

Improving Trace Cache Effectiveness with Branch Promotion and Trace Packing

Sanjay Jeram Patel Marius Evers Yale N. Patt
Advanced Computer Architecture Laboratory
Department of Electrical Engineering and Computer Science
The University of Michigan
Ann Arbor, Michigan 48109-2122
{sanjayp, olaf, 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. As a result, the trace cache delivers instructions at a high rate by supplying multiple fetch blocks each cycle.

In this paper, we examine two techniques to improve the number of instructions delivered each cycle by the trace cache. The first technique, branch promotion, dynamically converts strongly biased branches into branches with static predictions. Because these promoted branches require no dynamic prediction, the branch predictor suffers less from the negative effects of interference.

Branch promotion unlocks the potential of the second technique: trace packing. With trace packing, trace segments are packed with as many instructions as will fit, without regard to naturally occurring fetch block boundaries. With both techniques, the effective fetch rate of the trace cache jumps up 17% over a trace cache which implements neither.

On a machine where the execution engine has a very aggressive memory disambiguator, the performance of a machine using branch promotion and trace packing is on average 11% higher than a machine using neither technique.

1. Introduction

To maximize the performance of a wide-issue superscalar execution engine, the fetch mechanism must be capable of delivering at least the same bandwidth as the execution engine is capable of consuming. Fetch mechanisms consisting of a conventional instruction cache are limited by difficulty in fetching a branch and its target in a single cycle. They are limited to supplying on average one *fetch block* per cycle and therefore are not sufficient to deliver

the required bandwidth to an execution engine capable of executing instructions at a higher rate. Here, 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 [10, 11, 9] has been proposed as a technique that is able to overcome this fetch bandwidth limitation. By placing logically contiguous instructions in physically contiguous storage, the trace cache is able to supply multiple fetch blocks each cycle. The maximum number of fetch blocks supplied is limited by the size of a trace cache line and the capability of the branch predictor. If the branch predictor makes n individual predictions per cycle, then up to n fetch blocks can be supplied each cycle. Figure 1 shows the trace cache datapath.

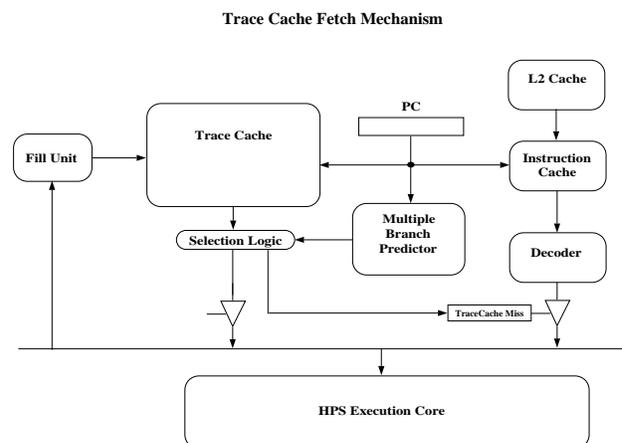


Figure 1. The trace cache datapath.

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 instructions for fetches that return instructions on the correct execution path — than a conventional instruction

cache fetch mechanism.

In this paper, we introduce two techniques for boosting the effective fetch rate of the trace cache. The first technique, called branch promotion, exploits the high frequency of conditional branches which are strongly biased in one direction. When the fill unit — the logic structure that creates the trace segments which are stored in the trace cache — detects a strongly biased branch, it promotes that branch into one with a static prediction. Promoted branches require no dynamic prediction and thus the number of promoted branches allowed on a trace cache line is not limited by the branch predictor bandwidth.

Branch promotion unlocks the potential of the second technique: trace packing. The fill unit can deal with blocks in two ways. It can (1) obey natural block boundaries present in the executable and merge a block into a pending segment only if it fits entirely or (2) split blocks arbitrarily, and in a greedy fashion place as many instructions into a pending segment as will fit. The remaining instructions of the block then begin the next trace segment. For example with trace packing, if there are 5 remaining instructions slots in a pending trace segment, an incoming block of 9 instructions would be split. The first 5 instructions would finish off the pending segment and the remaining 4 would begin the next trace segment. We show that branch promotion and trace packing, when used together, boost the delivered effective fetch rate of the trace cache by 17% over a trace cache that does neither. Furthermore, conditional branch prediction accuracy improves from an 8% misprediction rate to a 7% misprediction rate, due to the interference reduction from branch promotion.

When used together, these techniques boost the overall performance by 4% on the benchmarks simulated. In section 6, we present an analysis of the effects of branch promotion and trace packing on the delivered bandwidth of the front end. We show that the effective fetch rate is sufficiently maximized by using these techniques and that performance is limited by mispredicted branch resolution time, which increases as the effective fetch rate is increased. This increase is largely due to inefficiencies in the execution core. If the execution core performs perfect memory disambiguation, then branch promotion and trace packing boosts performance to 11% over a baseline with the same execution core.

