Trace Cache: a Low Latency Approach to High Bandwidth Instruction Fetching

Eric Rotenberg
Computer Science Dept.
Univ. of Wisconsin - Madison
ericro@cs.wisc.edu

Steve Bennett
Intel Corporation
sbennett@ichips.intel.com

James E. Smith
Dept. of Elec. and Comp. Engr.
Univ. of Wisconsin - Madison
jes@ece.wisc.edu

Copyright 1996 IEEE. Published in the Proceedings of the 29th Annual International Symposium on Microarchitecture, Dec. 2-4, 1996, Paris, France. 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

Trace Cache: a Low Latency Approach to High Bandwidth Instruction Fetching

Eric Rotenberg
Computer Science Dept.
Univ. of Wisconsin - Madison
ericro@cs.wisc.edu

Steve Bennett
Intel Corporation
sbennett@ichips.intel.com

James E. Smith
Dept. of Elec. and Comp. Engr.
Univ. of Wisconsin - Madison
jes@ece.wisc.edu

Abstract

As the issue width of superscalar processors is increased, instruction fetch bandwidth requirements will also increase. It will become necessary to fetch multiple basic blocks per cycle. Conventional instruction caches hinder this effort because long instruction sequences are not always in contiguous cache locations.

We propose supplementing the conventional instruction cache with a trace cache. This structure caches traces of the dynamic instruction stream, so instructions that are otherwise noncontiguous appear contiguous. For the Instruction Benchmark Suite (IBS) and SPEC92 integer benchmarks, a 4 kilobyte trace cache improves performance on average by 28% over conventional sequential fetching. Further, it is shown that the trace cache's efficient, low latency approach enables it to outperform more complex mechanisms that work solely out of the instruction cache.

1. Introduction

High performance superscalar processor organizations divide naturally into an instruction fetch mechanism and an instruction execution mechanism (Figure 1). The fetch and execution mechanisms are separated by instruction issue buffer(s), for example queues, reservation stations, etc. Conceptually, the instruction fetch mechanism acts as a "producer" which fetches, decodes, and places instructions into the buffer. The instruction execution engine is the "consumer" which removes instructions from the buffer and executes them, subject to data dependence and resource constraints. Control dependences (branches and jumps) provide a feedback mechanism between the producer and consumer.

Processors having this organization employ aggressive techniques to exploit instruction-level parallelism. Wide dispatch and issue paths place an upper bound on peak instruction throughput. Large issue buffers are used to maintain a *window* of instructions necessary for detecting parallelism, and a large pool of physical registers provides destinations for all the in-flight instructions issued from the window. To enable concurrent execution of instructions, the execution engine is composed of many parallel functional units. The fetch engine speculates past multiple branches in order to supply a continuous instruction stream to the window.

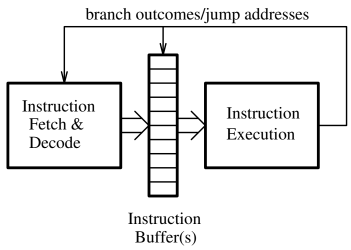

Figure 1. Decoupled fetch/execute engines.

The trend in superscalar design is to increase the scale of these techniques: wider dispatch/issue, larger windows, more physical registers, more functional units, and deeper speculation. To maintain this trend, it is important to balance all parts of the processor – any bottlenecks diminish the benefit of aggressive ILP techniques.

In this paper, we are concerned with instruction fetch bandwidth becoming a performance bottleneck. Instruction fetch performance depends on a number of factors. Instruction cache hit rate and branch prediction accuracy have long been recognized as important problems in fetch performance and are well-researched areas. In this paper, we are interested in additional factors that are only now emerging as processor issue rates exceed four instructions per cycle:

- *branch throughput* – If only one conditional branch is predicted per cycle, then the window can grow at the rate of only one basic block per cycle. Predicting multiple branches per cycle allows the overall instruction throughput to be correspondingly higher.
- *noncontiguous instruction alignment* – Because of branches and jumps, instructions to be fetched during any given cycle may not be in contiguous cache locations. Hence, there must be adequate paths and logic available to fetch and align noncontiguous basic blocks and pass them up the pipeline. That is, it is not enough

for the instructions to be present in the cache, it must also be possible to access them in parallel.

