Alternative Fetch and Issue Policies for the Trace Cache Fetch Mechanism

Copyright 1997 IEEE. Published in the Proceedings of Micro-30, December 1-3, 1997 in Research Triangle Park, North Carolina. Personal use of this material is permitted. However, permission to reprint/republish this material for advertising or promotional purposes or for creating new collective works for resale or redistribution to servers or lists, or to reuse any copyrighted component of this work in other works, must be obtained from the IEEE. Contact: Manager, Copyrights and Permissions / IEEE Service Center / 445 Hoes Lane / P.O. Box 1331 / Piscataway, NJ 08855-1331, USA. Telephone: + Intl. 908-562-3966.

Alternative Fetch and Issue Policies for the Trace Cache Fetch Mechanism

Daniel Holmes Friendly Sanjay Jeram Patel Yale N. Patt

Department of Electrical Engineering and Computer Science
The University of Michigan
Ann Arbor, Michigan 48109-2122
{ites, sanjayp, patt}@eecs.umich.edu

Abstract

The increasing widths of superscalar processors are placing greater demands upon the fetch mechanism. The trace cache meets these demands by placing logically contiguous instructions in physically contiguous storage. It is capable of supplying multiple fetch blocks each cycle.

In this paper we examine two fetch and issue techniques, partial matching *and* inactive issue*, that improve the overall performance of the trace cache by improving the effective fetch rate. We show that for the SPECint95 benchmarks partial matching increases the overall performance by 12% and adding inactive issue increases performance by 15%. Furthermore we apply these two techniques to issue blocks from trace segments which contain multiple execution paths.*

We conclude with a performance comparison between a trace cache implementing partial matching and inactive issue and an aggressive single block fetch mechanism. The trace cache increases performance by an average of 25% over the instruction cache.

1 Introduction

To take advantage of the increasing execution bandwidth of modern superscalar processors, fetch mechanisms must be able to supply a larger number of instructions to the processing core. Current instruction cache designs are limited by difficulty in fetching a branch and its target in a single cycle. This effectively limits the fetch bandwidth of an instruction cache to the average length of a fetch block. A fetch block roughly corresponds to a compiler basic block: it is a dynamic sequence of instructions starting at the current fetch address and ending at the next control instruction.

Recently, the trace cache [13, 14, 11] has been proposed as a technique that is able to overcome this bandwidth limitation. By placing logically contiguous instructions in physically contiguous storage, the trace cache is able to supply multiple fetch blocks each cycle.

The driving force behind the trace cache concept is its ability to deliver more instructions per cycle. The trace cache attains a higher effective fetch rate — the average number of correct instructions issued for each fetch that returns on-path instructions — than a conventional instruction cache fetch mechanism.

In this paper we examine several enhancements to the trace cache mechanism that increase the effective fetch rate without severely increasing the number of cycles lost to branch mispredictions and cache misses.

The first technique, partial matching, suggested by Rotenberg et al [14], allows the trace cache to respond with only the portion of a trace cache line selected by the predictor. Partial matching improves the effective fetch rate by increasing the percentage of instructions that are fetched from the trace cache.

The second technique, called inactive issue, allows for further improvement in the effective fetch rate by issuing all of the instructions fetched from the trace cache, even when they diverge from the predicted path. If the prediction is incorrect, these instructions will have been fetched, issued and possibly executed when the mispredicted branch resolves. In this way, the effective fetch rate of the processor is increased as more correct path instructions may be issued in the event of a branch misprediction. Inactive issue can be viewed as a limited form of hardware predication. Essentially, the blocks within a trace cache line which diverge from the predicted path are predicated, and their results do not change the state of the processor visible to subsequent instructions until they are known

to be on the correct path.

The use of partial matching and inactive issue enables the trace cache to depart from the single path paradigm. If we allow the fill unit to collect trace segments that form a multipath tree, the trace cache can use these techniques to fetch and issue instructions from both paths of a predicted branch.

This paper is divided into 8 sections. The next section discusses the related work in the field. Sections 3 and 4 explain the techniques of partial matching and inactive issue. Our simulation environment is detailed in section 5. Section 6 examines the performance impact of these techniques. The use of these techniques to enable dual path execution is presented in section 7. Section 8 provides a comparison with existing fetch technology. Section 9 gives a brief summary of the work presented.

2 Related work

A precursor to the trace cache was first introduced by Melvin, Shebanow and Patt [10]. They proposed the fill unit to compact a fetch block's worth of instructions into an entry in a decoded instruction cache. A hit in the decoded instruction cache results in a larger atomic unit of work than would be possible if the individual instructions were fetched and decoded one per cycle. In [9] the performance implications of the fill unit are discussed and the idea of dynamically combining fetch blocks into larger "execution atomic units" (EAUs) to further increase the fetch bandwidth is first proposed. Two other extensions of the original schemes were presented by Franklin and Smotherman. In [4], they applied the original fill unit idea to dynamically create VLIW instructions out of RISC-type operations. They reworked the fill unit finalization strategy by restricting the type of instruction dependencies allowed in a fill unit line and by filling both paths beyond a conditional branch. In [15], they demonstrated how a fill unit could help overcome the decoder bottleneck of a Pentium Pro type processor.

