Evaluation of Design Options for the Trace Cache Fetch Mechanism

Sanjay Jeram Patel, Daniel Holmes Friendly, and
Yale N. Patt Fellow, IEEE

Advanced Computer Architecture Laboratory
Department of Electrical Engineering and Computer Science
The University of Michigan
Ann Arbor, MI 48109-2122
{sanjayp, ites, patt}@eecs.umich.edu
Tel: (313) 936-0404
Fax: (313) 763-4617

ABSTRACT

In this paper, we examine some critical design features of a trace cache fetch engine for a 16-wide issue processor and evaluate their effects on performance. We evaluate path associativity, partial matching, and inactive issue, all of which are straightforward extensions to the trace cache. We examine features such as the fill unit and branch predictor design. In our final analysis, we show that the trace cache mechanism attains a 28% performance improvement over an aggressive single block fetch mechanism and a 15% improvement over a sequential multi-block mechanism.

Keywords: high bandwidth fetch mechanisms, trace cache, instruction cache, wide issue machines, speculative execution

1 Introduction

A microprocessor is composed of three fundamental components: a means to supply instructions, a means to supply the data needed by these instructions, and a means to process these instructions. For high performance, the instructions and data must be delivered at high rate to a processing core capable of effectively consuming them.

Delivering instructions at a high rate is not a straightforward task. Several factors degrade fetch engine performance. First, cache misses cause the fetch engine to stall and supply nothing until the miss is resolved. Second, branch mispredictions cause the fetch engine to supply instructions which will later be discarded. Third, changes in control flow inhibit the fetch engine from producing a full width of instructions. Due to the physical structure of instruction caches, it is difficult to fetch both a taken branch and its target in the same cycle.

Table 1 shows the average number of instructions between all branches and between taken branches (i.e., taken conditional branches, jumps, subroutine calls, returns and traps) for the SPECint95 benchmarks. Evident from this table is that if partial fetches due to branches are not dealt with, then fetch bandwidth will be limited to an average limit of 9 instructions per cycle.

	cmprs	gcc	go	ijpeg	li	m88k	perl	vrtx	avg
insts per branch	5.57	5.05	6.63	10.40	4.27	4.35	5.21	6.41	5.99
insts per taken branch	7.78	8.32	9.85	13.51	6.55	5.28	7.94	9.85	8.64

Table 1: Instruction run lengths when terminating fetches on branches.

A straightforward technique for dealing with partial fetches is to construct an instruction cache where multiple independent fetches can be performed each cycle. To do this, multiple addresses are used to index into the instruction cache via multiple read ports. After the cache lines are read, the requested instructions must be properly aligned and merged before they are supplied for execution. Such a solution adds considerable logic complexity in an already critical execution path of the processor. Either cycle time will be affected or extra pipeline stages will be required.

Recently proposed, the trace cache [20, 11, 22, 19] overcomes this bandwidth hurdle without requiring excessive logic complexity in the instruction delivery path. Like an instruction cache, the trace cache is accessed using the Program Counter. Unlike an instruction cache,

a trace cache line contains instructions as they appear in execution order, as opposed to the static order determined by the compiler. Two adjacent instructions in a trace cache line need not be adjacent in the executable image. A trace cache line stores a *segment* of the dynamic instruction stream. By placing logically contiguous instructions in physically contiguous storage, the trace cache is able to supply multiple fetch blocks each cycle. 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.

In addition to increasing the delivered instruction bandwidth by increasing the effective fetch rate (the average number of correct instructions issued for each fetch that returns on-path instructions), caching instructions as trace segments makes reprocessing them easier. Since instructions are written into the trace cache after being decoded, decode bits are stored along with each instruction minimizing the processing required when it is fetched again.

In this paper, we evaluate several extensions of the trace cache such as partial matching, path associativity, and inactive issue. We also examine some of the key components of the trace cache fetch mechanism such as the fill unit and multiple branch predictor.

2 Related Work

Cache organizations for simultaneously fetching multiple blocks have been studied by Yeh et al [28], Conte et al [4] and Seznec et al [24]. By multiporting the instruction cache and/or the branch target buffer (BTB) and generating multiple fetch addresses and branch predictions per cycle, these schemes are able to overcome the single fetch block bottleneck.

There are several compile-time approaches to solving the fetch bottleneck problem. With Block-Structured ISAs [14, 8], superblock scheduling [9] and trace scheduling [5, 3], the static form of the program is organized by the compiler into longer sequential units composed of

multiple basic blocks.

Melvin and Patt [15] discuss the performance implications of the fill unit and the idea of dynamically combining fetch blocks into larger "execution atomic units" (EAUs) to further increase the fetch bandwidth is first proposed. In 1994, Peleg and Weiser filed a patent on the trace cache concept [20]. A similar concept was proposed by Johnson [11]. The concept was further investigated by Rotenberg et al [22]. 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. Extensions and analysis of the trace cache mechanism were proposed by Patel et al [19, 6, 18] and trace cache implications on processor design were presented in [23]. A similar approach to caching dynamic instruction groups was presented in the DIF cache by Nair and Hopkins [17].

3 The Trace Cache Fetch Mechanism

We divide the trace cache fetch mechanism into four major components: a trace cache, a fill unit, a multiple branch predictor, and a conventional instruction cache. The trace cache is the main source of instruction supply and is filled with trace segments by the fill unit. The speculative sequencing of segments is performed by the multiple branch predictor. The instruction cache plays an important but supporting role, handling cases when the required instructions are not found in the trace cache. A high-level diagram of the mechanism is shown in figure 1.

In this section, we provide the details of a fetch engine for a 16-wide issue dynamically-scheduled machine. To meet the instruction bandwidth demands of this machine, 16 instructions and three conditional branches can be fetched per cycle.

3.1 The Trace Cache

The trace cache stores segments of the dynamic instruction stream, exploiting the fact that many branches tend to favor one outcome. If block A is followed by block B which in turn is followed by block C at a particular point in the execution of a program, there is a strong likelihood that they will be executed in that order again. After the first time they are executed in this order, they are stored in the trace cache as a single entry. Subsequent fetches of block A from the trace cache provide blocks B and C as well.