- *fetch unit latency* – Pipeline latency has a profound impact on processor performance. This is due to the cost of refilling the pipeline after incorrect control speculation. In the case of the fetch unit, we are concerned with the startup cost of redirecting fetching after resolving a branch misprediction, jump, or instruction cache miss. Inevitably, the need for higher branch throughput and noncontiguous instruction alignment will increase fetch unit latency; yet ways must be found to minimize the latency impact.

Current fetch units are limited to one branch prediction per cycle and can therefore fetch 1 basic block per cycle or up to the maximum instruction fetch width, whichever is smaller. The data in Table 1 shows that the average size of basic blocks is around 4 or 5 instructions for integer codes. While fetching a single basic block each cycle is sufficient for implementations that issue at most 4 instructions per cycle, it is not so for processors with higher peak issue rates. If we introduce multiple branch prediction [1][16], then the fetch unit can at least fetch multiple *contiguous* basic blocks in a cycle. Data for the number of instructions between taken branches shows that the upper bound on fetch bandwidth is still somewhat limited in this case, due to the frequency of taken branches. Therefore, if a taken branch is encountered, it is necessary to fetch instructions down the taken path in the same cycle that the branch is fetched.

Benchmark	taken %	avg basic block size	# instr between taken branches
eqntott	86.2%	4.20	4.87
espresso	63.8%	4.24	6.65
xlisp	64.7%	4.34	6.70
gcc	67.6%	4.65	6.88
sc	70.2%	4.71	6.71
compress	60.9%	5.39	8.85

Table 1. Branch and basic block statistics.

1.1. The trace cache

The job of the fetch unit is to feed the dynamic instruction stream to the decoder. A problem is that instructions are placed in the cache in their compiled order. Storing programs in static form favors fetching code that does not branch or code with large basic blocks. Neither of these cases is typical of integer code.

We propose a special instruction cache which captures dynamic instruction sequences. This structure is called a *trace cache* because each line stores a snapshot, or trace, of the dynamic instruction stream, as shown in Figure 2. A trace is a sequence of at most n instructions and at most m basic blocks starting at any point in the dynamic instruction stream. The limit n is the trace cache line size, and m is the branch predictor throughput. A trace is fully specified by a starting address and a sequence of up to $m - 1$ branch outcomes which describe the path followed. The first time a trace is encountered, it is allocated a line in the trace cache. The line is filled as instructions are fetched from the instruction cache. If the same trace is encountered again in the course of executing the program, i.e. the same starting address and predicted branch outcomes, it will be available in the trace cache and is fed directly to the decoder. Otherwise, fetching proceeds normally from the instruction cache.

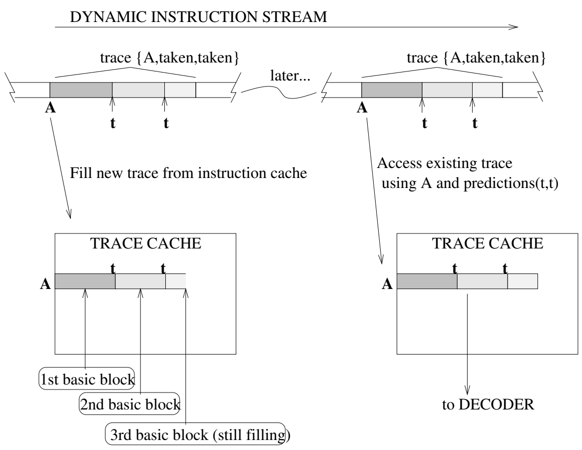

Figure 2. High level view of the trace cache.

The trace cache approach relies on dynamic sequences of code being reused. This may be the case for two reasons:

- *temporal locality* – like the primary instruction cache, the trace cache can count on instructions which have been recently used being used again in the near future.
- *branch behavior* – most branches tend to be biased towards one direction, which is why branch prediction accuracy is usually high. Thus, it is likely that certain paths through the control flow graph will be followed frequently.

1.2. Related prior work

Three recent studies have focused on high bandwidth instruction fetching and are closely related to the research reported here. All of these attempt to fetch multiple, possibly noncontiguous basic blocks each cycle from the instruction cache.