In 1994, Peleg and Weiser filed a patent on the trace cache concept [13]. The concept was further investigated by Rotenberg et al [14]. They presented a thorough comparison between the trace cache scheme and several hardware-based high-bandwidth fetch schemes and showed the advantage of using a trace cache, both in performance and latency. In this work they suggested the concept of partial matching. They also presented a multiple branch predictor capable of making three predictions per cycle. They, however, relegated the trace cache to a supporting role in the fetch engine, showing that in a machine with a 128KB instruction cache and a 4KB trace cache, performance is boosted by 28% on the Instruction Benchmark Suite (IBS) and the SPECint92 benchmarks over a single block fetch mechanism with a GAg single branch predictor.

Patel et al [11] examined a wide range of trace cache and fill unit characteristics. They showed that higher performance can be achieved when a fetch mechanism is partitioned into a large trace cache with a small instruction cache than with the opposite configuration. Furthermore, they showed that a fetch mechanism composed of a 128KB trace cache with a gshare multiple branch predictor outperforms a 128KB instruction cache with a very aggressive hybrid single branch predictor by 25%.

The inactive issue technique presented in section 4 is similar to the misprediction recovery cache work done by Bondi et al [1]. Both schemes reduce the effects of the branch misprediction penalty by either saving processed instructions on the other path or concurrently processing them along with the predicted path.

Franklin and Smotherman [4] examined creating tree-like VLIWs using a fill unit, a technique similar to what we propose in section 7. They examined the multipath trace segments in the context of statically scheduled machines. Similar work was done by Ebcioğlu. Here, we explore a similar concept applied to a dynamically scheduled machine.

3 Partial matching

Figure 1 shows the trace cache fetch mechanism. Hits in the trace cache result in multiple fetch blocks being delivered for issue. Misses are serviced by the instruction cache, which responds with up to one fetch block. A multiple branch predictor or path predictor sequences between trace segments. The fill unit creates trace segments by collecting and combining blocks as they are processed by the machine, either as they are issued or as they retire. Completed segments are written into the trace cache.

The advantage of using a trace cache is its ability to supply instructions across multiple fetch blocks each cycle. The branch predictor is used to determine which path from the current fetch address is most likely. If the segment contained in the trace cache differs from the path requested by the branch predictor, the difference must be resolved in some manner. The straight forward approach is to require that the segment supplied by the trace cache match the branch predictor. If they differ, the trace cache responds with

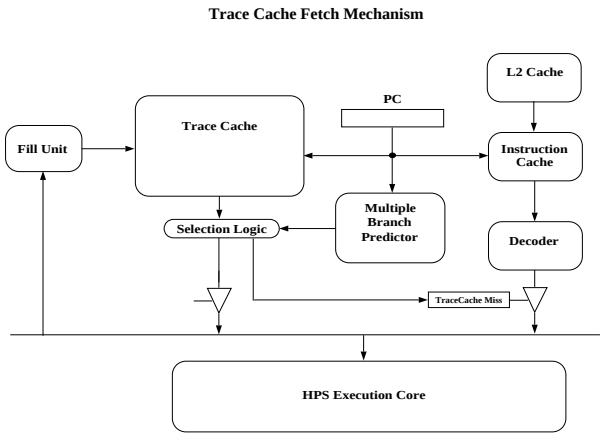

Figure 1: The trace cache datapath.

a miss and the instructions are fetched from the instruction cache.

Alternatively, the trace cache can be designed to signal a hit if the fetched segment only partially matches the requested path. Each fetch cycle the predictions made by the branch predictor are used to select which blocks within the segment fetched from the cache will be issued to the core. If a trace cache line contains ABC, and the predictor predicts ABD, blocks A and B are supplied. If the predictor selects AC, as can happen with some control flows, only A is supplied. This technique is called partial matching.

The use of partial matching allows requests to hit in the trace cache more frequently. In doing so there are two benefits. First, although the average number of instructions supplied by each trace cache hit will decrease, it will supply at least as many instructions as the instruction cache. Furthermore, if more than one block of the trace cache segment matches the requested path, multiple blocks will be issued. In this way partial matching is able to increase the number of instructions fetched. If the blocks are on the correct execution path, the effective fetch rate is boosted. The second benefit of partial matching is that it decreases the fetch engine's reliance on the supporting instruction cache. Fewer resources can be allocated to the instruction cache and those that are allocated can be used for other purposes, eg. unused read/write ports can be used for instruction prefetching.

In order to implement partial matching three things must happen: the matching blocks must be determined, blocks not selected for issue must be invalidated and the proper next fetch address must be generated. Detecting the matching blocks is done by comparing the direction of each internal branch on the selected cache line with the current prediction. Blocks which match are issued, blocks which do not match are invalidated. The invalidation need not happen immediately. Instructions can be invalidated in the issue stage before the renaming structures are updated. The next fetch address must be generated by the end of the fetch cycle in order to initiate another fetch in the following cycle. To accomplish this, each line of the trace cache contains the target addresses corresponding to all the possible exits from the trace cache segment. The branch prediction is used to select which of these targets to use as the next fetch address. A trace cache line containing up to three conditional branches can have up to four possible targets.

