Calder and Dirk Grunwald

Department of Computer Science
Campus Box 430
University of Colorado
Boulder, CO 80309
{calder,grunwald}@cs.colorado.edu

Abstract

Current trends in processor design are pointing to deeper and wider pipelines and superscalar architectures. The efficient use of these resources requires *speculative execution*, a technique whereby the processor continues executing the predicted path of a branch before the branch condition is resolved.

In this paper, we investigate the implications of speculative execution on instruction cache performance. We explore policies for managing instruction cache misses ranging from aggressive policies (always fetch on the speculative path) to conservative ones (wait until branches are resolved). We test these policies and their interaction with *next-line* prefetching by simulating the effects on instruction caches with varying architectural parameters. Our results suggest that an aggressive policy combined with next-line prefetching is best for small latencies while more conservative policies are preferable for large latencies.

1 Introduction

To keep up with ever shorter clock cycles and to exploit instruction level parallelism, next generation processors will support deeper and wider pipelines and out of order execution engines. This will require that processors execute more than one basic block at a time to keep the pipeline and the execution units busy. Hence many instructions will be issued before a conditional branch instruction can be resolved. This is called *speculative execution*.

To efficiently handle speculative execution, all modern processors include some form of branch prediction ranging from simple static schemes to very efficient dynamic branch architectures. The path that the processor takes upon reaching a conditional statement, as well as the time at which the processor identifies the statement as a branch, is dependent on this underlying branch architecture. A branch *misfetch* occurs when there is a delay in identifying an instruction as a branch or when a correctly predicted branch has to wait for its target address to be calculated. A branch *mispredict* occurs when an incorrect target address is predicted for a branch. On a branch misfetch or mispredict, the processor will start fetching instructions along the wrong execution path. If an instruction cache (I-cache) miss is then encountered, two detrimental effects might arise from fetching the missing line: (i) the line on the wrong path may displace useful instructions in the instruction cache, and (ii) the channel (bus) between the I-cache and the next level of the memory hierarchy might be busy while an I-cache miss on the correct path needs to be processed. On the other hand, if the instructions fetched on the wrong path will be used sooner than the displaced instructions, a prefetch of required instructions has been performed.

Note the distinction between wrong path and speculative path: a path is speculative whether or not it is on the wrong path, as long as a conditional branch upon which the execution depends has not been resolved.

A simple method to reduce the I-cache fetch penalty is *next-line* prefetching, whereby the line following the I-cache line currently being accessed is prefetched under certain conditions. Next-line prefetching has been shown to be quite successful for instruction caches but no study has been made of its impact in the context of speculative execution.

In this paper, we investigate methods for dealing with instruction cache misses encountered during speculative execution. We propose several alternatives upon encountering an I-cache miss while on the speculative path. We quantify the effects of handling an I-cache miss down a mispredicted path by classifying those I-cache misses as useful prefetches or as fetches that pollute the I-cache. In both cases, we consider the implications of consuming off-chip bandwidth even after the processor knows that it has gone down the wrong path. We also investigate the combined effects of these methods with a next-line prefetching strategy.

The rest of this paper

In §2, we describe related branch prediction, speculative execution, and prefetching studies. In §3, we describe the policies we investigated for handling I-cache misses during speculative execution. We use trace-driven simulation to compare the performance of the different policies. Section 4 describes the programs we traced and the baseline architecture. In §5, we compare the performance of the different policies with varying architectural parameters. Finally, we conclude in §6.

2 Prior and Related Work

This section describes prior work in branch prediction, instruction prefetching, and speculative execution.

2.1 Branch Prediction

Branch Target Buffers (BTB) have been used as a mechanism for branch and instruction fetch prediction, effectively predicting the prior behavior of a branch [Bray & Flynn 91, Lee & Smith 84, Smith 81, McFarling & Hennessy 86, Perleberg & Smith 93, Yeh & Patt 92b].

Traditionally, a BTB is organized as a cache with each entry consisting of the branch instruction address, a field used for prediction, and the target address of the branch. On a BTB hit and a prediction "taken" ("not taken"), the instruction at the target (fall-through) address is fetched. The Intel Pentium is an example of a modern architecture using a BTB – it has a 256-entry BTB organized as a four-way associative cache. Only branches that are "taken" are entered into the BTB. For each BTB entry, the Pentium uses a two-bit saturating counter to predict the direction of a conditional branch. In this BTB architecture, the branch prediction information is associated or *coupled* with the BTB entry. This implies that the direction of a conditional branch can only be predicted dynamically if the conditional branch address is found in the BTB. On a BTB miss, the branch must be predicted using static prediction; in the case of the Pentium the fall-through path is assumed.

An alternative BTB architecture is the *decoupled* design, where the branch prediction information is not associated with the BTB and is used for all conditional branches, including those not recorded in the BTB. [Calder & Grunwald 94] found that decoupled designs performed better than coupled designs. This allows conditional branches that do not hit in the BTB to use dynamic prediction. The PowerPC 604 is an example of an architecture using a decoupled design. It has a 64-entry fully associative cache that holds the target address of the branches most recently taken, and it uses a separate 512-entry pattern history table (PHT), indexed by the lower nine bits of the branch address to predict the direction for conditional branches.

There are several different PHT variations. [Pan et al. 92] and [Yeh & Patt 92a, Yeh & Patt 93] investigated *branch-correlation* or *two-level* branch prediction mechanisms. Although there are a number of variants, these mechanisms generally combine the history of several recent branches to predict the outcome of a branch. The simplest example is the *degenerate method*. When using a 2^k entry table, the processor maintains a k-bit shift register (the global history register) that records the outcome of previous branches (a taken branch is encoded as a 1, and a not-taken branch as a 0). The shift register is used as an index into the PHT, much as the program counter is used for a direct-mapped PHT. This provides contextual information and correlation about particular patterns of branches. Recently, [McFarling 93] showed that combining branch history with the branch's address was more effective. His method used the exclusive-or of the global history register and the branch address as the index into the PHT.

2.2 Instruction cache prefetching

The next-line prefetching policy was introduced by [Smith 82] for unified caches. Upon referencing line i, line $i+1$ can be prefetched. Possible options are: prefetch line $i+1$ unconditionally, prefetch only on a miss to line i, or prefetch line $i+1$ only if line i is referenced for the first time in the cache (a one-bit encoding is required). An extension to next-line prefetching where several consecutive data streams are prefetched in FIFO *stream buffers* has been proposed by [Jouppi 90]. It was found that 85% of the misses of a 4KB I-cache could be removed by a stream buffer with four 8-byte entries.

[Smith & W.-C.Hsu 92] studied next-line prefetching for instruction caches in machines with high bandwidth and large cache lines (e.g., 16 to 128 words). They found that the *fetchahead* distance, that is the number of instructions remaining to be issued in a line before the next line has to be fetched, is a critical parameter because of the large lines. They also examined *target* prefetching, which uses a table of target addresses for prefetching taken branches. Their results show that next-line prefetching performed slightly better than target prefetching, and when these two techniques were combined, the original miss rate was reduced by a factor of 2 to 3.