First, Yeh, Marr, and Patt [16] consider a fetch mechanism that provides high bandwidth by predicting multiple branch target addresses every cycle. The method features a *Branch Address Cache*, a natural extension of the branch target buffer [8]. With a branch target buffer, a single branch prediction and a BTB hit produces the starting address of the next basic block. Similarly, a hit in the branch address cache combined with multiple branch predictions produces the starting addresses of the next *several* basic blocks. These addresses are fed into a highly interleaved instruction cache to fetch multiple basic blocks in a single cycle.

A second study by Franklin and Dutta [2] uses a similar approach to the branch address cache (providing multiple branch targets), but with a new method for predicting multiple branches in a single cycle. Their approach hides multiple individual branch predictions within a single prediction; e.g. rather than make 2 branch predictions, make 1 prediction that selects from among 4 paths. This enables the use of more accurate two-level predictors.

Another hardware scheme proposed by Conte, Mills, Menezes, and Patel [1] uses two passes through an interleaved branch target buffer. Each pass through the branch target buffer produces a fetch address, allowing two nonadjacent cache lines to be fetched. In addition, the interleaved branch target buffer enables detection of any number of branches in a cache line. In particular, the design is able to detect short forward branches within a line and eliminate instructions between the branch and its target using a *collapsing buffer*. The work also proposes compiler techniques to reduce the frequency of taken branches.

Two previously proposed hardware structures are similar to the trace cache but exist in different applications. The fill unit, proposed by Melvin, Shebanow and Patt [10], caches RISC-like instructions which are derived from a CISC instruction stream. This predecoding eased the problem of supporting a complex instruction set such as VAX on the HPS restricted dataflow engine. Franklin and Smotherman [3] extended the fill unit's role to dynamically assemble VLIW-like instruction words from a RISC instruction stream, which are then stored in a *shadow cache*. The goal of this structure is to ease the issue complexity of a wide issue processor.

1.3. Problems with other fetch mechanisms

Recall that the job of the fetch unit is to feed the dynamic instruction stream to the decoder. Unlike the trace cache approach, previous designs have only the conventional instruction cache, containing a static form of the program, to work with. Every cycle, instructions from noncontiguous locations must be fetched from the instruction cache and assembled into the predicted dynamic sequence. There are problems with this approach:

- Pointers to all of the noncontiguous instruction blocks must be generated before fetching can begin. This implies a level of indirection, through some form of branch target table (e.g. branch target buffer), which translates into an additional pipeline stage before the instruction cache.
- The instruction cache must support simultaneous access to multiple, noncontiguous cache lines. This forces the cache to be multiported; if multiporting is done through interleaving, bank conflicts will occur.
- After fetching the noncontiguous instructions from the cache, they must be assembled into the dynamic sequence. Instructions must be shifted and aligned to make them appear contiguous to the decoder. This most likely translates into an additional pipeline stage after the instruction cache.

The trace cache approach avoids these problems by caching dynamic instruction sequences themselves, ready for the decoder. If the predicted dynamic sequence exists in the trace cache, it does not have to be recreated on the fly from the instruction cache's static representation. In particular, no additional stages before or after the instruction cache are needed for fetching noncontiguous instructions. The stages do exist, but not on the critical path of the fetch unit – rather, on the fill side of the trace cache. The cost of this approach is redundant instruction storage: the same instructions may reside in both the primary cache and the trace cache, and there might even be redundancy among lines in the trace cache.

1.4. Contributions

As with prior work in high bandwidth instruction fetching, this paper demonstrates the importance of fetching past multiple possibly-taken branches each cycle. Unlike other work in the area, we place equal emphasis on fetch unit latency. The end result is the trace cache as a means for low latency, high bandwidth instruction fetching.

Another contribution is a detailed simulation study comparing proposed high bandwidth fetch mechanisms including the trace cache. Previously, the approaches described in Section 1.2 could not be directly compared due to different experimental setups – different ISAs, processor execution models, branch predictors, caches, workloads, and metrics.

In the course of this work, many microarchitectural and logic design issues arose. We looked at issues for not only the trace cache, but other proposed mechanisms as well. The results of this detailed study are documented in [12].