Figure 2 shows an example of a trace cache fetch cycle. A request is made with the address of block A. The trace cache responds with a hit and drives out the selected segment composed of the blocks A, B, and C. The prediction structures are accessed concurrently with the trace cache. At the end of the cycle, the segment is matched with the prediction. Suppose the predictor selects the path ABD. Then only blocks A and B are supplied. Block D is requested in the following cycle. The concept of supplying only the matching portion of a trace cache line is called partial matching and its implications will be examined in section 5.1.

The trace cache can store segments containing up to 16 instructions, 3 of which may be conditional branches. The line is accessed by the address of the first instruction in the segment. The organization of the trace cache is similar to that of a conventional instruction cache, as the lines may be arranged in an associative manner. A hit is determined by a tag match.

Each line of the trace cache contains:

- 16 slots for instructions. Instructions are stored in decoded form and occupy approximately five bytes for a typical ISA. Up to three branches can be stored per line. Each instruction is marked with a two-bit tag indicating to which block it belongs.

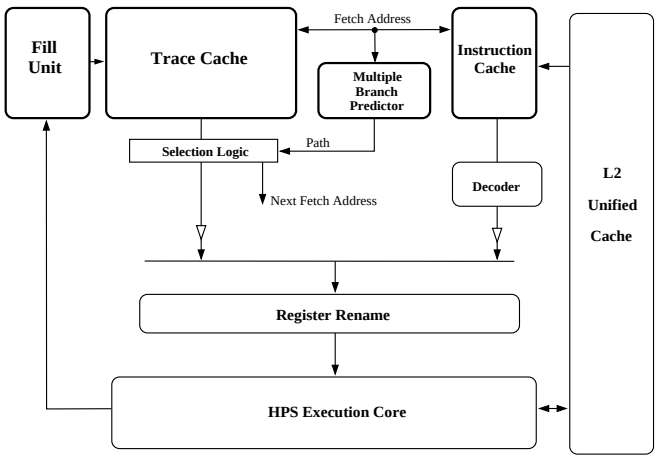

Figure 1: The trace cache fetch mechanism.

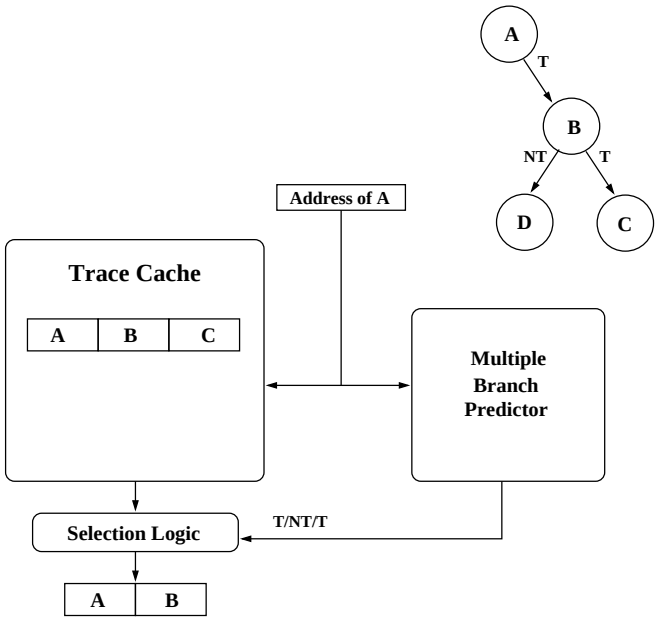

Figure 2: The trace cache and branch predictor are indexed with the address of block A. The inset figure shows the control flow from block A. The predictor selects path ABD. The trace cache only contains ABC. AB is supplied.

- Four target addresses. With three basic blocks per segment and the ability to fetch partial segments, there are four possible targets to a segment. The four addresses are explicitly stored allowing immediate generation of the next fetch address, even for cases where only a partial segment matches.

- Path information. This field encodes the number and directions of branches in the segment and includes bits to identify whether a segment ends in a branch and whether that branch is a return from subroutine instruction. In the case of a return instruction, the return address stack provides the next fetch address.

The total size of a line is around 97 bytes for a typical architecture: 5x16 bytes of instructions, 4x4 bytes of target addresses, and 1 byte of path information.

Instruction dependencies within a trace segment are predetermined before the segment is stored in the trace cache. This additional information allows for minimal decoding when the segment is fetched from the trace cache. Source operands produced by an instruction outside the segment are explicitly marked as requiring an external value. Instructions which produce a live-out value are marked as requiring a physical register. It is important to note that a complex dependency analysis across 16 instructions does not need to be performed on segments fetched from the trace cache. The concept of explicitly marking internal/external register values within a basic block was first described by Sprangle and Patt [25] and later adapted for use with the trace cache by Vajapeyam and Mitra [27].

Instructions within a segment can be arranged in an order that permits quick issue. Because the dependencies within a segment are explicitly marked, the ordering of instructions carries no significance. Instructions within the cache line can be arranged to mitigate the routing required to forward instructions to functional unit reservation stations. Friendly et al [7] examine a scheme where instructions within a segment are ordered to reduce the communication delays associated with data forwarding across many functional units.

3.2 The Fill Unit

The fill unit collects instructions as they are issued by the processor and combines them into trace segments. Conceptually, the instructions (in units of blocks) are latched by the fill unit in the order they were fetched. The fill unit merges the arriving blocks with blocks latched in previous cycles. The merge process involves creating dependency information and reordering instructions. The process continues until the segment becomes *finalized*, at which point the segment is written into the trace cache.

A segment becomes finalized when

1. it contains 16 instructions, or
2. it contains 3 conditional branches, or
3. it contains a single indirect jump, return, or trap instruction, or
4. merging the incoming block would result in a segment larger than 16 instructions.

Rule 1 is implied by the size of the trace cache line and rule 2 by the number of predictions supplied per cycle by the predictor. Because their targets vary, return instructions and indirect jumps cause finalization (rule 3). Unconditional branches and subroutine calls do not affect trace segment finalization.

Because blocks are being combined in a greedy fashion, there are cases where the fill unit stores multiple copies of basic blocks in the trace cache. Figure 3 shows a simple loop composed of five basic blocks. The fill unit can potentially create five different trace segments containing portions of this loop, all of which can be simultaneously resident in the trace cache. This block redundancy may degrade performance of the trace cache mechanism by displacing useful lines with redundant information. The tradeoff is between higher bandwidth from fetching larger segments versus lost bandwidth due to increased misses in the trace cache.