[Pierce & Mudge 94] investigated a prefetching algorithm where both paths of a conditional branch are prefetched. They called this scheme *wrong-path* prefetching because no attempt is made to prefetch from the predicted path. Their approach combines next-line prefetching and target prefetching. The next-line prefetching prefetches the fall-through path of the conditional branch while target prefetching prefetches the taken path. Unlike [Smith & W.-C.Hsu 92], the target address of the conditional branch is computed in the decode stage instead of using a table of target addresses. All prefetch line addresses are put into a prefetch queue and are processed in a priority order. An I-cache miss takes precedence over any prefetch, and target prefetches take precedence over next-line prefetches. They found that next-line prefetching accounts for 70 to 80% of the gain in performance with this scheme, and target prefetching accounts for the rest [Pierce 95].

Neither [Smith & W.-C.Hsu 92] nor [Pierce & Mudge 94] examined the effects of handling cache misses during speculative execution and its interaction with prefetching.

3 Instruction Cache Policies and Speculative Execution

In contrast with the studies mentioned above, we investigate the effect of speculative execution and instruction prefetching on a blocking I-cache and on a very simple non-blocking I-cache. The policies we investigate only require minor modifications to the design of I-caches on modern processors as opposed to a fully non-blocking pipelined memory system that is only used to facilitate aggressive prefetching. Even with the simple prefetching strategy we propose, the increase in memory bandwidth is significant; being more aggressive will only add more stress to the memory system.

We first describe policies that can be implemented during execution of a speculative path to handle I-cache misses. We then describe the variant of next-line prefetching that we investigate in this paper.

When an instruction cache miss occurs during speculative execution, we have a spectrum of possibilities ranging from the most pessimistic (stall until the branch is resolved) to the most optimistic (fetch the missing line independently of the depth of speculation).

At first glance, it seems ideal to service the I-cache miss only if the program is running on the correct path. This keeps executions of incorrect paths from polluting the I-cache and reduces the amount of memory traffic generated. We call a policy with that knowledge *Oracle* since it knows whether or not it is running along the correct path on an I-cache miss. In practice, it is impossible to implement Oracle because of the very definition of speculative execution. We include it as a yardstick to measure the performance of the different policies.

Policy	Description
Oracle	Only process I-cache misses on the right path.
Optimistic	Process all I-cache misses.
Resume	Like Optimistic, but allow execution to continue along the the correct path even if an I-cache miss is outstanding due to instruction fetches in the wrong path.
Pessimistic	On an I-cache miss, wait until all outstanding branches are resolved and until all previous instructions are decoded, and fetch only if on the correct path.
Decode	On an I-cache miss, wait until all previous instructions are decoded and fetch only if the instruction was not misfetched.

Table 1: *Summary of the instruction cache fetch policies.*

A second policy, which we call *Optimistic*, assumes that the branch prediction accuracy is good, and executing on the wrong path is rare. Moreover, the instructions fetched when executing down the wrong path may be useful later on, effectively performing a prefetch of these instructions. [1] This prefetch effect is the reason why Oracle may not be ideal.

If branches are mispredicted or misfetched a lot, the Optimistic policy will hurt performance because it may stall the processor waiting for an instruction cache miss that is not needed. To alleviate this problem, we introduce the *Resume* policy that allows the processor to keep running even when an instruction cache miss is outstanding from the wrong path. This is a very simple form of a non-blocking I-cache. However, an I-cache miss in the right path still needs to wait for a previously initiated fill from the wrong path to complete.

There is little additional hardware needed to implement the Resume policy. It consists of a buffer that can hold the missing cache line when it is returned from memory as well as the index where it needs to be stored in the I-cache. Storing the line in the cache will take place at the next I-cache miss, without interference with the normal operation of the cache. On the subsequent miss, the index of the missing line and the index in the resume buffer should be checked in case they are the same to avoid an unnecessary memory request.

At the other end of the spectrum, the *Pessimistic* policy avoids accessing memory (and hence servicing the I-cache miss) unless it is sure that the instruction will be used. This prevents cache pollution, because no useful lines are displaced by erroneous fetches, and it keeps the bus free in case it is needed right away (e.g., if there is an immediate cache miss on the correct path). However, if the branch prediction accuracy is high, then waiting until the branch is resolved will likely result in unnecessary stalls. The *Decode* policy attempts to alleviate these stalls by assuming that misfetches occur more often than mispredicts. On an I-cache miss, Decode waits until the previous instructions have been decoded and services the cache miss only if it is not for a misfetched instruction. Hence any pollution and bus blocking effects due to misfetches are avoided.

Table 1 summarizes the different instruction cache fetch policies we consider.

We will also assess next-line prefetching. The policy we assume is "maximal fetchahead and first time referenced." We chose this policy because it can be easily implemented with the modifications to the hardware already required to efficiently recover from wrong path I-cache misses. When a cache line, say line i, is loaded in the instruction cache for the first time, we set a bit to that effect. When an instruction of line i is fetched and the above mentioned bit is set, we initiate the prefetch of line $i+1$ (if it is not already in the cache and if the bus is free). At the same time we reset the bit for line i.

The writing of a prefetched line into the cache is handled as in Resume except that the prefetched line is written before the next prefetch is issued or at the next I-cache miss, whichever comes first. Since most modern processors have an instruction buffer, the I-cache will not be accessed by the fetch unit every cycle and prefetching could be done during the cycles where the I-cache is idle. [2] It is also possible to bank the I-cache into even and odd lines. Hence two tags may be read in at a time, and a read and write may occur simultaneously.

Next-line prefetching is advantageous for long sequential blocks and when branches are correctly predicted as not taken. However, it can hinder speculative execution for correctly predicted taken branches under the Optimistic and Resume policies since it might initiate requests that have to be completed before generating requests for the speculative path.

4 Experimental Methodology

We used trace driven simulation to evaluate the performance of the instruction fetch policies described in the last section. We instrumented programs from the SPEC92 benchmark suite and object-oriented programs written in C++. Table 2 describes the programs we simulated and the inputs used. We used ATOM [Srivastava & Eustace 94] to instrument the programs. Due to the structure of ATOM, we did not need to record traces and could trace very long-running programs.

4.1 Architectures Simulated and Performance Metrics

We simulated a four-way superscalar machine. The branch architecture consists of a decoupled 64-entry 4-way associative branch target buffer (BTB) to predict the target address for taken branches and a 512-entry pattern history table (PHT) for predicting conditional branches. The PHT uses McFarling's technique of XORing the global history register with the branch address to index into a table of saturating 2-bit counters. We assume all conditional branches take four cycles to resolve, and all branches take two cycles to decode. For all conditional and direct branches, it takes two cycles to calculate the branch target address if it is not found in the BTB. Hence for all of the architectures simulated, misfetched branches

[1] Most machines appear to employ the optimistic policy because the fetch unit is not aware that it is going down a speculative path.