4 Inactive issue

Although partial matching increases the number of instructions that are issued each cycle, it does not take advantage of the entire segment of instructions fetched from the cache. Blocks within the trace cache segment which do not match the predicted path are discarded. As long as the prediction is correct, this does not impact the effective fetch rate of the processor. If the prediction is incorrect however, an opportunity to issue a greater number of correct instructions has been missed. To take advantage of this opportunity we are proposing a new issue technique called inactive issue.

With inactive issue all of the blocks within a trace cache line are issued into the processor whether or not they match the predicted path. The blocks that do not match the prediction are said to be issued inactively. Although these inactive instructions are renamed and receive physical registers for their destination values, the changes they make to the register alias table (RAT) [5] are not considered valid for subsequent issue cycles. Thus instructions along the predicted path view the speculative state of the processor exactly as if the inactive blocks had not been issued. When the branch that ended the last active block resolves, if the prediction was correct, the inactive instructions are discarded. If the prediction was incorrect, the processor has already fetched, issued and possibly executed some instructions along the correct path.

This technique has a few advantages over partial matching. Primarily, it reduces the impact of branch mispredictions. Inactive issue is able to hide a portion of the branch resolution latency by making some correct path instructions (which follow a mispredicted branch) available for processing earlier. When the mispredicted branch resolves, the recovery state of

the processor is further along the correct path than it would have been if the inactive instructions had not been issued. Furthermore, if any inactively issued branches have resolved, the predictions made for them are not used. Their actual outcome is known.

Inactive issue also has the potential to reduce the amount of block duplication within the trace cache. Without inactive issue, when a mispredicted branch resolves, the next fetch might correspond to an address that is already within the trace cache but is not accessible, as it could be an interior block (ie. blocks B and C of the segment ABC.) With inactive issue however, the block no longer has to be refetched, and need not be rewritten to the trace cache into a new entry. Block duplication is therefore reduced and the trace cache storage is used more effectively. This also reduces the pressure on the multiple branch predictor, causing a reduction in interference. This has the potential to benefit large programs whose working sets conflict for space within the trace cache.

To implement inactive issue, modifications must be made to the renaming and recovery structures. The HPS model uses a checkpointed register alias table to maintain both the architectural and speculative state of the processor. The changes needed to implement inactive issue include adding an active bit to each checkpoint in the table. As the checkpoints are created, this bit is set if the instructions in the corresponding block are issued actively and the bit is cleared if the instructions are issued inactively. The most recent active checkpoint is used as the speculative state of the machine when new instructions are issued. When a branch resolves and is determined to be mispredicted, the inactive checkpoints immediately following the resolved checkpoint become active and all subsequent checkpoints, corresponding to instructions along the incorrect path, are flushed from the pipelines and the instruction window. The fetch proceeds from the (possibly predicted, possibly actual) target of the newly activated checkpoint. If the branch resolves correctly predicted, the inactive checkpoints are simply invalidated. Implementing inactive issue in other renaming schemes is also possible, however because of space constraints, we will not detail them here.

Figure 2 illustrates the blocks issued from a fetched cache line by the three different policies.

Another implementation issue concerning this technique is how the fill unit should treat instructions which are issued inactively. The fill unit can collect blocks either as they are issued or as they are retired. Results presented in [11] show that the performance difference between these two options is minimal. If

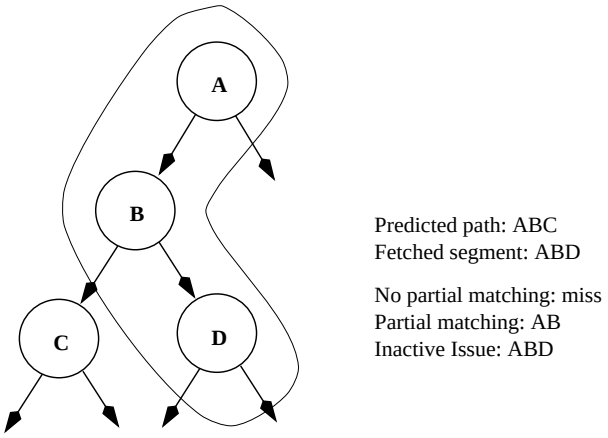

Figure 2: An example of the different issue policies.

the fill unit is designed to collect blocks at issue time, a choice must be made whether to include inactively issued instructions or to discard them. If the fill unit keeps inactive blocks, then the segments stored within the trace cache will tend to remain static. Since all of the blocks within the trace cache line are issued, the fill unit will have less opportunity to create new segments in response to changes in the predicted outcomes. Thus the segment ABC is likely to remain resident even if the predictor consistently requests blocks ABD. If the fill unit is designed to discard inactive blocks, it will respond to the changes in the branch predictor, creating new segments as the predictions through the control path change.