The problem arises because of blocks where several execution paths merge, such as block B in figure 3.

Rule 4 above treats basic blocks as atomic entities. A basic block is not split across two segments unless the block is larger than 16 instructions. In addition to exacerbating the block redundancy problem, splitting a basic block creates an additional block that may tax other structures such as the branch predictor. A study of effective techniques for splitting blocks was done by Patel et al [18].

Three outcomes are possible with the arrival of each new block of instructions: (1) the arriving block is merged with the unfinalized segment and the new, larger segment is not finalized. (2) the entire arriving block cannot be merged with the awaiting segment. The awaiting segment is finalized and the arriving block now occupies the fill unit. (3) the arriving block is completely merged with the awaiting segment and the new, larger segment is finalized.

3.3 The Branch Predictor

The branch predictor is a critical component in a high bandwidth fetch mechanism. To maintain a high rate of instruction supply, the predictor needs to make multiple accurate branch predictions per cycle. In the case of our trace cache mechanism, three predictions per cycle are required.

Two level adaptive branch prediction has been demonstrated to achieve high prediction accuracy over a wide set of applications [29]. In a two level scheme, the first level of history records the outcomes of the most recently executed branches. The second level of history, stored in the pattern history table (PHT), records the most likely outcome when a particular pattern in the first level history is encountered. In typical schemes, the PHT consists of saturating two-bit counters.

To make three predictions per cycle, we expand each PHT entry from a single two-bit counter into three two-bit counters, with each two-bit counter providing a prediction for a fetched branch. Evaluation of this PHT entry organization versus several others is provided in section 5.5.

The baseline configuration for the trace cache predictor uses the gshare scheme outlined by McFarling [13]. The global branch history is XORed with the current fetch address, forming an index into the PHT. This hashing better utilizes the PHT and improves prediction accuracy over other global history based schemes.

The branch predictor also contains a return address stack to predict the target addresses of return instructions.

3.4 The Instruction Cache

A conventional instruction cache supports the trace cache by supplying instructions when the trace cache does not contain the requested segment. Since hitting in the trace cache is the frequently occurring case, the supporting icache need not be enhanced for higher bandwidth. The icache therefore supplies up to one fetch block per cycle.

The instruction cache has two read ports to allow adjacent cache lines to be retrieved each cycle. By fetching two cache lines and realigning instructions, the fetch mechanism overcomes partial fetches due to cache line boundaries.

In the case of a trace cache miss and an instruction cache hit, up to a single basic block is supplied at the end of the cycle. If both the trace cache and instruction cache miss, then a request for the missing instruction cache line is made to the second level cache. The fetch mechanism stalls until the missing line arrives.

4 Experimental Setup

A pipeline simulator that allows the modeling of wrong path effects was used as the experimental model. The simulator was implemented using the SimpleScalar 2.0 tool suite and instruction set [1], which is a superset of the MIPS-IV ISA. In the execution model, all instructions undergo four stages of processing: fetch, issue, schedule, execute. Each stage takes at least one cycle.

The baseline 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 instruction cache. Each trace cache line contains up to 16 instructions, containing at most three conditional branches. To assist in the generation of target addresses for fetches from the icache, a 1KB branch target buffer (BTB) is included. A 1MB unified second level cache provides instruction and data with a latency of 8 cycles in the case of first level cache misses. The L2 miss latency to memory is 50 cycles. The baseline branch predictor modeled is a 15-bit version of the gshare predictor described in section 3.3 and provides up to three individual conditional branch predictions each cycle. The size of the pattern history table is fixed at 32K entries consisting of 3 two-bit counters (24KB of storage). An ideal return address stack is modeled.

The execution engine is composed of 16 functional units, each with a 32-entry reservation station. The functional units are uniform and capable of all operations. A 64KB L1 data cache was used. The model uses checkpoint repair [10] 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. Since this study is concerned with the fetch engine, many components of the execution engine are modeled at very aggressive design points thereby

reducing the potential for the backend to induce bottlenecks which may obscure bottlenecks in the frontend.

All experiments were performed on the SPECint95 benchmarks and on a benchmark suite consisting of several common C applications [26]. Table 2 lists the number of instructions simulated and the input set, if the input was derived from a standard input set[1]. All simulations were run until completion (except li and ijpeg, 500M instructions).

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
chess	119M	
ghostscript (gs)	180M	
pgp	322M	
plot	284M	
python	220M	
sim-outorder (ss)	100M	
tex	164M	

Table 2: Benchmarks and datasets used. All benchmarks, except li and ijpeg, were simulated to completion.

[1]Vortex 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.

5 Critical Issues

In this section, we use the experimental baseline to evaluate several design enhancements and configurations of the trace cache.

5.1 Partial Matching

Each fetch cycle, the predictions made by the branch predictor are used to select which blocks within the accessed segment will be issued to the execution engine. This process is referred to as partial matching [22, 6]. Alternatively, the trace cache can be designed to signal a hit only if *all* the blocks within the selected segment match; otherwise a miss is signaled. Figure 4 shows the performance difference between a trace cache that implements partial matching and one that does not. The average performance improvement for these benchmarks is 14%.

With partial matching, the number of requests that miss both the trace cache and small icache drops significantly because of the more flexible policy for matching valid lines. If the icache were made larger, the relative benefit with partial matching would be expected to diminish.

5.2 Path Associativity

Path associativity relaxes the constraint that different segments starting from the same fetch block cannot be stored in the trace cache at the same time. Path associativity allows segments ABC and ABD (see figure 2) to reside concurrently in the cache whereas a non-path associative trace cache allows only one segment starting at A to be resident in the trace cache at any instance in time.

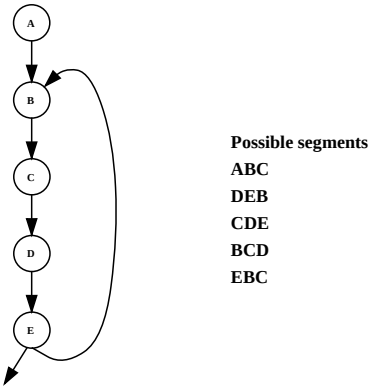

Figure 3: If the fill unit is able to create three-block segments for this path through a loop, then all five possible segments will be created and stored in the trace cache.