[2] The UltraSparc uses a 12 deep instruction buffer to free up its instruction cache for prefetching.

Programs	Description	Inst	% Branches
doduc	Monte Carlo simulation of the time evolution of a thermohydraulical modelization for a nuclear reactor's component. Input was ref.in.	1150	8.5
fpppp	Quantum chemistry benchmark measuring performance of two electron integral derivatives in the GaussianXX series of programs. Input was ref.in.	4330	2.8
su2cor	Quantum physics benchmark where masses of elementary particles are computed in the framework of the Quark-Gluon theory. Input was ref.in.	4780	4.4
ditroff	C version of the "ditroff" text formatter. Input was the collection of manual pages given to Groff.	39	17.5
gcc	GNU C Compiler, version 1.35. The measurements show only the execution of the "cc1" phase of the compiler. Input was the 4832-line 1stmt.i.	144	16.0
li	Lisp interpreter adapted from XLISP 1.6 by David Michael Betz. Input was a solution to the 8-queens problem.	1360	17.7
tex	A widely used text-formatting program, version 3.141. Input was "dvips.tex," a forty-five page manual.	148	10.0
cfront	The AT&T C++ to C conversion program, version 3.0.2. Input was `groff.C`, part of the GNU `troff` implementation. The input was first preprocessed with `cpp`.	16.5	13.4
db++	A version of the "delta-blue" constraint solution system written in C++. We used the example program that comes with the Deltablue system.	87	17.6
groff	Groff Version 1.9 — A version of the "ditroff" text formatter. Input was a collection of manual pages.	57	17.5
idl	Sample backend for the Interface Definition Language system distributed by the Object Management Group. Input was a sample IDL specification for an early release of the Fresco graphics library.	21.1	19.6
lic	Part of the Stanford University Intermediate Format (SUIF) compiler system. It is a **l***inear* **i***nequality* **c***alculator*. Input was the largest distributed example.	6	16.5
porky	Part of the Stanford University Intermediate Format (SUIF) compiler system. It performs a variety of compiler optimizations. We used it to perform constant folding, constant propagation, reduction detection and scalarization for a large C program.	164	19.8

Table 2: *General information about the benchmarks used in this study. Instruction counts are in millions. % Branches gives the percentage of executed instructions that were branches. The programs were compiled on a DEC 3000-400 which uses the Alpha AXP-21064 processor. We used the DEC FORTRAN compiler, the DEC C compiler, and the DEC C++ compiler. The systems were running the standard OSF/1 V1.3 operating systems. All programs were compiled with standard optimization (O).*

	% Cache Miss		PHT Mispredict ISPI		BTB Misfetch ISPI		BTB Mispredict ISPI	
Program	8K	32K	B1	B4	B1	B4	B1	B4
doduc	2.94	0.48	0.22	0.37	0.04	0.04	0.00	0.00
fpppp	7.27	1.08	0.08	0.12	0.01	0.01	0.00	0.00
su2cor	1.33	0.00	0.08	0.10	0.00	0.00	0.00	0.00
ditroff	3.18	0.58	0.44	0.64	0.22	0.22	0.00	0.00
gcc	4.48	1.71	0.53	0.63	0.28	0.28	0.05	0.05
li	3.33	0.06	0.35	0.54	0.24	0.24	0.04	0.04
tex	2.85	1.00	0.27	0.36	0.11	0.11	0.03	0.03
cfront	7.24	2.63	0.50	0.56	0.34	0.34	0.05	0.05
db++	1.57	0.42	0.16	0.41	0.13	0.13	0.01	0.01
groff	5.33	1.68	0.42	0.57	0.39	0.38	0.06	0.06
idl	2.17	0.67	0.30	0.49	0.10	0.11	0.04	0.05
lic	3.93	1.68	0.45	0.56	0.27	0.27	0.00	0.00
porky	2.51	0.66	0.42	0.48	0.20	0.20	0.04	0.04
Average	3.70	0.97	0.32	0.45	0.18	0.18	0.03	0.03

Table 3: *Instruction cache and branch prediction characteristics. Miss rates are given for direct mapped 8K and 32K instruction caches. The PHT and BTB results are in terms of instruction issue slots lost per correct path instruction. The PHT and BTB are shown for one (B1) and four (B4) unresolved branches.*

have a two cycle (eight instruction issue slots) penalty and mispredicted branches have a four cycle (sixteen instruction issue slots) penalty.

We varied the number of unresolved branches allowed at a time between 1, 2, and 4. We simulated both 8K and 32K direct mapped caches, and we examined the effect of a low (5 cycle) cache miss penalty and a high (20 cycle) penalty. In this paper, we will only report the most interesting of these data in the interest of brevity.

Our primary metric is *instructions slots lost per instruction* (ISPI). This metric measures the number of lost instruction issue slots due to stalls created by misfetched instructions, mispredicted branches, and instruction cache misses. By assuming perfect pipelining, no data cache misses, and no issue slots lost to misaligned branches and targets, this metric effectively gives the maximum instruction issue rate possible for the configurations we considered. With out-of-order execution engines and non-blocking data caches, the instruction fetch architecture needs to issue as many useful instructions as possible to keep the execution units busy. We note that it is possible that the increased memory bandwidth requirements of some of the fetch policies may have a net negative impact due to contention with the data cache miss requests. Hence we also report the increase in memory bandwidth required by the more aggressive policies.

Table 3 shows the miss rate for direct mapped 8K and 32K caches for the programs we simulated. The programs we simulated had a non-trivial miss rate for an 8K cache (on average 3.7%). The table also shows the ISPI caused by mispredicted and misfetched branches due to the PHT and BTB.

The BTB architecture used in each simulation updated the BTB speculatively. After branch instructions were decoded, predicted taken branches had their target address inserted into the BTB. The table shows that speculatively updating the BTB, even to a depth of four unresolved branches, has little effect on the performance of the BTB compared to only processing one unresolved branch.

We modeled a simple PHT architecture that waits until a branch is resolved before updating the global history register and the 2-bit counter. This architecture tries to avoid conflicts in the PHT during speculative execution by XORing the global history register with the branch address. Table 3 shows that the performance of the PHT decreases (i.e., the PHT ISPI increases) with deeper speculation.

5 Results

In this section, we present the results of our simulations. We first present the results of the baseline architecture with the branch architecture described in Section 4.1: four instructions issued per cycle, a speculative depth of 4 unresolved branches, an 8K direct-mapped I-cache (32 bytes lines), and a small I-cache miss penalty (5 cycles). Next we vary the parameters, looking at longer miss penalties, deeper speculation, and larger caches. Finally, we consider the effects of next-line prefetching.