If the fill unit collects blocks as they retire, the distinction between active and inactive instructions becomes moot. The advantage of using a retire time fill unit is that it knows the true directions of all conditional branches and thus will consider only the on-path instructions. All off-path instructions are discarded and all on-path instructions are included, regardless of their issue status. The main drawback of the retire time fill unit is implementation complexity. Instructions must be driven out of the instruction window into the fill unit as they retire or, alternatively, the fill unit must be capable of buffering as many instructions as the window. For the experiments in this paper the fill unit collects instructions as they retire.

5 Experimental model

An executable-driven HPS [12] simulator modeling wrong path effects was used for this study. The simulator was implemented using the simplescalar 2.0 tool suite [2]. All instructions undergo four stages of pro-

cessing: fetch, issue, schedule, execute. All stages take at least one cycle. Figure 3 shows a block diagram of the pipeline model.

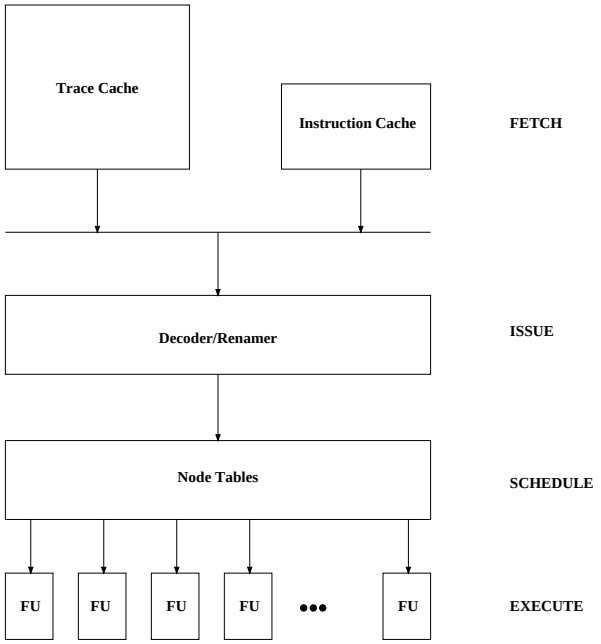

Figure 3: Trace Cache/HPS pipeline diagram.

The fetch engine, capable of supplying up to 16 instructions per cycle, includes a large 1K entry (approximately 64KB) trace cache and a smaller supporting instruction cache. Each trace cache line may contain up to 16 instructions, comprising at most three fetch blocks. (Unconditional branches are regarded as noops by the trace cache and thus are not considered to terminate blocks.) Returns, indirect branches and serializing instructions force the termination of a trace cache segment. Subroutine calls do not. The first level instruction cache was fixed at 4KB. Both caches were modeled as 4-way set associative. A 256KB unified second level cache provides instruction and data with a latency of six cycles in the case of first level cache misses. The branch predictor modeled, a variation of the gshare predictor designed for use with the trace cache [11], provides up to three predictions per cycle. The size of the pattern history table was fixed at 16K entries. An ideal return address stack was modeled.

The trace cache we modeled has no path associativity. Only one segment starting at a particular fetch block can be stored in the cache at any time (ie. ABC and ABD cannot be concurrently resident). See [11] for a discussion and performance results on path associativity. Also, the fill unit writes only segments which are composed of retired blocks (ie. retire time fill unit).

Operation	Latency
simple INT ops	1
IMULT	3
IDIV	20
LOAD	2 (min)
STORE	2
FADD	2
FMUL	4
FDIV	12

Table 1: Functional unit latencies.

Benchmark	Input set
compress	test.in
gcc	jump.i
go	2stone9.in
ijpeg	penguin.ppm
li	train.lsp
m88ksim	dhry.test
perl	scrabbl.pl
vortex	vortex.in

Table 2: Benchmarks and datasets used. All benchmarks were simulated to completion.

The execution engine of the simulator includes an instruction window of 16x32 instructions and 16 functional units. The functional units are uniform and capable of all operations. A 32KB L1 data cache was used. The latencies of the operations are listed in Table 1. The HPS model uses checkpoint repair [5] to recover from branch mispredictions and exceptions. The execution engine is capable of creating three checkpoints each cycle.

All experiments were performed on the SPECint95 benchmarks. Table 2 lists the benchmarks and the input sets [1]. All simulations were run until completion (except ijpeg, 500M instructions).

6 Results

We have simulated non-partial matching, partial matching and inactive issue for several different configurations, varying the size of the trace cache and the

[1] Vortex and go were simulated with abbreviated versions of the test input. Compress was simulated on a modified version of the test input with an initial list of 30000 elements.

branch predictor. As the trends remained consistent throughout, we will only present the results for a processor with a 64KB trace cache using a gshare style multiple predictor maintaining 14 bits of history. Figure 4 presents the speedup of the partial matching and inactive issue techniques over the baseline trace cache.