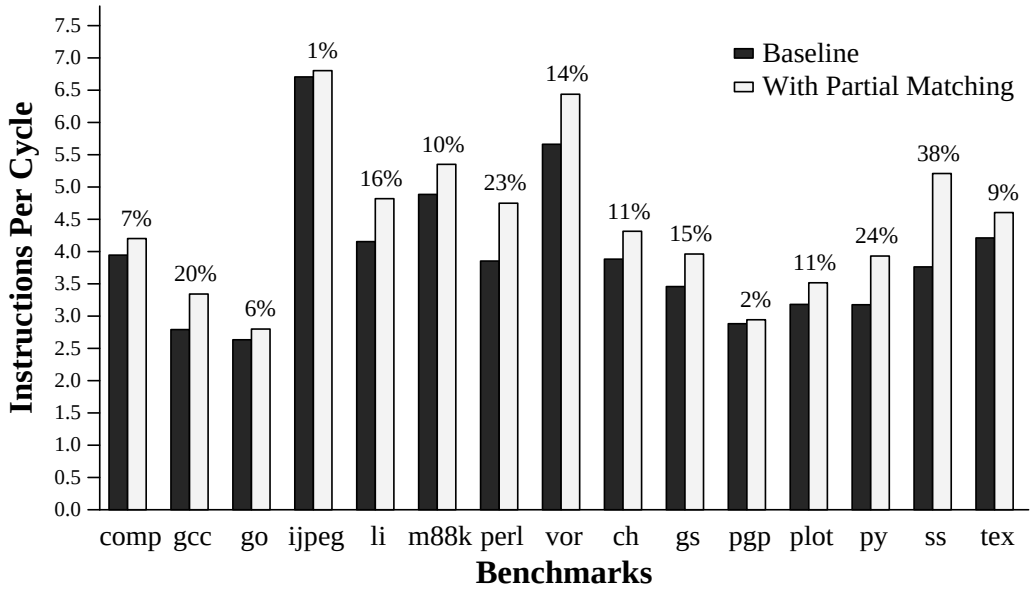

Figure 4: Partial Matching. The performance difference between a trace cache that partially matches segments and one that does not is around 14%

The data plotted in Figure 5 indicate that path associativity has little effect on the performance of the baseline trace cache. Adding path associativity increases the number of segments that map into a particular set; thus additional misses may occur due to increased set conflicts. Therefore the experiment was conducted on a path-associative trace cache with set-associativity of 4 and of 8. In both cases, the performance gain from path-associativity is slight.

Path associativity requires extra gates in the trace selection logic to select the longest matching trace segment. It is likely that the line selection time for the path associative cache will be longer, possibly increasing cache cycle time.

5.3 Inactive Issue

Partial matching increases the number of instructions that are issued each cycle but it does not take advantage of the entire segment of instructions fetched from the trace 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.

With inactive issue [6] all blocks within a trace cache line are issued into the processor core 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 [10] 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.

Inactive issue reduces the impact of branch mispredictions. It allows a portion of the branch resolution latency to be hidden 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.

To implement inactive issue, modifications must be made to the renaming and recovery structures. Our execution 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 target of the newly activated checkpoint. If the branch was correctly predicted, the inactive checkpoints are simply invalidated once its outcome is known.

Figure 6 illustrates the blocks issued from a fetched cache line by the three different policies.

Figure 7 presents the performance benefits of inactive issue. Over the baseline configuration, inactive issue offers a 17% performance boost. However, comparing figure 7 with figure 4, one can notice a slight 3% increase from inactive issue over partial matching. In-

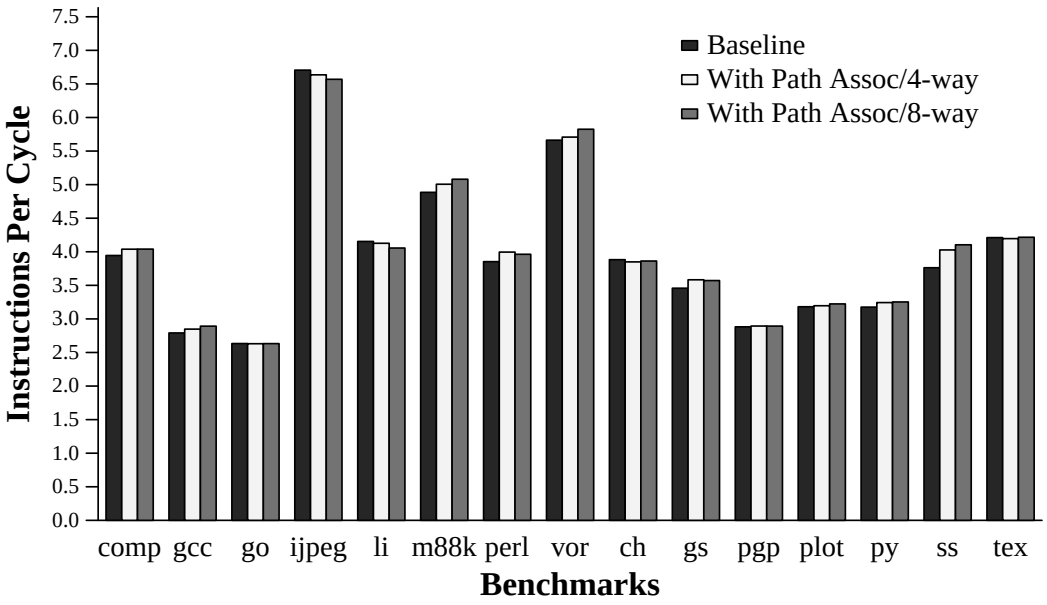

Figure 5: Performance of Path Associativity.