5.1 Baseline results

5.1.1 Miss ratios

Although miss ratios are not the primary metric when looking at speculative execution, they show the pollution and prefetching effects that result from the execution of the speculative path. We have recorded the miss ratios of the Oracle and Optimistic policies[3] and partitioned them as follows:

Program	BM	SPo	SPr	WP	TR
doduc	2.58	0.10	0.36	0.58	1.11
fpppp	7.18	0.03	0.08	0.15	1.01
su2cor	1.24	0.01	0.09	0.10	1.01
ditroff	2.27	0.38	0.92	2.01	1.46
gcc	3.09	0.48	1.40	3.25	1.52
li	2.43	0.42	0.90	2.05	1.47
tex	2.36	0.25	0.49	1.24	1.35
cfront	5.22	0.63	2.02	4.67	1.45
db++	1.15	0.23	0.42	1.02	1.52
groff	3.72	0.70	1.61	3.95	1.57
idl	1.67	0.14	0.49	1.03	1.31
lic	2.56	0.36	1.37	2.62	1.41
porky	1.81	0.35	0.70	1.67	1.53
Average	2.87	0.32	0.83	1.87	1.36

Table 4: *Categorization of miss ratios. Legend: BM - Both Miss, SPo - Spec Pollute, SPr - Spec Prefetch, WP - Wrong Path, and TR - Traffic Ratio. Traffic ratio is the ratio of the number of misses with Optimistic to that of Oracle.*

- Misses that occur in both Oracle and Optimistic policies (Both Miss).
- Misses that occur only in Optimistic on the correct speculative path (Spec Pollute). These misses are the result of the pollution caused by instruction fetches on wrong paths.
- Misses that occur only in Oracle (Spec Prefetch). These misses are prevented in the Optimistic case because of the prefetching effect of wrong path execution.
- Misses that occur only in Optimistic on the wrong path (Wrong Path). The main cost of these misses is increased memory bandwidth.

Table 4 shows that the effect of prefetching (Spec Prefetch) is more beneficial than the pollution effect (Spec Pollute). In the case of the Fortran programs, both effects are minimal. For the C and C++ programs, the number of misses due to pollution (Spec Pollute), when using Optimistic fetching, is approximately half the number of misses prevented by correct prefetching (Spec Prefetch) down the incorrect path. The salient feature is that the number of misses that occur on the wrong path in speculative execution (Wrong Path) is quite high, yielding an overall miss ratio for Optimistic (Both Miss + Spec Pollute + Wrong Path) that can be up to 57% higher (e.g., $groff$) than that of Oracle (Both Miss + Spec Prefetch). The last three columns of the table show the increase in memory bandwidth required by Optimistic and Resume over Oracle and Pessimistic.

Recall that miss ratios on the wrong path are not a serious impediment if they do not increase the miss ratio on the right path (i.e., if Spec Pollute is small) and if they do not stall the pipeline much longer than it takes to resolve a branch condition. The latter drawback will be greatly reduced when the latency to the next level is small (e.g., on the order of a mispredict penalty) and can

[3] Pessimistic and Oracle generate the same number of I-cache misses. Optimistic and Resume generate the same number of I-cache misses.

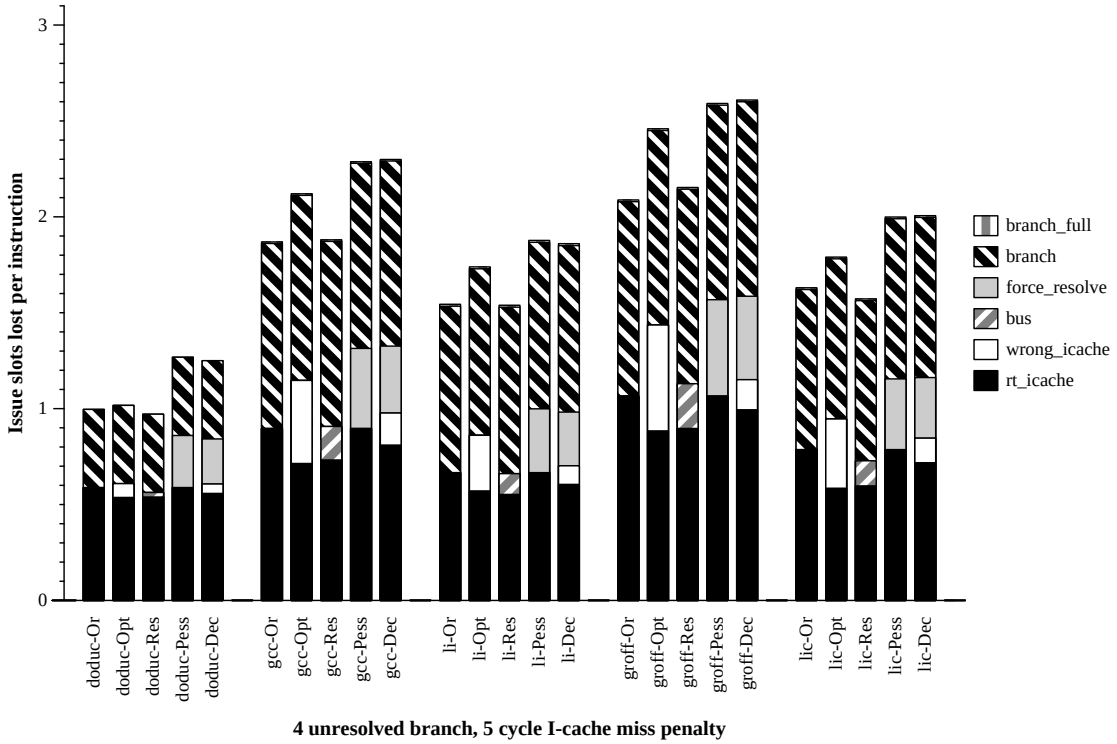

Figure 1: *Breakdown of the penalty components for the base architecture. The height of each bar shows the total ISPI penalty for each policy and benchmark (the lower the height, the better the performance). The first two components (branch_full and branch) are unaffected by the different instruction miss policies.*