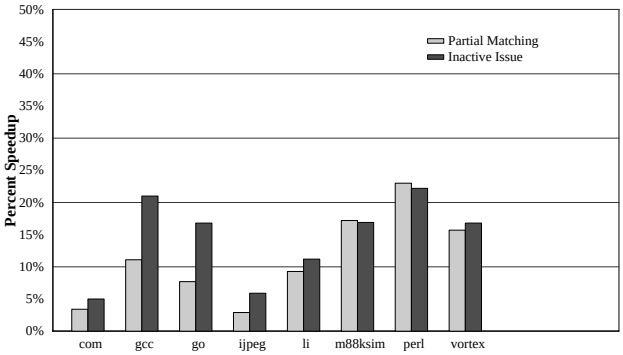

Figure 4: Relative speedup of partial matching and inactive issue over a non-partial matching trace cache.

As the figure shows, partial matching significantly increases the performance across all of the benchmarks. The improvement over the non-partial matching configuration is 12% on average. The actual performance, in instructions retired per cycle, is presented in table 3.

Benchmark	No Partial Matching	Partial Matching	Inactive Issue
compress	4.20	4.34	4.41
gcc	2.61	2.90	3.16
go	2.28	2.46	2.66
ijpeg	6.28	6.46	6.65
li	4.05	4.47	4.51
m88ksim	4.45	5.22	5.20
perl	3.81	4.69	4.66
vortex	5.23	6.05	6.11

Table 3: Instructions retired per cycle on the three configurations.

Adding inactive issue shows a further improvement over non-partial matching. The increase, averaged across all the benchmarks, is 15%. For the benchmarks that are easily predicted, adding inactive issue on to partial matching shows no significant improvement.

To understand the performance benefits of partial matching and inactive issue we examine how these different schemes impact the three main factors which affect the delivery of fetch bandwidth: effective fetch rate, total cache miss penalty and total branch misprediction penalty.

The benefits of both partial matching and inactive issue derive partly from their ability to boost the number of instructions supplied by the fetch mechanism each cycle. Table 4 lists the effective fetch rate of each configuration for all of the benchmarks. The average increase in effective fetch rate for the partial matching configuration is 18%, while for inactive issue the average increase over the non-partial matching case is 20%. For both techniques the increase is due to the ability of the trace cache to provide a larger portion of the selected trace cache line.

Benchmark	No Partial Matching	Partial Matching	Inactive Issue
compress	8.89	10.27	10.41
gcc	7.39	8.79	9.29
go	8.40	9.07	9.67
ijpeg	11.28	11.98	12.22
li	7.13	9.17	9.32
m88ksim	8.48	10.77	10.77
perl	8.67	10.42	10.49
vortex	9.24	10.71	10.74

Table 4: Effective fetch rates for the 3 issue techniques examined.

The second factor affecting fetch bandwidth is the total cache miss penalty. Table 5 lists the number of cycles spent waiting for instructions from the caches each time an on-path fetch is initiated. Partial matching reduces the cache miss penalty by an average of 61%, and adding inactive issue reduces it by an average of 64%. Both techniques increase the number of cycles in which multiple blocks are issued, lowering the total number of cache accesses made. In addition, without partial matching the instruction cache responds to requests which can be fulfilled by the trace cache, increasing the redundancy between the trace cache and the instruction cache and making inefficient use of the storage in the fetch mechanism. For m88ksim and perl, more efficient use of the storage results in a significant reduction in cache miss cycles. Inactive issue, as mentioned in section 4, is able to reduce some of the block duplication by allowing inactively issued blocks to remain as interior blocks. This accounts for the modest reduction in cache miss penalty over the partial matching technique.

The third factor affecting fetch bandwidth is the

Benchmark	No Partial Matching	Partial Matching	Inactive Issue
compress	14,619	-55%	-54%
gcc	12,086,462	-38%	-45%
go	4,145,186	-29%	-43%
ijpeg	82,974	-41%	-43%
li	588,612	-97%	-97%
m88ksim	7,295,252	-99%	-98%
perl	1,677,552	-92%	-91%
vortex	9,554,403	-35%	-37%

Table 5: The number of cycles spent by the baseline trace cache fetching on path instructions and the percent reductions for partial matching and inactive issue.

Benchmark	No Partial Matching	Partial Matching	Inactive Issue
compress	11,208,943	+12%	+11%
gcc	25,763,672	+11%	0%
go	40,357,295	+4%	-1%
ijpeg	28,898,835	+4%	-1%
li	18,316,735	+14%	+14%
m88ksim	44,583,054	+9%	+9%
perl	4,149,096	+11%	+13%
vortex	4,134,975	+18%	+15%

Table 6: The number of cycles spent by the baseline trace cache fetching wrong path instructions and the percentage difference for partial matching and inactive issue.

branch misprediction penalty. Table 6 lists the number of cycles spent fetching instructions from the wrong path for each of the techniques. Partial matching and inactive issue increase the penalty, by an average of 10% and 8% respectively. Misprediction penalty is based on two factors: the accuracy of the branch predictor and number of cycles between when a branch is mispredicted and when it resolves (branch resolution time). With partial matching, the average branch resolution time increases because more fetches are supplied by the trace cache resulting in more blocks being fetched per cycle. Fetching more blocks per cycle results in instructions waiting longer for data dependencies to be satisfied. For instance, on gcc the average branch resolution time increases from 7.64 cycles without partial matching to 8.60 cycles with partial matching. Some of the lost bandwidth is regained when inactive issue is added because it slightly reduces the average resolution time — inactive issue allows processing of the correct path to be overlapped with the incorrect path — and increases the effectiveness of the branch predictor by reducing interference.