2. Related work

A precursor to the trace cache was first introduced by Melvin, Shebanow and Patt [7]. They proposed the fill unit to compact a fetch block’s worth of instructions into an entry in a decoded instruction cache. 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. In [13], 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 [10]. The concept was further investigated by Rotenberg et al [11]. 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 [12], Rotenberg et al presented the design and performance implications of a processor which treats traces as the atomic unit of execution. Friendly et al [5] investigated two techniques, partial matching and inactive issue, which boost performance by 15% over a trace cache not implementing either.

3. Experimental model

An executable-driven simulator which allows the modeling of wrong path effects was used for this study. The simulator was implemented using the simplescalar 2.0 tool suite [1]. All instructions undergo four stages of processing before retirement: fetch, issue, schedule, execute. All stages take at least one cycle. Figure 2 shows a block diagram of the pipeline model.

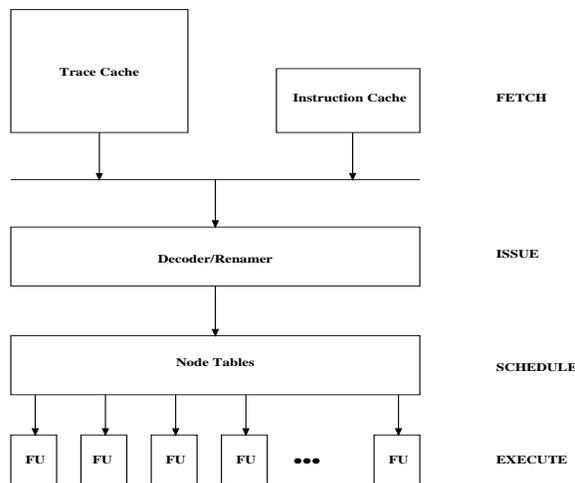


Figure 2. Simulator pipeline diagram.

The fetch engine, capable of supplying up to 16 instructions per cycle, includes a large 2K entry (approximately 128KB for instruction storage), 4-way set associative trace cache and a 4KB, 4-way supporting instruction cache. Each trace cache line may contain up to 16 instructions, comprising at most three fetch blocks. (Unconditional branches are not considered to terminate blocks within trace segments.) Returns, indirect branches and serializing instructions force the termination of a trace cache segment. Subroutine calls

do not. A 1MB unified second level cache provides instruction and data with a latency of six cycles in the case of first level cache misses. Misses in the second level cache take a minimum of 50 cycles to be fetched from memory.

The branch predictor modeled is the gshare predictor designed for use with the trace cache in [9] and provides up to three individual conditional branch predictions each cycle. The size of the pattern history table was fixed at 16K entries, each entry consisting of 7 2-bit counters (32KB of storage), as shown in figure 3. An ideal return address stack was modeled.

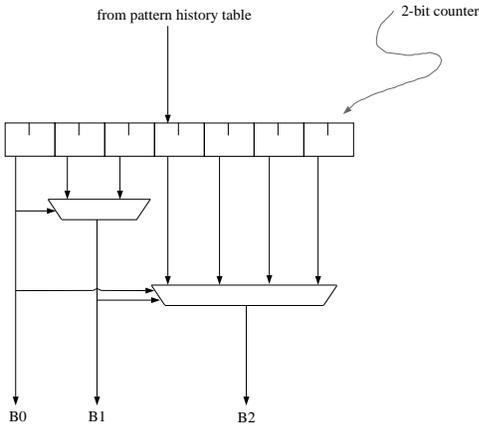


Figure 3. Multiple Branch Predictor.

The trace cache modeled does not utilize path associativity, meaning 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 [9] for a discussion and performance results on path associativity. Also, the fill unit collects blocks after they retire, building trace segments from the retired instruction stream.

Furthermore, the trace cache uses inactive issue [5]. With inactive issue, all 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. 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. We refer to this model as the baseline to which we apply the techniques discussed in sections 4 and 5

The execution engine is composed of 16 functional units, each unit capable of all operations. Instructions are dispatched for execution from a 64-entry reservation station (or node table) associated with each functional unit. A 64KB L1 data cache is used for data supply. The model uses checkpoint repair [6] to recover from branch mispredictions and exceptions. The execution engine is capable of creating

up to three checkpoints each cycle, one for each fetch block supplied. The memory scheduler waits for addresses to be generated before scheduling memory operations. No memory operation can bypass a store with an unknown address.

For reference, we have provided simulation results for a front end composed of a large dual-ported 128KB instruction cache capable of supplying a single fetch block per cycle. Since this model requires only a single prediction each cycle, aggressive hybrid branch prediction can be used. We used a hybrid predictor composed of a gshare component using 15 bits of global branch history and PAs component using 15 bits of local branch history and an 4K entry branch history table. The selector is accessed using the same 15-bit index as the gshare component. The total size data size of this predictor is approximately 32KB.

All experiments were performed on the SPECint95 benchmarks and on several common UNIX applications [14]. Table 1 lists the number of instructions simulated and the input set, if the input was derived from a standard input set¹. All simulations were run until completion (except li and ijpeg).