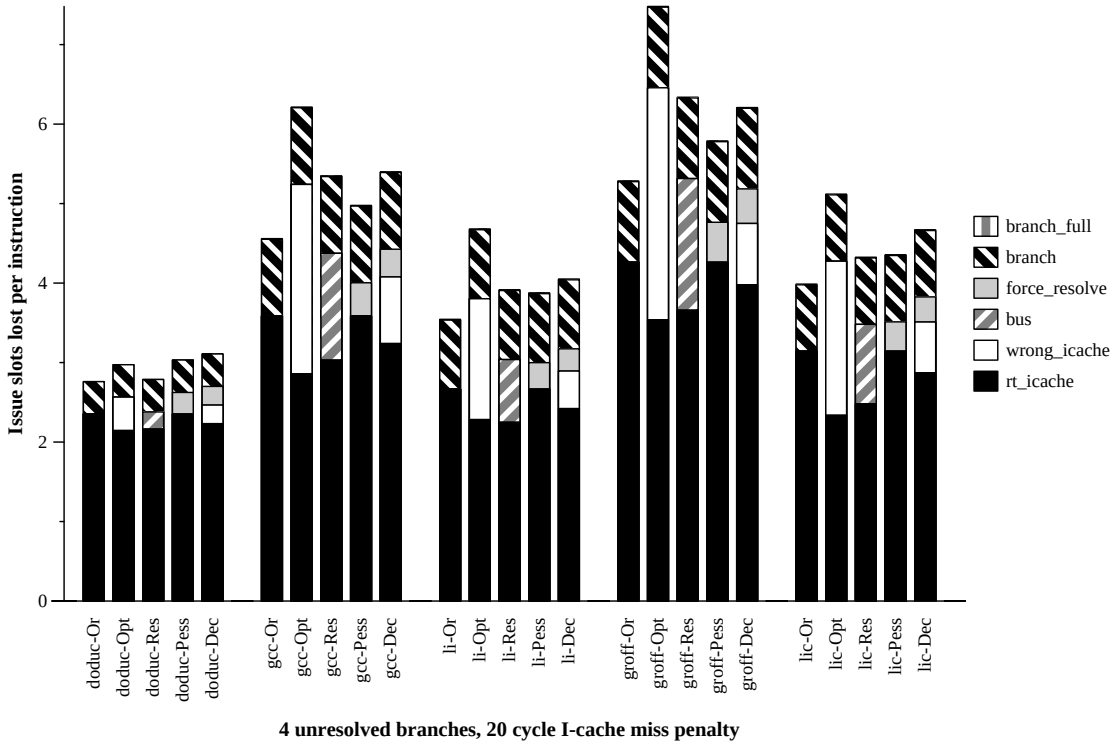

Figure 2: *Effect of long miss latency. This figure shows the ISPI penalty for a machine with long I-cache miss penalty, as well as how the increased latency affects the various ISPI components. Note the different scales between this figure and Figure 1.*

be further alleviated by the Resume policy. On the other hand, we should expect to see Optimistic perform worse with increased memory latency since there is a good chance that the Wrong Path misses will delay some I-cache misses on the right path.

5.1.2 Policies

A detailed breakdown of the components contributing to the penalty ISPI for five of the benchmarks is shown in Figure 1. These five applications, Fortran program *doduc*, the C programs *gcc* and *li*, and the C++ programs *groff* and *lic*, are representative of the results we saw for the all the benchmarks.

We first present some general observations for the baseline architecture:

- Optimistic is always better than Pessimistic.
  The average penalty of Optimistic is about 12% lower than that of Pessimistic with a slighter higher advantage for the Fortran programs and a slightly smaller one for the C++ programs.
- Resume performs the best, and does as well as Oracle.
  For the C and C++ programs, Resume yields a 6% improvement over Optimistic. This modest improvement justifies the small added hardware complexity of implementing Resume.
- Decode shows almost no difference in ISPI from Pessimistic.

We can understand these results better by looking at the different components that compose the overall ISPI penalty.

- *rt_icache* and *wrong_icache* are the ISPI's due to waiting for the instruction cache to refill on the right and wrong paths respectively. The I-cache misses on the wrong path are not as expensive as misses along the right path because some of the latency is hidden by the processor resolving outstanding branches.
- *bus* is the penalty from waiting for the bus when a previously outstanding cache miss has not completed yet. This happens with Resume when a miss from the wrong path has not yet completed and an I-cache miss is outstanding in the right path. It is zero for the other policies because Resume is the only policy that allows the processor to continue while an I-cache miss is being serviced.
- *force_resolve* is the penalty along the correct path of waiting for a branch to resolve before fetching a line that missed in the cache. It is zero for Optimistic and Resume because these policies do not wait for branches to be resolved before servicing a miss.
- *branch* is the cost of misfetched and mispredicted branches.
- *branch_full* reflects the number of issue slots lost as the machine waits for a previous branch to resolve because of the limited number of outstanding unresolved branches that can be handled by the machine.

branch_full is zero in the graphs because allowing four outstanding branches is sufficient to hide the latency of branch resolution. For Optimistic, *wrong_icache* reflects the cost of servicing misses on the wrong path. For Resume, *bus* reflects this cost. The difference between *bus* in the Resume policy and *wrong_icache* in the Optimistic policy reflects the improvement from resuming execution along the correct path immediately upon finding out about the mispredict/misfetch.

doduc is typical of loop intensive programs in that the effect of speculative execution on the instruction cache is minimal. These programs tend to have only a few loop branches that account for the majority of branches executed in the program. Notice the small *wrong_icache* component even with Optimistic. This is due to a rather small number of branches in these programs and a high prediction accuracy. Hence the effects of branches are limited. In these types of programs, Pessimistic and Decode are penalized because wrong path misses rarely happen.

Looking at the other programs, we see that Pessimistic and Decode effectively place a tax on I-cache misses (i.e., *force_resolve*). They force instruction cache misses to wait either for the previous instruction to be decoded (to guard against a misfetch) or, in the case of Pessimistic, for all previous branches to be resolved (to guard against a mispredict). The figure shows that the cost of the extra I-cache misses while executing the wrong path is less than (for Resume) or equal to (for Optimistic) that of this tax on the I-cache miss time (for Pessimistic and Decode). Optimistic is better than Pessimistic and Decode because of the prefetching effect of the wrong path misses (i.e., the *rt_icache* component of Optimistic is lower). The cost of wrong path misses is small because the I-cache penalty is small and some of the miss latency on the wrong path is covered up by the processor resolving the branch.

The *force_resolve* components of the Pessimistic and Decode are almost equal. This indicates that most of the waiting time for these policies is due to waiting for the previous branch instruction to get decoded. The occasional trip down the mispredicted path slightly improves performance for Decode over Pessimistic in some applications (*doduc, fppp, su2cor, li, tex,* and *idl*) because of the prefetch effects which decreases *rt_icache* wait times. However, for other applications (*gcc,cfront,db++,groff, lic,* and *porky*), *wrong_icache* increases ISPI more than the decrease in *rt_icache*.

Fetching instructions into the I-cache on the wrong path has two effects: prefetch and pollution. The interaction of these effects can be seen in the *rt_icache* components of the different policies. If this component becomes greater than *rt_icache* in Oracle, then the pollution effect dominates. If less then prefetch dominates.

Comparing the *rt_icache* components of Oracle to Optimistic and Resume policies shows that the prefetch effect is significant and dominates the pollution effect as mentioned before. This is the case for all the benchmarks (c.f., Table 4). However, the cost of the prefetch in Optimistic (i.e., waiting for the cache miss to complete in the wrong path, *wrong_icache*) is more than the improvement in instruction cache miss rate on the correct path. The penalties are not as bad as predicted by the I-cache miss numbers. For example, *groff* doesn't have a 57% increase in the ISPI as indicated by just looking at the instruction cache miss rate, rather some of the miss latency is hidden. By making the wrong path misses even cheaper, Resume brings performance closer to or better than that of Oracle.