1.5. Paper overview

In the next section the trace cache fetch unit is described in detail. Section 3 follows up with an analysis of other proposed high bandwidth fetch mechanisms. In Section 4 we describe the simulation methodology including the processor model, workload, and performance metric. Simulation results are presented in Section 5. As part of the study in Section 5, we compare the trace cache with previously proposed high performance fetch mechanisms.

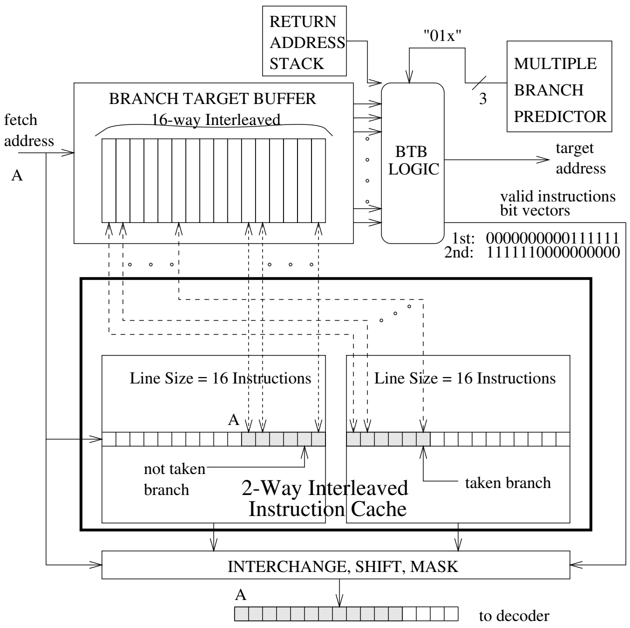

Figure 3. The core fetch unit.

2. Trace cache

In Section 1.1 we introduced the concept of the trace cache – an instruction cache which captures dynamic instruction sequences. We now present a trace cache implementation. Because the trace cache is not intended to replace the conventional instruction cache or the fetch hardware around it, we begin with a description of the core fetch mechanism. We then show how the core fetch unit is augmented with the trace cache.

2.1. Core fetch unit

The core fetch unit is implemented using established hardware schemes. It is called *interleaved sequential* in [1]. Fetching up to the first predicted taken branch each cycle can be done using the combination of an accurate multiple branch predictor [16], an interleaved branch target buffer (BTB) [1][8], a return address stack (RAS) [6], and a 2-way interleaved instruction cache [1][4]. Refer to Figure 3.

The core fetch unit is designed to fetch as many contiguous instructions possible, up to a maximum instruction limit and a maximum branch limit. The instruction constraint is imposed by the width of the datapath, and the branch constraint is imposed by the branch predictor throughput. For demonstration, a fetch limit of 16 instructions and 3 branches is used throughout.

The cache is interleaved so that 2 consecutive cache lines can be accessed; this allows fetching sequential code that spans a cache line boundary, always guaranteeing a full cache line or up to the first taken branch [4]. This scheme requires minimal complexity for aligning instructions: (1) logic to swap the order of the two cache lines (interchange switch), (2) a left-shifter to align the instructions into a 16-wide instruction latch, and (3) logic to mask off unused instructions.

All banks of the BTB are accessed in parallel with the instruction cache. They serve the role of (1) detecting branches in the instructions currently being fetched and (2) providing their target addresses, in time for the next fetch cycle. The BTB must be n-way interleaved, where n is the number of instructions in a cache line. This is so that all instructions within a cache line can be checked for branches in parallel [1]. The BTB can detect other types of control transfer instructions as well. If a jump is detected, the jump address may be predicted. (Jump target predictions are not considered in this paper, however.) Return addresses can almost always be obtained with no penalty by using a call/return stack. If the BTB detects a return in the instructions being fetched, it pops the address at the top of the RAS.

Notice in Figure 3 that the branch predictor is separate from the BTB. This is to allow for predictors that are more accurate than the 1-bit or 2-bit counters normally stored with each branch entry in the BTB. While storing counters with each branch achieves multiple branch prediction trivially, branch prediction accuracy is limited. Branch prediction is fundamental to ILP, and should have precedence over other factors. For high branch prediction accuracy, we use a 4kB GAg(14) correlated branch predictor [15]. The 14 bit global branch history register indexes into a single pattern history table. This predictor was chosen for its accuracy and because it is more easily extended to multiple branch predictions than other predictors which require address information [16][2]. Multiporting the pattern history table and changing its organization slightly extends the single correlated branch predictor to multiple predictions each cycle, as proposed in [16]. (Refer to [12] for an implementation.)