However, for some benchmarks the resolution actually increases with inactive issue because contention for execution resources increases. If the instructions being issued inactively are on the wrong path they reduce the availability of the functional units for on-path instructions. An inactive, off-path instruction may get scheduled for execution instead of an on-path instruction. This is possible even with an oldest-first scheduling policy because with inactive issue multiple execution paths are inserted into the instruction window. If the inactive instruction gets scheduled, then an opportunity to start the on-path instruction has been lost. This increase in contention causes the average branch resolution time for perl to rise from 10.79 cycles with partial matching to 11.00 cycles with inactive issue.

7 Dual path extensions

The trace caches explored in [13, 14, 11] contain trace segments which are dynamic sequences of blocks. Each subsequent block in a segment is the target of the previous block. Partial matching enables another option: a trace segment need not consist of blocks from a single path of execution. The segment can be a multipath tree. The multiple branch predictor selects which path to issue from the fetched segment. If inactive issue is implemented, then the blocks on the non-selected paths are issued inactively.

In its simplest form, the multipath trace segment contains two separate paths, where the paths do not merge. Two cases of these simple dual path trace segments (along with normal single path segment) are shown in figure 5. The type-A segment is an example of a trace segment considered by the previous work on trace caches. The type-B and type-C segments are the simple dual-path cases. Here, the selected path is issued actively and the other path is issued inactively. If the selected path was incorrect and the inactive path was correct, then the inactive instructions need not be refetched.

Type-A segments can be trivially constructed by the fill unit. To create type-B and type-C segments requires an enhancement to the fill algorithm: if an inactive (or discarded, if filling at retire) block fits into the pending segment, add it. Any subsequent inactive (discarded) block is discarded. The next active block

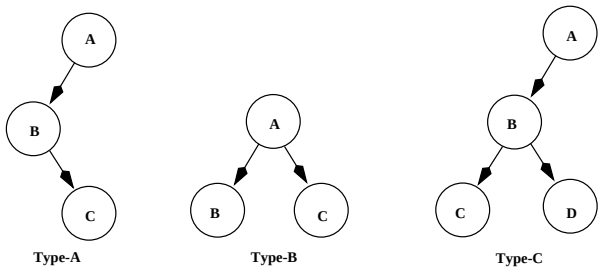

Figure 5: Three types of simple trace segments are shown here. Type-A contains only one path. Type-B and type-C trace segments encapsulate two paths.

will share the same predecessor block as the inactive (discarded) block. If the active block fits, add it and finalize. A simple dual path trace segment is created. If it doesn't fit, delete the inactive (discarded) block and add the active block as normal.

Each trace cache line must now contain five target addresses, since the type-C segment has five possible successors. Furthermore, the issue mechanism must be able to distinguish which instructions belong to which path so the instructions along the predicted path can be actively issued. This requires that each instruction contain a two-bit field indicating the directions of the previous branches in the segment. This field is compared with the prediction to decide whether the instruction should be actively or inactively issued.

The performance of a fetch mechanism which allows type-A, type-B, and type-C segments is shown in figure 6. The performance of compress, go, gcc, and ijpeg is enhanced over the inactive issue case. While some benchmarks benefit from the dual path segments, others do not. If branches within a segment are biased then it is better to have more blocks on the likely path than to have more blocks from each path. Forks within a segment are most useful when the corresponding branch is unbiased.

To make effective use of dual path segments, the fill unit must decide when a path should contain a fork and when it should not. Hardware schemes, such as a confidence mechanism proposed by Jacobsen [6], can assist the fill unit by identifying branches which the branch predictor is not predicting well. Whenever an unbiased branch is encountered, the pending segment is a possible candidate for dual paths. Furthermore, more complex dual path segments can be formed in addition to those presented in figure 5, such as segments which contain both a fork and a join.