In summary, when the cost of an instruction cache miss is relatively small, the Resume policy should be adopted. Compared to Pessimistic it takes advantage of branch prediction accuracy; compared to Optimistic it reduces the cost of waiting for unneeded cache misses in progress.

	1 Unresolved Branch					2 Unresolved Branches					4 Unresolved Branches				
Program	Oracle	Opt	Res	Pess	Dec	Oracle	Opt	Res	Pess	Dec	Oracle	Opt	Res	Pess	Dec
doduc	1.19	1.20	1.17	1.46	1.43	1.10	1.12	1.08	1.37	1.35	1.00	1.02	0.97	1.27	1.25
fpppp	1.64	1.64	1.64	2.24	2.22	1.59	1.60	1.59	2.19	2.18	1.58	1.59	1.58	2.18	2.17
su2cor	0.46	0.45	0.45	0.58	0.56	0.40	0.39	0.38	0.52	0.49	0.37	0.36	0.36	0.50	0.47
ditroff	2.02	2.09	2.01	2.35	2.29	1.68	1.80	1.67	2.01	1.96	1.52	1.68	1.52	1.84	1.84
gcc	2.33	2.46	2.34	2.73	2.71	1.99	2.19	2.01	2.40	2.39	1.87	2.11	1.88	2.28	2.30
li	2.04	2.10	2.01	2.35	2.31	1.65	1.72	1.62	1.98	1.91	1.54	1.73	1.54	1.88	1.86
tex	1.28	1.34	1.28	1.55	1.52	1.11	1.19	1.12	1.38	1.36	1.07	1.18	1.07	1.34	1.33
cfront	2.68	2.88	2.69	3.32	3.30	2.45	2.73	2.46	3.09	3.10	2.40	2.73	2.41	3.06	3.09
db++	1.43	1.50	1.46	1.58	1.56	1.00	1.09	1.03	1.15	1.15	0.87	0.98	0.90	1.02	1.09
groff	2.53	2.75	2.59	3.02	2.99	2.18	2.47	2.24	2.67	2.66	2.09	2.43	2.15	2.58	2.60
idl	1.74	1.79	1.74	1.94	1.93	1.30	1.35	1.29	1.51	1.49	1.09	1.15	1.07	1.30	1.28
lic	2.13	2.22	2.10	2.48	2.46	1.77	1.89	1.72	2.13	2.11	1.63	1.78	1.57	2.00	2.01
porky	2.00	2.11	2.02	2.24	2.23	1.49	1.61	1.50	1.74	1.72	1.25	1.40	1.26	1.50	1.51
Average	1.80	1.89	1.81	2.14	2.12	1.52	1.63	1.52	1.86	1.84	1.41	1.55	1.41	1.75	1.75

Table 5: *Effect of speculation depth. This table shows the ISPI penalty for a direct mapped 8K cache with a miss penalty of 5 cycles, when up to one, two, and four unresolved branches are allowed.*

5.2 Variations on the base architecture

5.2.1 Influence of the miss penalty

Figure 2 gives the breakdown of the components that contribute to the ISPI like Figure 1 but for a machine that has a 20 cycle I-cache miss penalty. When the miss penalty is large compared to the mispredict and misfetch penalties, predicting on the wrong path plays a significant effect. When predictions are very accurate, as in the Fortran programs, Optimistic is still better than Pessimistic. For the other benchmarks, Pessimistic becomes better than Optimistic by an average of 12% for the C and 16% for the C++ programs for deep speculation and less when only one branch can be left unresolved. Tying up the interface to the next level of the hierarchy becomes increasingly important when the Optimistic and Resume policies are implemented (c.f., *wrong_icache* and *bus* in Figure 2).

As mentioned earlier, the goal of Resume is to cut down on the penalty when an I-cache miss is encountered on the wrong path and thus it performs significantly better than Optimistic for large cache penalties. On average, Pessimistic and Resume have approximately the same performance. However, as noted in the previous subsection, Resume increases memory traffic that may tie up bus bandwidth while Pessimistic does not.

5.2.2 Influence of the depth of speculation

Increasing the depth of speculation yields an average improvement (decrease in ISPI) in the Oracle policy of over 16% when moving from 1 to 2 unresolved branches and another 7% improvement when moving from 2 to 4 unresolved branches (Table 5). If the Fortran programs are discarded (improvements there are only 6% when passing from depth 1 to depth 2 and 5% from 2 to 4) then the improvements are more important: 24% and 8% respectively. The other policies follow the same pattern. Although not shown, the same trend holds true for large miss penalties.

The main reason for the difference is that there is a trade-off between the amount of time due to increased mispredict/misfetch penalties on a deep speculative path and the larger stall time waiting until branches are resolved in a shallow speculative path. In Figure 1, there is no slow down due to reaching the limit of speculation depth (*branch_full* is small or non-existent) when up to 4 branches can be outstanding. However, this slow down dominates other effects when only one branch can be left unresolved and accounts for the superiority of the deep speculation for all policies. A minor counter-effect is the increase in waits due to cache misses on the wrong path (*wrong_icache*) in the deep speculative case which slightly reduces the advantage of the Optimistic policy.

5.2.3 Influence of cache size

Program	Oracle	Opt	Res	Pess	Dec
doduc	0.52	0.53	0.51	0.56	0.57
fpppp	0.35	0.35	0.35	0.44	0.44
su2cor	0.12	0.12	0.12	0.12	0.12
ditroff	1.03	1.08	1.01	1.10	1.10
gcc	1.33	1.43	1.32	1.49	1.51
li	0.89	1.04	0.92	0.90	0.96
tex	0.70	0.74	0.69	0.80	0.80
cfront	1.50	1.70	1.50	1.74	1.79
db++	0.65	0.69	0.65	0.69	0.69
groff	1.39	1.56	1.43	1.55	1.58
idl	0.79	0.82	0.77	0.85	0.85
lic	1.19	1.29	1.17	1.36	1.37
porky	0.89	0.93	0.88	0.95	0.97
Average	0.87	0.94	0.87	0.97	0.98

Table 6: *Effect of Cache Size. This table shows the ISPI penalty for direct mapped 32K caches with a miss penalty of 5 cycles.*

Table 6 shows the ISPI penalty for an architecture with a 32K I-cache. When the cache size is large (32K), the miss ratios are small, often less than 1%. Therefore the impact of the various policies will be reduced. The average ISPI difference between Resume and Pessimistic is only 10%, as opposed to 19% for the 8K cache. However, for applications that have a high miss rate (i.e., *gcc*, *tex*, *cfront*, *groff*, and *lic*), Resume still offers a modest improvement (8-12%) in instruction issue rates over Optimistic.

5.3 Next-line prefetching

Next-line prefetching has been shown to significantly improve the performance of the I-cache [Smith & W.-C.Hsu 92]. In this section, we look into the interaction between prefetching and speculative execution.