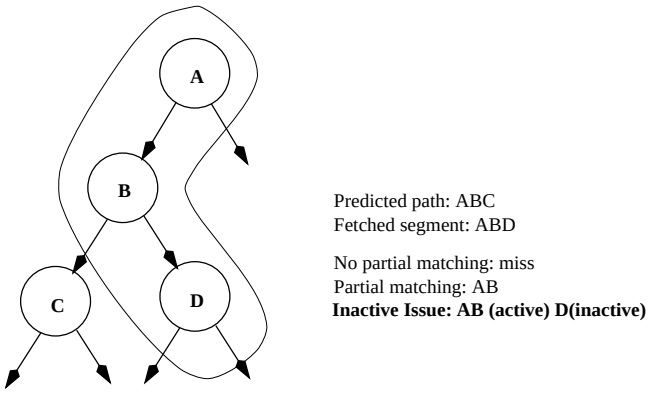

Figure 6: An example of the different issue policies.

active issue is an extension of partial matching, and designed as a hedge against branch mispredictions. The value of inactive issue is greater for less accurate branch prediction. For example, inactive issue is more helpful on programs which are harder on the branch predictor as shown in the boost gained on gcc, go, and pgp.

5.4 Fill Unit Issues

5.4.1 Block Collection

The fill unit collects blocks of instructions as they are processed and produces segments to store into the trace cache. The fill unit can collect these blocks at any point in the processor pipeline. In this experiment, we determine whether the blocks should be collected as instructions are issued into the instruction window or when they are retired.

Figure 8 shows that the differences in performance between the two schemes are slight. For many benchmarks, the speculative segment creation is beneficial, while for some (gcc, go), it degrades performance. The fill unit collecting instructions at issue time provides increased traffic to the trace cache because segments collected while executing on a wrong execution path are also written into the trace cache. In some cases this generates useful segments, but in other cases it evicts useful segments from the trace cache. The baseline configuration collects blocks at retire time.

A fill unit that collects at retire time only writes segments from the correct execution path to the trace cache. However, it suffers from an increased latency between the initial fetch of a block and its collection into a segment and subsequent storage into the trace cache. This can potentially impact the first few iterations of a tight loop, which will be fetched from the instruction cache until the first iteration retires. In the next section we show that this is not a significant influence on performance.

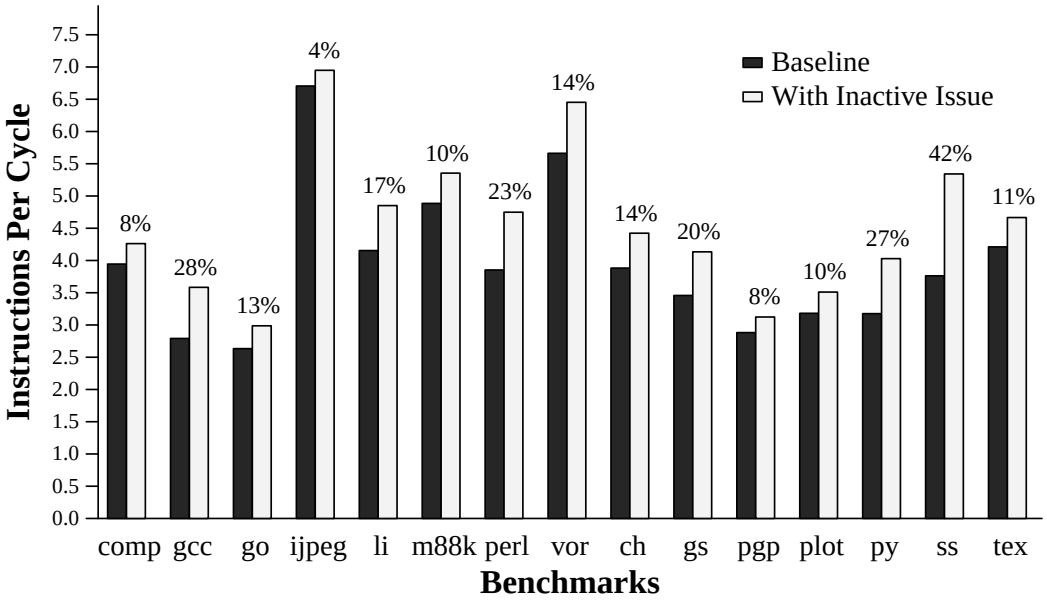

Figure 7: Performance of Inactive Issue.

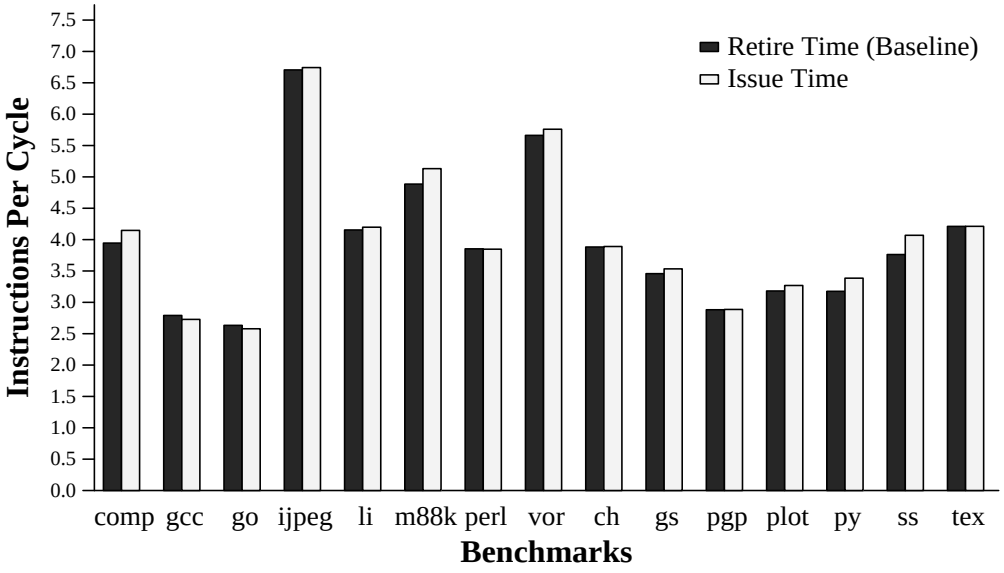

Figure 8: Issue vs. Retire. This plot shows that collecting instructions at issue time is not very different from collecting instructions at retire time.

5.4.2 Fill Unit Latency

As blocks of instructions are latched into the fill unit, some processing is required before the composite segment can be written to the trace cache. The dependencies within the arriving block must be recorded to reflect the values produced by the awaiting blocks. Possibly, the instructions within the segment may need to be reordered so that they can be quickly routed to functional unit node tables when the segment is refetched. To perform these operations, the fill unit may require several cycles.