To take proper advantage of all the available options will require that a great deal of work be done in

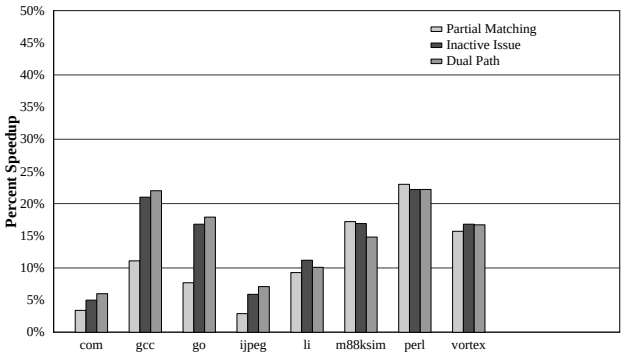

Figure 6: Speedup comparison of partial matching, inactive issue and dual path segments.

the fill unit. However, as the fill unit is off the critical processing path, its latency does not have a significant impact on performance. In figure 7, we show overall performance as the latency of the fill unit is increased up to 10 cycles. These results may seem counter-intuitive. There are two reasons for this behavior. Most importantly, this latency is incurred after the initial demand for the instructions has been satisfied. Therefore a penalty is only paid if the execution path loops back to these instructions before the segment has been written to the trace cache. Furthermore, in the process of satisfying the original demand request the blocks are brought at least into the instruction cache. Therefore any subsequent requests that occur before the full segment has been written into the trace cache can still be satisfied by the instruction cache. For some benchmarks (eg. vortex), a longer fill unit latency results in a slightly higher performance. A longer latency sometimes delays the update of a useful trace cache segment with a less useful one.

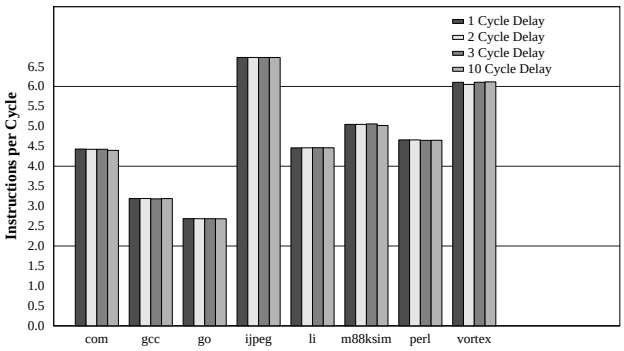

Figure 7: The effect of fill unit latency on performance. Performance is shown in IPC.

8 Comparison to an icache mechanism

The experiments thus far have evaluated various schemes which can be used with the trace cache fetch mechanism. An assessment of the mechanism would not be complete without a comparison to current dominant techniques for fetch engine design. Here we present a comparison of a trace cache configuration with an aggressive single block fetch mechanism.

The components of the single block mechanism we modeled are approximately the same size and access complexity as their trace cache counterparts. The single block mechanism consists of a single cycle, 64KB, 4-way set associative, dual-ported instruction cache capable of fetching up to 16 instructions or until the first control flow instruction each cycle. The next fetch address calculation is assisted by a 4KB branch target buffer. The single branch predictor is a hybrid predictor, consisting of two components: a 14-bit PAs predictor and a 14-bit gshare predictor. The selection between the components is done by a 14-bit gshare-style selector. Combining a per-address predictor with a gshare predictor was first proposed by McFarling [8]. Using a two-level mechanism to select between the two was proposed by Chang et al [3]. A similar version of this predictor is implemented in the DECChip 21264 [7]. Despite being limited to a single fetch block per cycle, such a mechanism attains a high fetch bandwidth because of the high accuracy of its branch predictor. The predictor attains a 93% prediction accuracy for gcc and 85% for go. The multiple branch predictor averages 90% for gcc and 80% for go.

Figure 8 presents the performance of a trace cache implementing partial matching and inactive issue relative to that of an icache mechanism. The trace cache achieves a 25% improvement overall. The actual performance, in instructions per cycle is presented in table 7. The benchmark go suffers from a high branch misprediction rate and a high cache miss rate in the trace cache. As a result, the performance of go on the trace cache is 1% lower than on an icache.

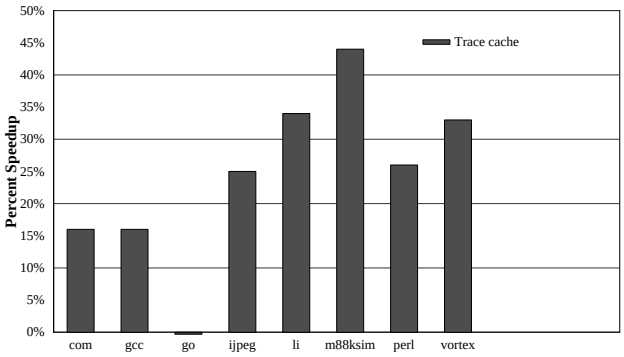

Figure 8: The performance speedup of a 64KB trace cache fetch mechanism with inactive issue against an aggressive single block fetch mechanism.

Benchmark	icache	trace cache
compress	3.79	4.41
gcc	2.64	3.18
go	2.68	2.66
ijpeg	5.33	6.65
li	3.35	4.50
m88ksim	3.60	5.20
perl	3.71	4.66
vortex	4.59	6.11

Table 7: Performance in IPC for the two fetch mechanisms.

9 Conclusions

In this paper we have examined two trace cache issue techniques that significantly improve the effective fetch rate. By allowing the trace cache to respond with only a portion of the requested path, partial matching improves performance of the SPECint95 benchmarks by an average of 12%. The benchmarks showing the largest improvements are perl, increasing 23%, and m88ksim, increasing 17%.