Figure 3 shows the ISPI breakdown for the base configuration with and without next-line prefetching. The bars labelled with "Pref" reflects the performance of the I-cache fetch policy that includes next-line prefetching.

In general, next-line prefetching improves the performance of all the policies. It also decreases the relative difference between the different policies. For example, without next-line prefetching the difference between Resume and Pessimistic for *gcc* is 0.36 ISPI. With next-line , this difference decreases to 0.15 ISPI, but Resume still performs better than Pessimistic. This shows that processing speculative I-cache misses in the Resume policy without next-line prefetching gives approximately the same performance as the Pessimistic policy with next-line prefetching.

Unlike the case where prefetching is not used, the *rt_icache* components for both Resume and Pessimistic are effectively equal, which implies that the wrong path prefetching effect from Resume is small. The decreased *bus* component for Pessimistic reflects the overlap between prefetching and waiting for branches to be resolved on an I-cache miss. However, not all of the stall time can be overlapped, and Pessimistic still performs worse than Resume when the miss latency is small.

For the case with a large I-cache miss penalty, Figure 4 shows that next-line prefetching may be detrimental to performance. Even Oracle shows a decrease in performance because required I-cache misses have to wait for the bus when a prefetch is in progress. With long I-cache miss latencies, as shown before, aggressive instruction fetch activity may actually hurt performance, and one needs to be more cautious.

Program	Oracle	Resume	Pessimistic
doduc	1.22	1.28	1.23
fpppp	1.02	1.03	1.03
su2cor	1.26	1.27	1.26
ditroff	1.41	1.68	1.47
gcc	1.39	1.62	1.45
li	1.29	1.62	1.29
tex	1.34	1.54	1.38
cfront	1.35	1.56	1.39
db++	1.43	1.74	1.47
groff	1.46	1.71	1.49
idl	1.64	1.81	1.67
lic	1.28	1.52	1.32
porky	1.51	1.83	1.54
Average	1.35	1.56	1.38

Table 7: *Effect of prefetching on memory traffic for the baseline architecture. The numbers indicate the ratio between the number of memory accesses by the policy with next-line prefetching and the number of memory accesses by Oracle without next-line prefetch.*

Table 7 shows the net increase in memory traffic of the different policies with next-line prefetching. The high numbers indicate that extra memory traffic is generated by prefetching even for the Pessimistic policy. Memory traffic increases as much as 67% for *idl*. As mentioned previously, this increased memory traffic may lead to degraded performance when the processor contends with the prefetch for the limited bandwidth available to memory.

Summary

Our results show that the policy of choice depends on the latency between the first level I-cache and the next level in the memory hierarchy. When the latency is small (comparable to the mispredict penalty) then Resume is best. It combines the benefits of not stalling when on the correct speculative path and of prefetching instructions when on the incorrect one. It reduces the stalling penalty when an I-cache miss is in progress by allowing resumption of execution as soon as a mispredict/misprefetch has been detected. The combination of Resume and next-line prefetching further decreases ISPI.

The memory traffic generated by the aggressive policies becomes increasingly important with higher latencies. Our results show that when the miss penalty is high, Pessimistic performs as well as Resume on average with less memory traffic. Next-line prefetching, which generates significant extra memory traffic, does not significantly decrease ISPI and therefore is not recommended in this case.

For all policies, deeper speculation reduces ISPI. The decrease in stalling time waiting for branches to be resolved dominates the increase in mispredict/misfetch penalties and the increased I-cache misses on the incorrect path.

6 Conclusion

In this paper, we assessed the impact of I-cache miss policies in the context of a superscalar architecture with speculative execution. In addition to the unrealizable Oracle policy which always knows the correct execution path, we examined aggressive fetch policies (Optimistic and Resume) and conservative ones (Pessimistic and Decode). We also investigated the effect of combining next-line prefetching with these policies. We considered a blocking I-cache, with the possibility of having a single outstanding miss request in the Resume and next-line prefetching cases.

We advocate the Resume policy with next-line prefetching if the latency is small (e.g., for an on-chip hierarchy of caches). The extra hardware complexity required by this policy is small, and our results show that it offers a modest gain in instruction issue bandwidth over the other policies. If the latency is large, both Resume and Pessimistic without next-line prefetching do well. However, Pessimistic is preferable because it requires less memory bandwidth and is easier to design. Further study should indicate whether more complex I-cache structures and memory interfaces, such as non-blocking I-caches and pipelining miss requests, and software techniques, like profile driven basic-block reordering, will significantly improve the I-cache performance at a reasonable cost and complexity.

Acknowledgements

We would like to thank Amitabh Srivastava and Alan Eustace for developing ATOM, Digital Equipment Corporation for an equipment grant, and the anonymous referees for providing helpful comments. This work was partially supported by an ARPA Fellowship in High Performance Computing administered by the Institute for Advanced

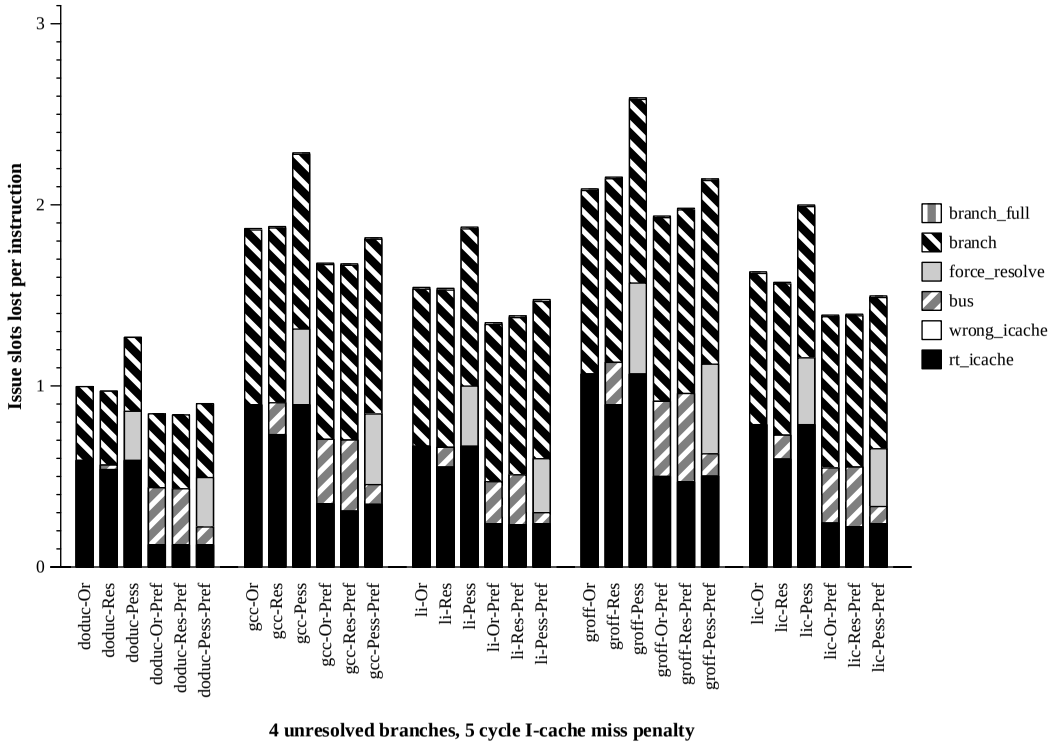

Figure 3: *Effect of next-line prefetching. This figure shows the breakdown of the penalty components for the baseline architecture with and without next-line prefetching for Oracle, Resume, and Pessimistic. The bars labeled with Pref reflects the performance of the I-cache fetch policy with next-line prefetching.*