The purpose of this experiment is to determine the sensitivity of the trace cache fetch mechanism to fill unit latency. Figure 9 shows the results of varying the number of cycles from the arrival of the terminal block to the point it is written into the trace cache.

The results show that a fill unit with a 10-cycle latency has a negligible loss in performance over a single-cycle fill unit. There are two reasons for this counter-intuitive behavior. First, the next fetch of that segment usually does not occur within the next 10-cycles. Second, if it does, it can be supplied by the icache (which is likely to contain the fetch block as it recently satisfied the original request for the block). For some benchmarks (e.g., gs), a longer fill unit latency results in a slightly higher performance. A longer latency sometimes delays the replacement of a useful trace cache segment with a less useful one.

5.5 Branch Predictor Issues

The multiple branch predictor is a crucial element of the trace cache fetch mechanism. If the trace cache is not supported by a predictor capable of making accurate predictions, gains in effective fetch rate will be offset by losses from more discarded fetches, likely resulting in a loss in performance.

In this section we examine several organizations for the pattern history table (PHT).

A pattern history table entry contains the most likely branch outcome when a particular pattern is encountered in the first level history. For our baseline multiple branch predictor, we are using a pattern history table entry composed of 3 two-bit saturating counters.

This entry format was derived as a cost-effective version of the scheme used in [19, 6] where a PHT entry is composed of 7 two-bit counters. Figure 10 shows how the seven counters are used to supply three predictions per cycle. The first two-bit counter supplies the prediction for the first branch and is used to select which of two two-bit counters supplies the prediction for the second branch. Both predictions are used to select one of four two-bit counters to supply the prediction for the third branch. All three predictions are made with a single access to the PHT.

The configurations evaluated in this experiment include the 7 counter scheme, the 3 counter scheme, and a scheme presented by Menezes et al [16] where each PHT entry contains the most likely path through a program subgraph containing 3 branches. In this scheme, the PHT entries are 4 bits wide: 3 bits to encode the likely path (8 paths are possible) and a fourth bit which records the likeliness of this path. Figure 11 shows the performance comparison of the various PHT schemes. The size of the predictor is kept roughly constant. The 7 counter scheme uses 14 bits of history and requires 32KB of storage, the 3 counter scheme uses 15 bits of history and requires 24KB of storage, the likely path scheme uses 16 bits of history and requires 32KB of storage.

In general, the 3 counter scheme slightly outperforms the other two schemes. It outperforms the 7 counter scheme because of better utilization: many counters within an entry in the 7 counter scheme are likely to go unused because of the prevalence of biased branches. The 3 counter scheme outperforms the likely path scheme because of more hysteresis.

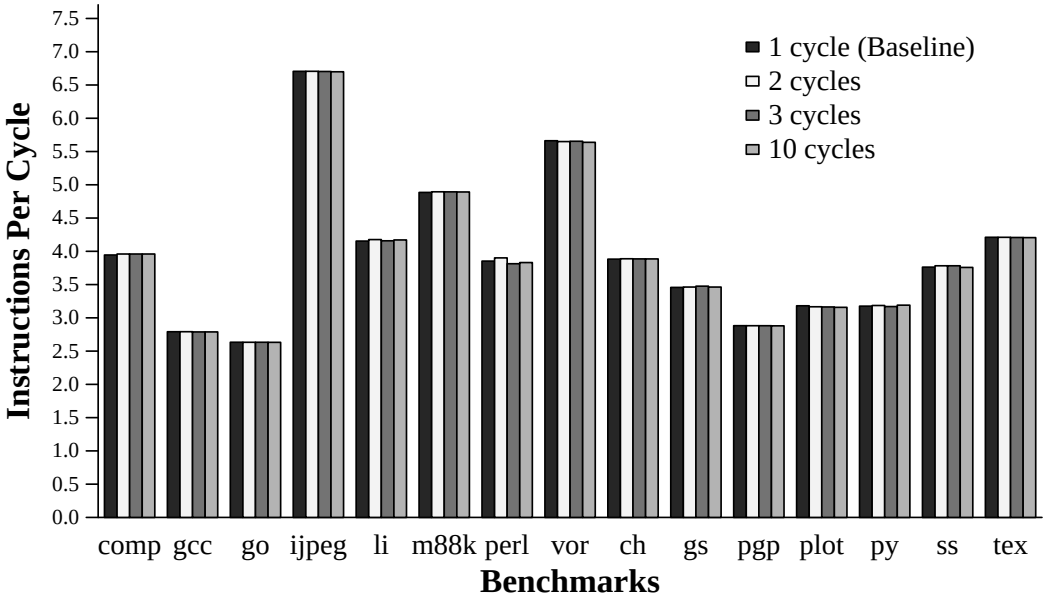

Figure 9: Fill Unit Latency is not a major influence on performance.

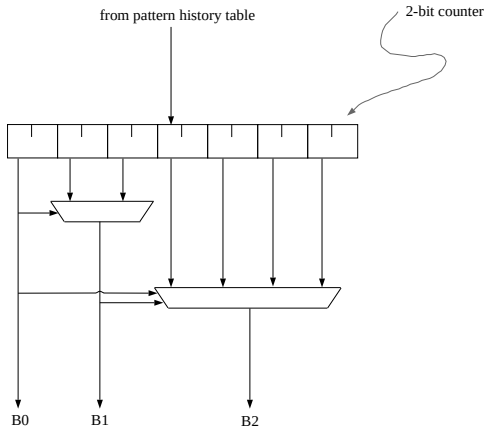

Figure 10: Seven two-bit counters are used to provide three predictions. B0 is the prediction for the first branch, B1 is the prediction of the second and B2 is for the third.

5.6 Comparison to an Aggressive Single Block Mechanism

The experiments thus far have evaluated various design options for the trace cache fetch mechanism. We provide a comparison of the enhances trace cache to the current dominant techniques for fetch engine design. Rotenberg et al. [22] presented a thorough comparison of the trace cache's performance on the SPECint92 and IBS benchmarks to a few of the hardware-based multiple block fetch techniques mentioned in section 2. Here we present a comparison of our baseline configuration with a aggressive single block fetch mechanism and an icache capable of fetching up to the first taken branch.