By issuing all of the instructions fetched from the trace cache, even when they diverge from the predicted path, inactive issue is able to improve performance over the non-partial matching case by an average of 15%. Adding inactive issue on top of partial matching boosts the performance of gcc by 21% and of go by 17% over a trace cache not implementing partial matching.

Both partial matching and inactive issue work to increase the effective fetch rate of the processor and decrease the cache miss penalty. In doing so, they improve the performance of the fetch mechanism. The use of these techniques is essential in order to take advantage of the fetch bandwidth potential of the trace cache. Furthermore, these techniques enable other options, such as dual-path segments, which show promise for greater increases in fetch bandwidth. Our preliminary study of the dual path work has shown that if the fill unit builds simple dual path segments whenever possible, performance increases slightly for the compress, go, gcc, and ijpeg benchmarks.

As the fill unit may be required to accomplish more complex tasks in the creation of dual path segments, we have also examined the impact of increased latency through the fill unit. Our study has shown that adding multiple cycles of latency in the fill unit has very little impact on the performance of the processor.

Finally we have compared the performance of the various trace cache design options to that of an aggressive, similarly sized single block fetch mechanism. The results show that the trace cache outperforms the single block fetch mechanism by an average of 25%.

10 Acknowledgements

We would like to thank the other current members of the HPS research group, Rob Chappell, Marius Evers, Peter Kim, Paul Racunas, Jared Stark, and several of its alumni. We would like to acknowledge the contributions of Todd Austin and thank our corporate sponsors — Intel, Motorola, and NCR — for funding our work.

References

[1] J. O. Bondi, A. K. Nanda, and S. Dutta, "Integrating a misprediction recovery cache (mrc) into a superscalar pipeline," in *Proceedings of the 29th Annual ACM/IEEE International Symposium on Microarchitecture*, pp. 14 – 23, December 1996.

[2] D. Burger, T. Austin, and S. Bennett, "Evaluating future microprocessors: The simplescalar tool set," Technical Report 1308, University of Wisconsin - Madison Technical Report, July 1996.

[3] P.-Y. Chang, E. Hao, T.-Y. Yeh, and Y. N. Patt, "Branch classification: A new mechanism for improving branch predictor performance," in *Proceedings of the 27th Annual ACM/IEEE International Symposium on Microarchitecture*, pp. 22–31, 1994.

[4] M. Franklin and M. Smotherman, "A fill-unit approach to multiple instruction issue," in *Proceedings of the 27th Annual ACM/IEEE International Symposium on Microarchitecture*, pp. 162–171, 1994.

[5] W. W. Hwu and Y. N. Patt, "Checkpoint repair for out-of-order execution machines," in *Proceedings of the 14th Annual International Symposium on Computer Architecture*, pp. 18–26, 1987.

[6] E. Jacobsen, E. Rotenberg, and J. E. Smith, "Assigning confidence to conditional branch predictions," in *micro96*, pp. 142–152, 1996.

[7] J. Keller, *The 21264: A Superscalar Alpha Processor with Out-of-Order Execution*, Digital Equipment Corporation, Hudson, MA, October 1996.

[8] S. McFarling, "Combining branch predictors," Technical Report TN-36, Digital Western Research Laboratory, June 1993.

[9] S. W. Melvin and Y. N. Patt, "Performance benefits of large execution atomic units in dynamically scheduled machines," in *Proceedings of the 1989 International Conference on Supercomputing*, pp. 427–432, 1989.

[10] S. W. Melvin, M. C. Shebanow, and Y. N. Patt, "Hardware support for large atomic units in dynamically scheduled machines," in *Proceedings of the 21st Annual ACM/IEEE International Symposium on Microarchitecture*, pp. 60–63, 1988.

[11] S. J. Patel, D. H. Friendly, and Y. N. Patt, "Critical issues regarding the trace cache fetch mechanism," Technical Report CSE-TR-335-97, University of Michigan Technical Report, May 1997.

[12] Y. Patt, W. Hwu, and M. Shebanow, "HPS, a new microarchitecture: Rationale and introduction," in *Proceedings of the 18th Annual ACM/IEEE International Symposium on Microarchitecture*, pp. 103–107, 1985.

[13] A. Peleg and U. Weiser. *Dynamic Flow Instruction Cache Memory Organized Around Trace Segments Independant of Virtual Address Line.* U.S. Patent Number 5,381,533, 1994.

[14] E. Rotenberg, S. Bennett, and J. E. Smith, "Trace cache: a low latency approach to high bandwidth instruction fetching," in *Proceedings of the 29th Annual ACM/IEEE International Symposium on Microarchitecture*, pp. 24 – 34, December 1996.

[15] M. Smotherman and M. Franklin, "Improving cisc instruction decoding performance using a fill unit," in *Proceedings of the 28th Annual ACM/IEEE International Symposium on Microarchitecture*, pp. 219–229, 1995.