BTB logic combines the BTB hit information with the branch predictions to produce the next fetch address, and to generate trailing zeroes in the *valid instruction bit vectors* (if there is a predicted taken branch). The leading zeroes in the valid instruction bit vectors are determined by the low-order bits of the current fetch address. The masking logic is controlled by these bit vectors.

Both the interchange and shift logic are controlled by the low-order bits of the current fetch address. *This is a key point: the left-shift amount is known at the beginning of the fetch cycle, and has the entire cache access to fanout to the shifter datapath.* Further, if a transmission gate barrel shifter is used, instructions pass through only one transmission gate delay with a worst case capacitive loading of 15 other transmission gates on both input and output. In summary, control is not on the critical path, and datapath delay is minimal. Therefore, in our simulations we treat the core fetch unit as a single pipeline stage.

2.2. Adding the trace cache

The core fetch unit can only fetch contiguous sequences of instructions, i.e. it cannot fetch past a taken branch in the same cycle that the branch is fetched. The trace cache provides this additional capability. The trace cache together with the core fetch unit is shown in Figure 4.

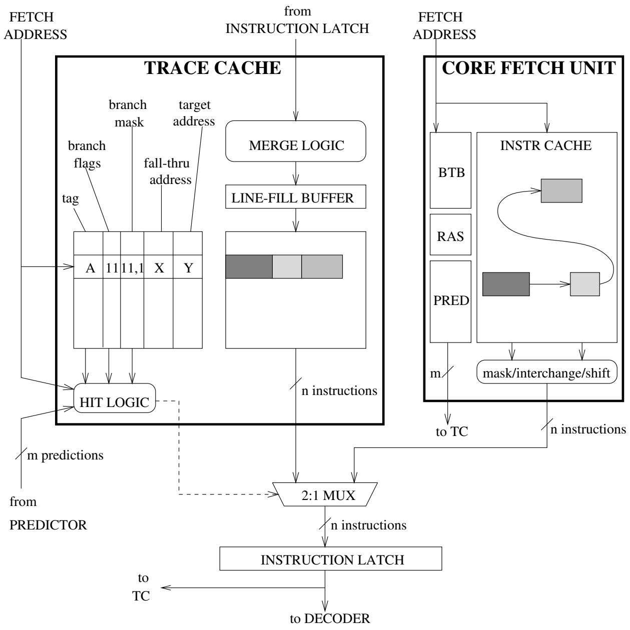

Figure 4. The trace cache fetch mechanism.

The trace cache is made up of instruction traces, control information, and line-fill buffer logic. The length of a trace is limited in two ways – by number of instructions n and by number of basic blocks m. The former limit n is chosen based on the peak dispatch rate. The latter limit m is chosen based on n and the average number of instructions in a basic block. m also determines, or is constrained by, the number of branch predictions made per cycle. In Figure 4, $n = 16$ and $m = 3$. The control information is similar to the tag array of standard caches but contains additional state information:

- *valid bit*: indicates this is a valid trace.
- *tag*: identifies the starting address of the trace.
- *branch flags*: there is a single bit for each branch within the trace to indicate the path followed after the branch (taken/not taken). The m^{th} branch of the trace does not need a flag since no instructions follow it, hence there are only $m - 1$ bits instead of m bits.
- *branch mask*: state is needed to indicate (1) the number of branches in the trace and (2) whether or not the trace ends in a branch. This is needed for comparing the correct number of branch predictions against the same number of branch flags, when checking for a trace hit. This is also needed by the branch predictor to know how many predictions were used. The first $\lceil log_2(m + 1) \rceil$ bits encode the number of branches. One more bit indicates if the last instruction in the trace is a branch; if true, the branch's corresponding branch flag does not need to be checked since no instructions follow it.
- *trace fall-through address*: next fetch address if the last branch in the trace is predicted not taken.
- *trace target address*: next fetch address if the last branch in the trace is predicted taken.