The components of the single fetch block mechanism are approximately the same size and access complexity as the trace cache counterparts in the baseline configuration. The single block mechanism consists of a single cycle, 128KB, 4-way set associative instruction cache capable of fetching two consecutive cache lines and supplying up to 16 instructions or until the first control flow instruction each cycle. The next fetch address is generated with an 8KB branch target buffer. The single branch predictor is a hybrid predictor, consisting of two components: a 15-bit PAs predictor and a 15-bit gshare predictor. The selection between the components is done by a 15-bit gshare-style selector. Combining a per-address predictor with a gshare predictor has been shown to be an effective way of boosting predictor accuracy [13, 2]. This fetch mechanism is similar to the one used on the Alpha 21264 [12].

We also compare the trace cache to a mechanism where the instruction cache is capable of supplying a sequential stream of instructions beyond conditional branches which are predicted to be not taken. This configuration requires the use of a multiple branch predictor (up to three branches can be fetched) and therefore does not benefit from the high prediction accuracy attainable with the hybrid predictor. We call this scheme the sequential block icache.

Figure 12 displays the experimental results. The trace cache outperforms both schemes

for most benchmarks. Listed on the graph are the percentage increases over the single block icache scheme. On average, the trace cache delivers a performance improvement of 28% over the single block icache and a 15% improvement over the sequential block icache.

The large boost in performance of the trace cache mechanism comes from a significant increase in average effective fetch rate — the average number of instructions delivered per on-path fetch. Our experimental results show (not presented here) that this rate doubles with a trace cache over the single block icache.

While the increased fetch rate improves the performance of the trace cache mechanism, the losses due to branch mispredictions and, to a lesser extent, cache misses degrade performance. Our experimental data indicates that the conditional branch misprediction rate drops from 6.6% with the trace cache to 5.5% with the single block icache.

6 Conclusions

In this paper we have examined some of the critical design parameters of the trace cache fetch mechanism. The trace cache supplies multiple fetch blocks of instructions each cycle by storing logically contiguous instruction sequences in physically contiguous storage.

We have demonstrated that the ability to partially match a trace segment provides an average 14% performance boost over a configuration which requires a complete match. Inactive issue is a hedge against branch mispredictions and yields 17% improvement over the baseline and is particularly helpful on benchmarks which suffer from poor prediction accuracy.

We have also demonstrated that trace creation can be done speculatively with no degradation in performance. Creating traces speculatively at issue time may allow for for simpler implementations. In addition, the latency in creating traces has negligible effects on performance.

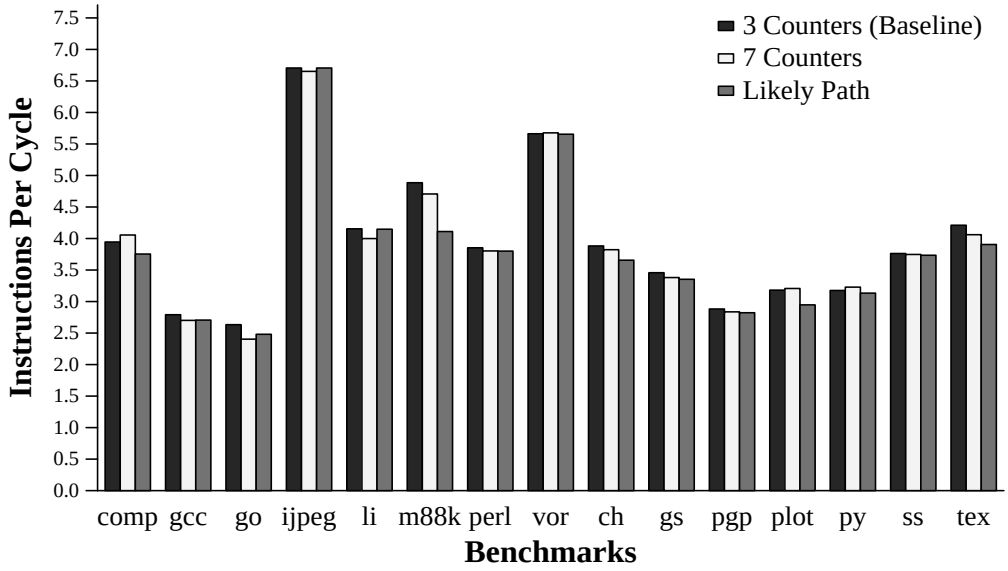

Figure 11: Evaluation of various pattern history table entry configurations.

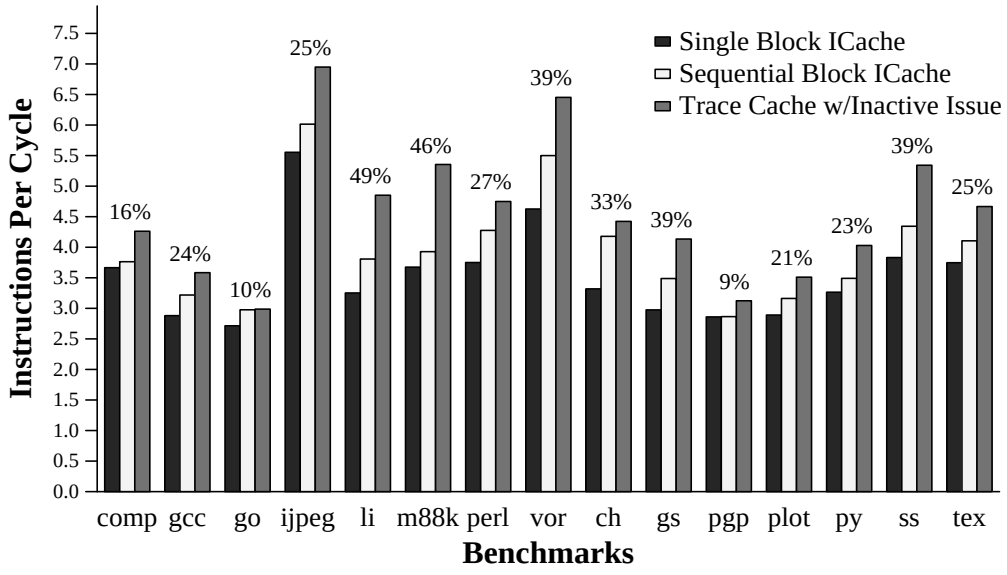

Figure 12: The performance of the baseline trace cache fetch mechanism against two icache fetch schemes.