Benchmark	Inst Count	Input Set
compress	95M	test.in
gcc	157M	jump.i
go	151M	2stone9.in
ijpeg	500M	penguin.ppm
li	500M	train.lsp
m88ksim	493M	dhry.test
perl	41M	scrabbl.pl
vortex	214M	vortex.in
gnuchess	119M	
ghostscript	180M	
pgp	322M	
python	220M	
gnuplot	284M	
sim-outorder	100M	
tex	164M	

Table 1. Benchmarks

4. Branch promotion

The central benefit of the trace cache is its ability to boost the effective fetch rate beyond a single fetch block. As reported in previous studies, the trace cache delivers approximately 10.5 instructions per cycle, about two fetch blocks. Figure 4 is a histogram in which instruction fetches on the

¹Vortex and go were simulated with abbreviated versions of the SPECint95 test input set. Compress was simulated on a modified version of the test input with an initial list of 30000 elements.

correct execution path are categorized by size. The data histogram was collected on the baseline configuration running the benchmark gcc. The conditions which limit the size of each fetch and their frequencies are identified on the graph.

There are seven major conditions which limit a fetch:

1. **Partial Match.** The path predicted by the branch predictor differed from the path of the trace segment and only a portion of the segment was subsequently issued.
2. **Atomic Block.** The fill unit was forced to create a segment smaller than maximum size because the subsequent block in the retire stream was larger than the space remaining in the pending segment. Here the fill unit is not using the trace packing technique and is treating fetch blocks atomically.
3. **ICache.** The fetch was serviced by the icache and it was terminated by a control instruction or a cache line boundary² before 16 instructions were fetched.
4. **Mispred BR.** A mispredicted branch terminated the fetch. All instructions within the segment after the branch issued inactively also contribute to the current fetch.
5. **Max Size.** The trace segment or icache fetch contained 16 instructions.
6. **Ret, Indir, Trap.** Returns, indirect jumps, and traps cause the pending segment to be finalized.
7. **Maximum BRs.** The fill unit created a segment containing three branches and all three were on path and issued actively from the current fetch.

Figure 4 shows that a large number of fetches reach the maximum branch limit before a full segment of 16 instructions is collected.

In order to address this limitation in an effective manner, we draw upon a frequently reported [3] characteristic of conditional branches: during execution, over 50% of conditional branches are strongly biased. When such a branch is detected, the branch will be converted by the fill unit into a branch with a built-in static prediction. We call this process branch promotion. The concept is similar to branch filtering proposed by Chang et al [2]. A promoted branch requires no dynamic prediction and therefore need not consume branch predictor bandwidth when it is fetched. Its likely target instruction is either included within the trace segment or will be fetched in the subsequent fetch cycle. Two types of promoted conditional branches can be dynamically created —

²The instruction cache implements split line fetching, however cache line boundaries can still terminate a fetch if the request for the second line misses in the cache.

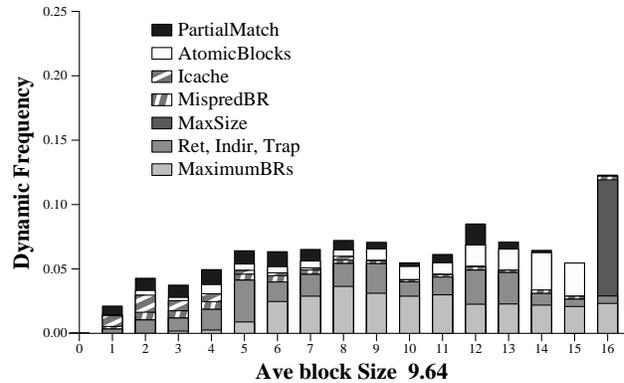


Figure 4. The fetch width breakdown for gcc with a baseline 128KB trace cache.

ones that are strongly biased towards not taken and ones strongly biased towards taken. With two bits, the fill unit can encode a branch as promoted and designate its likely outcome.

Candidate branches can be detected via a hardware mechanism similar to the mechanism used for branch filtering [2]. A table, indexed by branch address, called the branch bias table, is shown in figure 5. It contains the previous outcome of the branch and the number of consecutive times the branch has had that same outcome. The bias table is updated whenever a branch is retired. The fill unit indexes the bias table whenever a conditional branch is added to the pending segment, which in our case is also at retire time. If the number of consecutive outcomes of the branch is beyond a *threshold*, the branch is promoted. For the experiments involving promoted branches presented here, the size of the bias table was fixed at 8K entries. We model a tagged bias table.

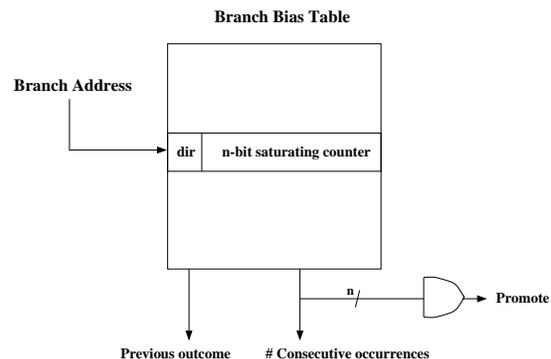


Figure 5. Diagram of the branch bias table.

A promoted branch that is mispredicted is said to fault and can be handled like any ordinary instruction which has an exception during execution. A promoted branch which

faults is demoted back to a normal conditional branch if there are two or more consecutive outcomes in the other direction or if there is a miss in the bias table. The rationale is to inhibit the final iteration of a loop branch from demoting an otherwise strongly biased branch.