The trace cache is accessed in parallel with the instruction cache and BTB using the current fetch address. The predictor generates multiple branch predictions while the caches are accessed. The fetch address is used together with the multiple branch predictions to determine if the trace read from the trace cache matches the predicted sequence of basic blocks. Specifically, a trace cache hit requires that (1) the fetch address match the tag and (2) the branch predictions match the branch flags. The branch mask ensures that the correct number of prediction bits are used in the comparison. On a trace cache hit, an entire trace of instructions is fed into the instruction latch, bypassing the instruction cache.

On a trace cache miss, fetching proceeds normally from the instruction cache, i.e. contiguous instruction fetching. The line-fill buffer logic services trace cache misses. In the example in Figure 4, three basic blocks are fetched one at a time from the instruction cache, since all branches are predicted taken. The basic blocks are latched one at a time into the line-fill buffer; the line-fill control logic serves to merge each incoming block of instructions with preceding instructions in the line-fill buffer. Filling is complete when either n instructions have been traced or m branches have been detected in the trace. At this point the contents of the line-fill buffer are written into the trace cache. The branch flags and branch mask are generated during the line-fill process, and the trace target and fall-through addresses are computed at the end of the line-fill. If the trace does not end in a branch, the target address is set equal to the fall-through address.

There are different classes of control transfer instructions – conditional branches, unconditional branches, calls or direct jumps, returns, indirect jumps, and traps – yet so far only conditional branches have been discussed. The complex alternative for handling all of these cases is to add additional bits to each branch flag to distinguish the type of control transfer instruction. Further, the line-fill buffer must stop filling a trace when a return, indirect jump, or trap is encountered, because these control transfer instructions have an indeterminite number of targets, whereas the predictor can only predict one of two targets. Lastly, the branch mask and the hit logic are made slightly more complex since unconditional branches and calls should not be involved in prediction (the outcome is known).

We simplify these complications in two ways. First, the trace cache does not store returns, indirect jumps, or traps at all; the line-fill buffer aborts a fill when it detects any of these instructions. Second, unconditional branches and calls can be viewed as conditional branches that are extremely predictable; from this point of view, they can be grouped into the conditional branch class and not be treated any differently. With these two simplifications, the trace cache has only to deal with conditional branches.

The size of a direct mapped trace cache with 64 lines, $n = 16$, and $m = 3$ is 712 bytes for tags/control and 4 kilobytes for instructions (comparable in area to the correlated branch predictor, 4kB). This configuration is used in the experiments which follow.

2.3. Trace cache design space

The trace cache depicted in Figure 4 is the simplest design among many alternatives. It is the implementation used in simulations of the trace cache. However, the design space deserves some attention:

1) associativity: The trace cache can be made more associative to reduce conflict misses.

2) multiple paths: A downside of the simple trace cache is that from a given starting address, only one trace can be stored. It might be advantageous to be able to store multiple paths eminating from a given address. This can be thought of as another form of associativity – *path associativity*. Adding path associativity could reduce thrashing between traces that start at the same address.

3) partial matches: An alternative to providing path associativity is to allow *partial hits*. If the fetch address matches the starting address of a trace and the first few branch predictions match the first few branch flags, provide only a prefix of the trace. This is in place of the simple "all or nothing" approach we use. The additional cost of this scheme is that intermediate basic block addresses must be stored for the same reason that trace target and fall-through addresses are stored. Also, there is the question of whether or not a partial hit be treated as a miss (i.e. to replace the line or not).

4) other indexing methods: The simple trace cache indexes with the fetch address and includes branch predictions in the tag match. Alternatively, the index into the trace cache could be derived by concatenating the fetch address with the branch prediction bits. This effectively achieves path associativity while keeping a direct mapped structure, because different paths starting at the same address now map to consecutive locations in the trace cache.

5) fill issues: While the line-fill buffer is collecting a new trace, the trace cache continues to be accessed by the fetch unit. This means a miss could occur in the midst of handling a previous miss. The design options in order of increasing complexity are: ignore any new misses, delay servicing new misses until the line-fill buffer is free, or provide multiple line-fill buffers to support concurrent misses. Another issue is whether to fill the trace cache with speculative traces or to wait for branch outcomes before committing a trace.