When compared with an aggressive single block fetch icache, the trace cache attains an average performance increase of 28% and attains a 15% improvement over a sequential block icache. Much of this performance increase comes from the increase in effective fetch rate, which is twice that of the single block engine.

Because it is a low-complexity technique for delivering high instruction bandwidth, the trace cache will be an important component of future microprocessors [21]. There remain many important issues which need resolving. We are currently focusing on quantifying and reducing instruction redundancy in the trace cache and on developing techniques for creating longer trace segments.

7 Acknowledgements

We would like to thank several other members of the HPS research group, Rob Chappell, Marius Evers, Peter Kim, Paul Racunas, Jared Stark, and several of its alumni, in particular Mike Shebanow. We would like to thank our corporate sponsors — Intel, HAL, and NCR — for funding our work.

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, 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.

[3] R. P. Colwell, R. P. Nix, J. J. O'Donnell, D. B. Papworth, and P. K. Rodman, "A VLIW architecture for a trace scheduling compiler," *IEEE Transactions on Computers*, vol. 37, no. 8, pp. 967–979, August 1988.

[4] T. M. Conte, K. N. Menezes, P. M. Mills, and B. A. Patel, "Optimization of instruction fetch mechanisms for high issue rates," in *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, 1995.

[5] J. A. Fisher, "Trace scheduling: A technique for global microcode compaction," *IEEE Transactions on Computers*, vol. C-30, no. 7, pp. 478–490, July 1981.

[6] 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.

[7] D. H. Friendly, S. J. Patel, and Y. N. Patt, "Putting the fill unit to work: Dynamic optimizations for trace cache microprocessors," in *Proceedings of the 30th Annual ACM/IEEE International Symposium on Microarchitecture*, 1997.

[8] E. Hao, P.-Y. Chang, M. Evers, and Y. N. Patt, "Increasing the instruction fetch rate via block-structured instruction set architectures," in *Proceedings of the 29th Annual ACM/IEEE International Symposium on Microarchitecture*, 1996.

[9] W. W. Hwu, S. A. Mahlke, W. Y. Chen, P. P. Chang, N. J. Warter, R. A. Bringmann, R. G. Ouellette, R. E. Hank, T. Kiyohara, G. E. Haab, J. G. Holm, and D. M. Lavery, "The superblock: An effective technique for VLIW and superscalar compilation," *Journal of Supercomputing*, vol. 7, no. 9-50, , 1993.

[10] 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.

[11] J. D. Johnson, "Expansion caches for superscalar microprocessors," Technical Report CSL-TR-94-630, Stanford University, Palo Alto CA, June 1994.

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

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

[14] S. Melvin and Y. Patt, "Enhancing instruction scheduling with a block-structured ISA," *International Journal on Parallel Processing*, 1994.

[15] S. W. Melvin and Y. N. Patt, "Performance benefits of large execution atomic units in dynamically scheduled machines," in *Proceedings of Supercomputing '89*, pp. 427–432, 1989.

[16] K. N. Menezes, S. W. Sathaye, and T. M. Conte, "Path prediction for high issue-rate processors," in *Proceedings of the 1997 ACM/IEEE Conference on Parallel Architectures and Compilation Techniques*, 1997.

[17] R. Nair and M. E. Hopkins, "Exploiting instruction level parallelism in processors by caching scheduled groups," in *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pp. 13–25, 1997.

[18] S. J. Patel, M. Evers, and Y. N. Patt, "Improving trace cache effectiveness with branch promotion and trace packing," in *Proceedings of the 25th Annual International Symposium on Computer Architecture*, 1998.

[19] 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.

[20] 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.

[21] F. Pollack, Description of the intel microprocessor roadmap, Press briefing, October 1998.

[22] 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.

[23] 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.

[24] A. Seznec, S. Jourdan, P. Sainrat, and P. Michaud, "Multiple-block ahead branch predictors," in *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems*, 1996.

[25] E. Sprangle and Y. Patt, "Facilitating superscalar processing via a combined static/dynamic register renaming scheme," in *Proceedings of the 27th Annual ACM/IEEE International Symposium on Microarchitecture*, pp. 143–147, 1994.

[26] 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*, pp. 34 – 43, 1997.

[27] S. Vajapeyam and T. Mitra, "Improving superscalar instruction dispatch and issue by exploiting dynamic code sequences," in *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pp. 1–12, 1997.

[28] T.-Y. Yeh, D. Marr, and Y. N. Patt, "Increasing the instruction fetch rate via multiple branch prediction and branch address cache," in *Proceedings of the International Conference on Supercomputing*, pp. 67–76, 1993.

[29] T.-Y. Yeh and Y. N. Patt, "Two-level adaptive branch prediction," in *Proceedings of the 24th Annual ACM/IEEE International Symposium on Microarchitecture*, pp. 51–61, 1991.

Sanjay J. Patel is currently a PhD candidate in Computer Science and Engineering at the University of Michigan, Ann Arbor. He is investigating techniques for high bandwidth instruction supply for processors in the era of 100M and 1B transistors. His research interest include trace caches, branch prediction, memory bandwidth issues, and the performance simulation of microarchitectures. He received a MS from the University of Michigan and has worked for Digital Equipment Corporation.

Dan Friendly is a PhD candidate in computer science and engineering at the University of Michigan. His research interests include high bandwidth fetch mechanisms, superscalar architectures, and speculative execution techniques. He received a BA in American social issues from Macalester College and an MS in computer science from the University of Michigan.

Yale N. Patt is Professor of Electrical Engineering and Computer Science at the University of Michigan, where he teaches undergraduate and graduate courses in computer architecture, and directs the PhD research of nine graduate students in computer architecture, high-performance processor design, and computer systems implementation. Patt earned his BS from Northeastern University and his MS and PhD from Stanford University, all in electrical engineering. He received the 1995 IEEE Emanuel R. Piore Award and the 1996 ACM/IEEE Eckert-Mauchly Award. He is a Fellow of the IEEE.