Table 2 shows the average effective fetch rate for branch promotion with various values for the *threshold*. Included are the effective fetch rates delivered by the icache and the baseline configurations described in section 3. For a threshold value of 64, the effective fetch rate is increased by an average of 7% over the baseline.

Configuration	Ave effective fetch rate
icache	5.11
baseline	10.67
threshold = 8	11.35
threshold = 16	11.38
threshold = 32	11.39
threshold = 64	11.40
threshold = 128	11.35
threshold = 256	11.33

Table 2. The average effective fetch rate with and without branch promotion.

With branch promotion, much fewer fetches are limited by branch predictor bandwidth. Figure 6 is similar to figure 4 and shows a histogram of fetch sizes on the benchmark gcc annotated with reasons for fetch termination. The threshold for branch promotion is 64 consecutive occurrences. Compared to figure 4, there are fewer fetches terminated because of the maximum branch limit. For this benchmark, the effective fetch rate is 10.24 instructions per cycle, a 6% increase over the baseline.

In addition to increasing the effective fetch rate of the trace cache, branch promotion removes easily predictable branches from the domain of the dynamic predictor. Since these branches do not update the predictor’s pattern history table, interference [16] is reduced in a manner similar to branch filtering. Their outcomes, however, are added to the global branch history to maintain the integrity of the predictor’s information. Because interference is reduced, prediction accuracy improves overall. Figure 7 shows the percent change in the number of conditional branches on the correct execution path which are mispredicted for three configurations (threshold 64, 128, and 256) compared to the baseline. In most configurations, the number of mispredictions is reduced, in some cases significantly. For gcc and go, promotion at threshold=64 reduces the number of mispredicted conditional branches to about 80% of the baseline. Overall, the branch misprediction rate drops from 8% on the baseline

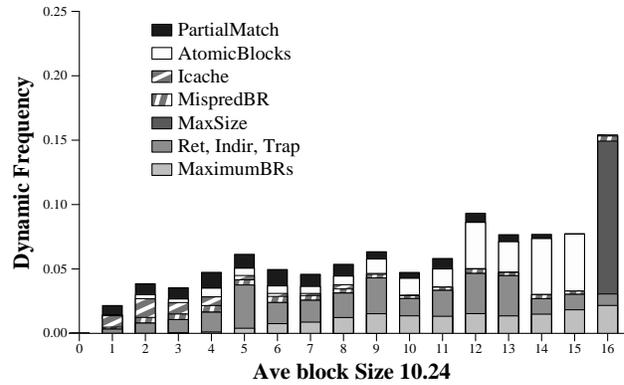


Figure 6. The fetch width breakdown for gcc with a 128KB trace cache with branch promotion.

to 7% for threshold=64. However, premature promotion can lead to frequent faulting (faults also count as mispredictions), as is the case with plot. Increasing the threshold for promotion reduces the effect of premature promotion, as branches that pass the larger threshold are more likely to remain biased.

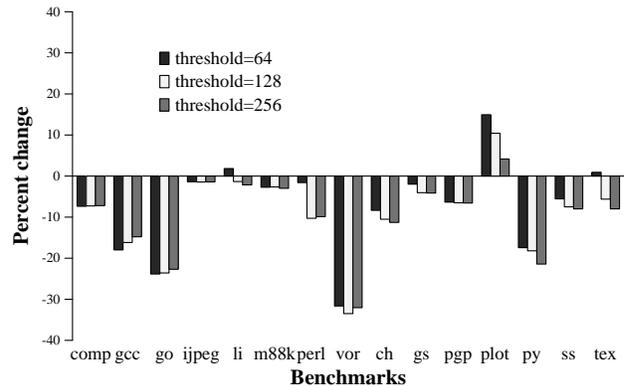


Figure 7. The percent change, relative to the baseline, in the number of mispredicted branches when branches are promoted.

Branch promotion effectively enlarges the execution atomic unit of instructions from the perspective of the trace cache. All instructions within such an atomic unit are guaranteed to either all execute or all not execute. Conditional branches terminate an execution atomic unit whereas promoted branches need not. Similar to the handling of an exceptional instruction, if a promoted branch faults, the machines is backed up to the previous checkpoint (the end of the previous block) and machine resumes with the promoted branch executing in the correct direction.

Another result of promotion is that fewer non-promoted

branches are encountered per fetch of 16 instructions. Previous trace cache studies measured that at least three branches were required to effectively deliver bandwidth for a 16-wide machine. The data in table 3 shows that with promotion on average 80% of all fetches require only one dynamic branch prediction.

This has strong implications for the branch predictor used to sequence through trace segments. The first prediction is the most essential since the majority of fetches will require only it and no more. With promotion, in the average case, a trace segment has fewer successors, thus the predictor needs to select between fewer outcomes. If the trace cache is being used for, say, an 8-wide machine, then promotion opens the possibility of using aggressive hybrid single branch prediction with the trace cache. Such a technique would allow for a high effective fetch rate with highly accurate branch prediction.