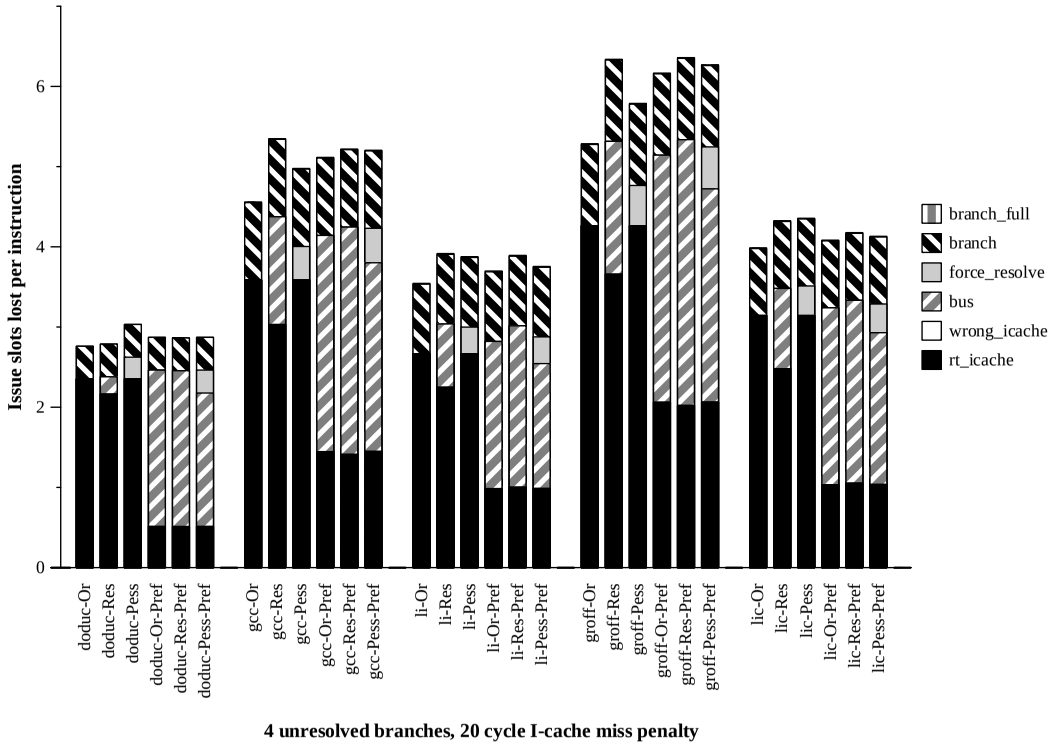

Figure 4: *Next-line prefetching with long I-cache miss penalty. The figure shows the effect of prefetching on Oracle, Resume, and Pessimistic for a system with a long I-cache miss latency.*

Computer Studies, University of Maryland, in part by NSF grants No. ASC-9217394, No. CCR-9123308, and No. CCR-9401689, in part by Intel Corp., and in part by ARPA contract ARMY DABT63-94-C-0029.

References

[Bray & Flynn 91] Bray, B. and Flynn, M. J. Strategies for branch target buffers. In *24th Annual International Symposium and Workshop on Microprogramming*, pages 42–50. ACM, 1991.

[Calder & Grunwald 94] Calder, B. and Grunwald, D. Fast & accurate instruction fetch and branch prediction. In *21st Annual International Symposium on Computer Architecture*. ACM, April 1994.

[Jouppi 90] Jouppi, N. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In *17th Annual International Symposium of Computer Architecture*, pages 364–373. ACM, May 1990.

[Lee & Smith 84] Lee, J. K. F. and Smith, A. J. Branch prediction strategies and branch target buffer design. *IEEE Computer*, 21(7):6–22, January 1984.

[McFarling & Hennessy 86] McFarling, S. and Hennessy, J. Reducing the cost of branches. In *13th Annual International Symposium of Computer Architecture*, pages 396–403. ACM, 1986.

[McFarling 93] McFarling, S. Combining branch predictors. TN 36, DEC-WRL, June 1993.

[Pan et al. 92] Pan, S.-T., So, K., and Rahmeh, J. T. Improving the accuracy of dynamic branch prediction using branch correlation. In *Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 76–84, Boston, Mass., October 1992. ACM.

[Perleberg & Smith 93] Perleberg, C. and Smith, A. J. Branch target buffer design and optimization. *IEEE Transactions on Computers*, 42(4):396–412, April 1993.

[Pierce & Mudge 94] Pierce, J. and Mudge, T. Wrong-path instruction prefetching. Technical Report CSE-222-94, University of Michigan, 1994.

[Pierce 95] Pierce, J. E. *Cache Behavior in the Presence of Speculative Execution - The Benefits of Misprediction*. PhD dissertation, University of Michigan, 1995.

[Smith & W.-C.Hsu 92] Smith, J. and W.-C.Hsu. Prefetching in supercomputer instruction caches. In *Supercomputing '92*, pages 332–339, November 1992.

[Smith 81] Smith, J. E. A study of branch prediction strategies. In *8th Annual International Symposium of Computer Architecture*. ACM, 1981.

[Smith 82] Smith, A. J. Cache memories. *Computing Surveys*, 14(3):473–530, September 1982.

[Srivastava & Eustace 94] Srivastava, A. and Eustace, A. Atom: A system for building customized program analysis tools. In *Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation*. ACM, 1994.

[Yeh & Patt 92a] Yeh, T.-Y. and Patt, Y. N. Alternative implementations of two-level adaptive branch predictions. In *19th Annual International Symposium of Computer Architecture*, pages 124–134, Gold Coast, Australia, May 1992. ACM.

[Yeh & Patt 92b] Yeh, T.-Y. and Patt, Y. N. A comprehensive instruction fetch mechanism for a processor supporting speculative execution. In *25th Annual International Symposium on Microarchitecture*, pages 129–139, Portland, Or, December 1992. ACM.

[Yeh & Patt 93] Yeh, T.-Y. and Patt, Y. N. A comparison of dynamic branch predictors that use two levels of branch history. In *20th Annual International Symposium of Computer Architecture*, pages 257–266, San Diego, CA, May 1993. ACM.