Configuration	0 or 1 predictions	2 predictions	3 predictions
baseline	54%	18%	28%
threshold = 64	85%	12%	3%

Table 3. The number of predictions required each fetch cycle, averaged over all benchmarks.

The multiple branch predictor used in the baseline is not well utilized if branches are promoted. Most of trace segments will use only the first counter, leaving the remaining 6 unused. To adjust for this, we restructured the pattern history table into three separated tables, each producing a prediction. The first table contains 64K 2-bit counters and provides the prediction for the first branch. The second table contains 16K 2-bit counters and provides the prediction for the second branch. The third table contains 8K 2-bit counters and provides the prediction for the third branch. The total cost of this predictor is 24KB. With an 8KB bias table, the storage cost of branch promotion is roughly the same as the baseline.

Branch promotion can be done statically, as well. The ISA must allow for extra encodings to communicate strongly biased branches to the hardware. However, branches which switch outcomes during execution but remain biased or are sensitive to input data may be missed during static analysis. There are a few advantages: branches need not go through a warm-up phase before being detected as promotable and branches which have irregular behavior but are strongly biased can be more easily detected statically.

5. Trace Packing

Branch promotion results in a 7% percent increase in overall effective fetch rate. However from the fetch termination histogram in figure 6, it is evident that by promoting branches, we are increasing fetch bandwidth, only to be limited by the treatment of fetch blocks as atomic entities within the fill unit. With atomic treatment of fetch blocks, the fill unit will not divide a block of newly retired instructions across trace segments (unless the block is larger than 16 instructions). If the pending trace segment contains 13 instructions, then a block of 9 instructions will cause the segment of 13 instructions to be written and the block of 9 will begin a new segment.

There is a strong rationale for treating fetch blocks as atomic entities. The instruction may be stored in multiple locations in the trace cache. Figure 8 shows a loop composed of three fetch blocks. If blocks are treated atomically, three trace segments containing the loop blocks are formed in the steady state: AB, CA, BC. But if the fill unit is allowed to fragment a block, a process we call trace packing, then eleven segments could be created. The problem gets significantly worse if there are different control paths within the loop.

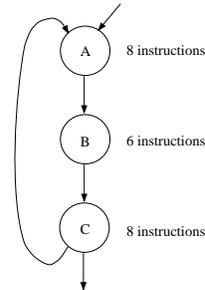


Figure 8. A loop composed of 3 fetch blocks.

On the positive side, the redundant storage can potentially increase the delivered fetch rate, e.g. loops will be dynamically unrolled so that a maximum number of blocks can be fetched per cycle. But the primary cost of this redundancy is increased contention for trace cache lines. With non-atomic treatment of blocks, trace segments can be packed with more instructions, but at the cost of increased redundancy within the trace cache.

Figure 9 shows the effective fetch rate for the baseline configuration with and without trace packing. The increase in effective fetch rate over the baseline is on average 7%. The cost of this slight boost in effective fetch rate is an increase in cache misses.

Both branch promotion and trace packing are limited by the problem the other is solving. The key to unlocking the fetch potential of the trace cache is to use both techniques.

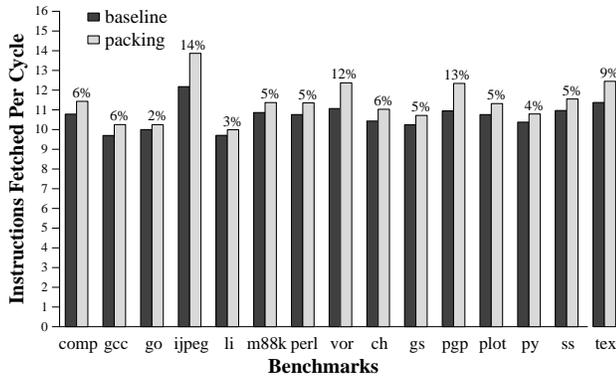


Figure 9. The effective fetch rates with and without trace packing.

Figure 10 compares the effective fetch rate for 5 configurations: the icache, the baseline trace cache, a trace cache with promoted branches, a trace cache with trace packing, and a trace cache using both promoted branches and trace packing. The threshold for branch promotion is set to 64. With both techniques in place, the fetch bandwidth is boosted by an average of 17% across all the benchmarks simulated. The percentages above the bars indicate the increase with both packing and promotion over the baseline. In many cases, the total increase is more than the sum of the individual increases, as is the case with the benchmarks gcc, chess, gnuplot, and simplescalar, and on the overall average.

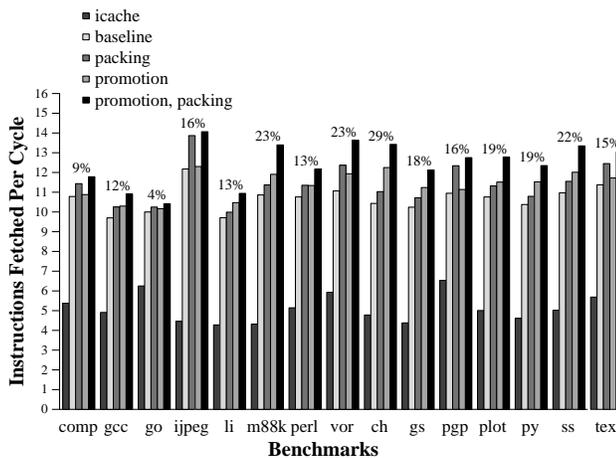


Figure 10. The effective fetch rates for all techniques.

As mentioned, a serious drawback to trace packing is the effect the block fragmentation has on trace cache contention. The increased cache contention problem stems from the fact that trace segments now can begin at any in-

struction. In configurations where blocks are treated atomically, trace segments are naturally *synchronized* at fetch block boundaries. Either an entire block is replicated or nothing from that block is replicated. With trace packing, trace segments are no longer synchronized at fetch block boundaries and the instruction replication grows significantly. To deal with this problem, and the negative effect on the prediction mechanism, we have examined a scheme where the fill unit only packs chunks of n instructions. For example, if $n=2$ and the entire block doesn't fit, then only an even number of instructions is added to the pending segment. Blocks now fragment at half the number places as compared to unregulated trace packing.

Another scheme we have investigated only packs traces if the number of unused instruction slots is more than or equal to half the number of instructions in the pending segment OR the pending segment contains a backwards branch with displacement of 32 or fewer instructions. Firstly, trace packing is most important when the number of unused instructions slots in a trace segment is large. Not only will valuable space on the cache line go unused when such a segment is stored in the cache, but whenever this segment is fetched, fetch bandwidth will be wasted. By only packing a trace when the amount of unused space is high, we only incur the fragmentation costs when the potential payoff is high. Secondly, the potential payoff is also higher for small tight loops. Since unrolling may significantly boost fetch bandwidth for a small loop, the benefit may overcome the costs of fragmentation, particularly if the number of iterations is high. This scheme is referred to as cost-regulated trace packing.

The percentage change in cache misses for unregulated trace packing, regulating at every other instruction ($n=2$) and every fourth instruction ($n=4$) and the cost-regulated scheme is provided in table 4 for the six benchmarks which suffer from a significant number of cache misses. The percentage increase is over a trace cache which performs promotion at threshold = 64. For all packing schemes, branch promotion is also performed at threshold = 64.

Benchmark	unreg	cost-reg	n = 2	n = 4
gcc	26.9%	13.2%	22.3%	15.8%
go	28.4%	11.6%	23.9%	15.9%
vortex	18.1%	15.0%	11.1%	4.5%
gs	29.5%	16.2%	22.8%	14.1%
python	38.9%	1.5%	18.2%	13.0%
tex	95.6%	39.5%	74.6%	52.8%
Ave Eff Fetch Rate	12.47	12.23	12.42	12.18

Table 4. Percent increase in cache miss cycles of packing over the promotion configuration.

The techniques for regulating the redundancy within the trace cache are crucial if the size of the fetch mechanism is smaller than 128KB. We model a fetch mechanism of 128KB and simulate on benchmarks which may not be representative, in terms of size, to other common applications. The lost fetch bandwidth due to cache misses is fairly minor. If the fetch mechanism is smaller, such techniques to regulate redundancy may be necessary to realize performance.

6. Implications on performance

Figure 11 shows the overall performance, in instructions retired per cycle (IPC), of the icache configuration, the baseline configuration, and a configuration which both promotes branches (threshold = 64) and performs cost-regulated trace packing (as described in section 5). The performance improvement of the new configuration over the baseline for each benchmark is indicated on the graph. Overall, with branch promotion and trace packing, performance increases by 4% over the baseline model and 36% over the icache model. On the benchmark gcc, the performance improves by 7% over the baseline and 29% over the icache.

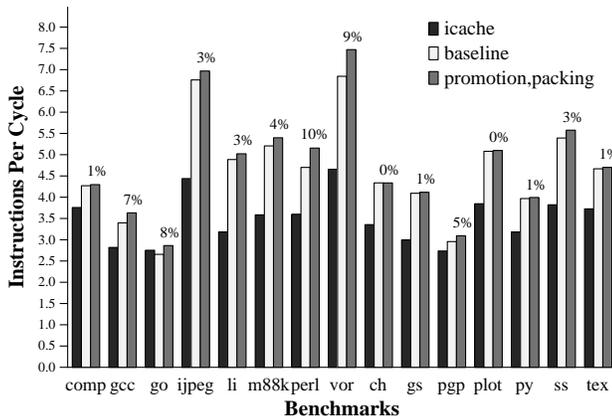


Figure 11. The overall performance of promotion and trace packing, compared to the icache and baseline configurations

The large increase in effective fetch rate and modest reduction in branch mispredictions translates into a minor increase in overall performance. Figure 12 identifies where fetch cycles are being spent for the configuration using the new techniques. We classify each fetch cycle into one of six categories:

1. fetch cycles that result in instructions on the correct execution path (Useful Fetch),
2. fetch cycles that produce instructions off the correct execution path (Branch Misses),

3. fetch cycles that produce no instructions because of a cache miss (Cache Misses),
4. cycles where the fetch mechanism is stalled due to a full instruction window (Full Window),
5. cycles where the fetch mechanism was stalled due to trap instructions (Traps), and
6. fetch cycles where the wrong fetch address was generated (Misfetches).

For all but one of the benchmarks (vortex), most of the lost fetch bandwidth is a result of branch mispredictions.

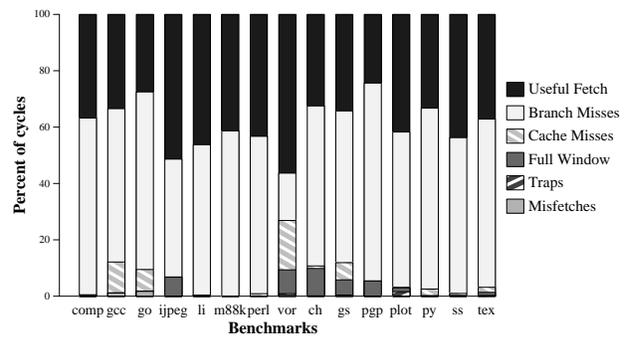


Figure 12. An accounting of all fetch cycles for a configuration with promotion and packing.

Figure 13 shows the percent increase in the number of cycles lost due to branch mispredictions between the baseline and a model which promotes branches and uses cost-regulated trace packing. Most benchmarks suffer from an increase in lost bandwidth due to branch mispredictions.

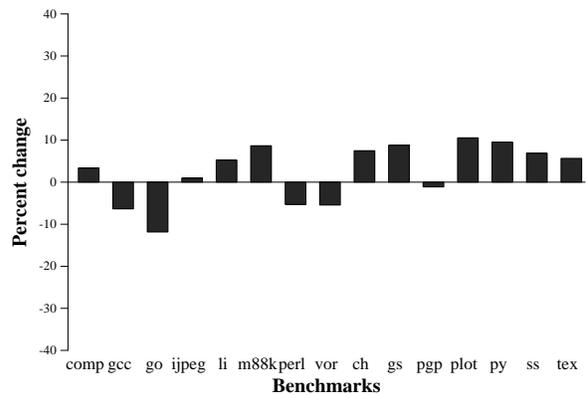


Figure 13. The percent change, relative to the baseline, in the number of fetch cycles lost due to mispredictions.

Figure 14 shows the percent change in the number of branches mispredicted (both conditional and indirect. Returns are predicted ideally.) The decrease in mispredictions is due to the reduction in pattern history table interference from branch promotion. Despite a decrease in mispredictions, the benchmarks compress, jpeg, m88ksim, gnuchess, python, and simplescalar still suffer from an increase in lost cycles due to mispredictions.

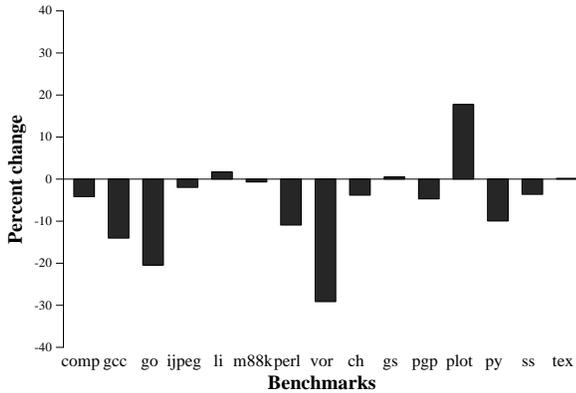


Figure 14. The percent change, relative to the baseline, in the number of mispredicted branches.

The number of cycles lost due to branch mispredictions is a product of the number of branches mispredicted and the average number of cycles it takes to resolve each of these mispredicted branches. Mispredicted branch resolution time is the number of cycles between when a branch is predicted and when it is detected that the branch was mispredicted and a new fetch address generated. Figure 15 shows the percent increase in mispredicted branch resolution time for the two configurations on each benchmark. The average increase is 8%. As branches are fetched earlier, they must wait longer in the instruction window before their source operands are ready and execution resources are available. Therefore their resolution time increases. The average resolution times for the benchmark plot decreases because the number of mispredicted branches is significantly higher. Branches that are fetched after a misprediction recovery are more likely to find ready operands, and therefore do not wait as long in the instruction window.

The increase in resolution time indicates that the execution engine is consuming instructions at a slower rate than the fetch engine is delivering them. Any inefficiencies in the execution engine which introduce artificial stalls (such as false dependencies or functional unit limitations) will have a greater negative effect since more of the computation needed to resolve a branch must be done after the branch is fetched and predicted. The increase indicates that more aggressive and speculative techniques need to be used

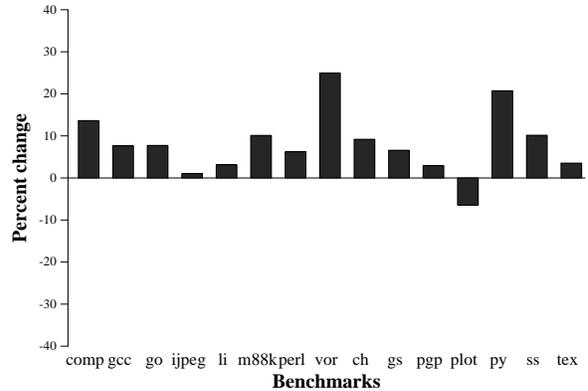


Figure 15. The percent change, relative to the baseline, in the number of cycles to resolve a mispredicted branch.

in the execution engine to consume the bandwidth now being supplied by this aggressive front end. Techniques which expedite memory operations via aggressive memory disambiguation, for example memory dependence speculation [8] or memory renaming [15], are more critical.

In order to evaluate these effects, we performed experiments with a more aggressive, less restrictive execution engine. Figure 16 shows the performance of the icache, the baseline trace cache and the trace cache with promotion and packing.

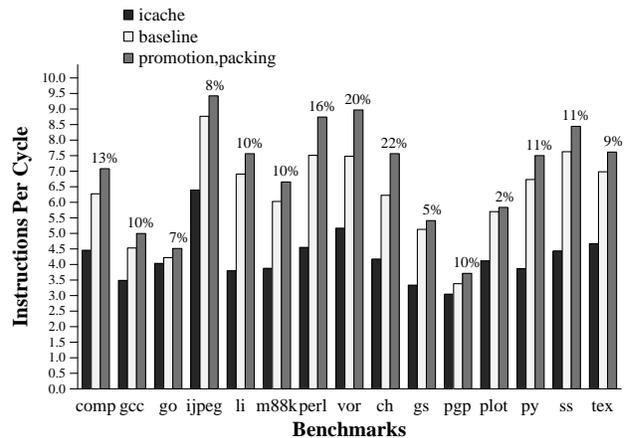


Figure 16. The overall performance, given an ideal, aggressive execution engine.

The percent increases in performance of the new configuration over the baseline for each benchmark is denoted on the graph. All three models schedule memory operations optimally (ie., all dependencies between loads and stores are speculated correctly). The overall performance improvement is 11% over the enhanced baseline. The gain

over the enhanced icache model is 63%. For high performance, it is important not only to predict branch accurately, but also to resolve them quickly.

7. Conclusions

We have described two techniques for the trace cache mechanism, branch promotion and trace packing, which when used together boost the effective fetch rate by 17% over a trace cache which uses neither. Branch promotion boosts the fetch rate by removing branches from the domain of the dynamic predictor. As a result, branch predictor bandwidth plays a lesser role in dictating how many instructions can be packed into a trace cache line. Promotion also reduces the interference in the branch predictor, increasing its effectiveness on the branches which still require dynamic prediction. Because it reduces the required branch prediction bandwidth, promotion is a very attractive option for near-term high performance designs. For instance, promotion opens the possibility of using aggressive single hybrid branch prediction with the trace cache for an 8-wide fetch engine. Trace packing boosts the fetch rate by packing as many instructions into a trace segment as will fit.

Even with a large improvement in fetch rate and a decrease in the number of branches which are mispredicted, a trace cache which uses both schemes only realizes a 4% improvement in performance. Promotion and packing lose performance potential due to an increase in branch resolution time. This increase is evidence that a bottleneck in the execution core is strongly affecting performance. If the execution core is made more aggressive, with highly aggressive memory disambiguation, then more of the potential of branch promotion and trace packing can be realized. A 11% increase in performance due to branch promotion and trace packing results with an execution core which performs ideal memory scheduling.

References

- [1] 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.
- [2] P.-Y. Chang, M. Evers, and Y. N. Patt. Improving branch prediction accuracy by reducing pattern history table interference. In *Proceedings of the 1996 ACM/IEEE Conference on Parallel Architectures and Compilation Techniques*, 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*, pages 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*, pages 162–171, 1994.
- [5] D. H. Friendly, S. J. Patel, and Y. N. Patt. Alternative fetch and issue techniques from the trace cache fetch mechanism. In *Proceedings of the 30th Annual ACM/IEEE International Symposium on Microarchitecture*, 1997.
- [6] 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*, pages 18–26, 1987.
- [7] 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*, pages 60–63, 1988.
- [8] A. Moshovos, S. E. Breach, T. N. Vijaykumar, and G. S. Sohi. Dynamic speculation and synchronization of data dependences. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, 1997.
- [9] 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.
- [10] A. Peleg and U. Weiser. Dynamic flow instruction cache memory organized around trace segments independent of virtual address line. U.S. Patent Number 5,381,533, 1994.
- [11] 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*, 1996.
- [12] E. Rotenberg, Q. Jacobsen, Y. Sazeides, and J. E. Smith. Trace processors. In *Proceedings of the 30th Annual ACM/IEEE International Symposium on Microarchitecture*, 1997.
- [13] 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*, pages 219–229, 1995.
- [14] J. Stark, P. Racunas, and Y. N. Patt. Reducing the performance impact of instruction cache misses by writing instructions into the reservation stations out-of-order. In *Proceedings of the 30th Annual ACM/IEEE International Symposium on Microarchitecture*, pages 34–43, 1997.
- [15] G. S. Tyson and T. M. Austin. Improving the accuracy and performance of memory communication through renaming. In *Proceedings of the 30th Annual ACM/IEEE International Symposium on Microarchitecture*, 1994.
- [16] C. Young, N. Gloy, and M. D. Smith. A comparative analysis of schemes for correlated branch prediction. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 276–286, 1995.