6) judicious trace selection: There are likely to be traces that are committed but never reused. These traces may displace useful traces, causing needless misses. To improve trace cache hit rates, the design could use a small buffer to store recent traces; a trace in this buffer is only committed to the trace cache after one or more hits to that trace.

7) victim trace cache: An alternative to judicious trace selection is to use a victim cache [5]. It may keep valuable traces from being permanently displaced by useless traces.

3. Other high bandwidth fetch mechanisms

In this section we analyze the organization of two previously proposed fetch mechanisms aimed at fetching and aligning multiple noncontiguous basic blocks each cycle. The analysis compares these mechanisms against the trace cache, with latency being a key point for comparison.

3.1. Branch address cache

The branch address cache fetch mechanism proposed by Yeh, Marr, and Patt [16] is shown in Figure 5. There are four primary components: (1) a branch address cache (BAC), (2) a multiple branch predictor, (3) an interleaved instruction cache, and (4) an interchange and alignment network. The BAC extends the BTB to multiple branches by storing a tree of target and fall-through addresses as depicted in Figure 6. The depth of the tree depends on the number of branches predicted per cycle.

In Figure 5, light grey boxes represent non-control transfer instructions and dark grey boxes represent branches; the fields in the BAC correspond to the tree in Figure 6, as indicated by the address labels A through O. The diagram depicts the two-stage nature of the design. In the first stage, an entry containing up to 14 basic block addresses is read from the BAC. From these addresses, up to 3 basic block addresses corresponding to the predicted path are selected. In this example, the next 3 branches are all predicted taken, corresponding to the sequence of basic blocks $\{$C,G,O$\}$. In the second stage, the cache reads the three basic blocks in parallel from its multiple banks. Since the basic blocks may be placed arbitrarily into the cache banks, they must pass through an alignment network to align them into dynamic program order and merge them into the instruction latch.

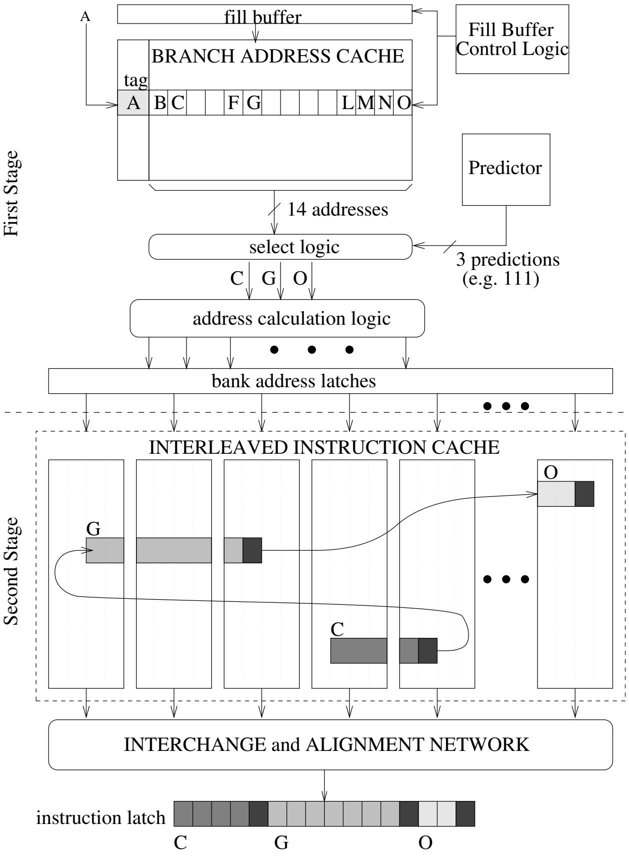

Figure 5. Branch address cache approach.

The two stages in this design are pipelined. During the second stage, while basic blocks $\{C,G,O\}$ are being fetched from the instruction cache, the BAC begins a new cycle using address O as its index. In general, the last basic block address indexing into the instruction cache is also the index into the BAC.

If an address misses in the BAC, an entry is allocated for the portion of the control flow graph which begins at that address. Branch target and fall-through addresses are filled in the entry as paths through the tree are traversed; an entry may contain holes corresponding to branches which have not yet been encountered.

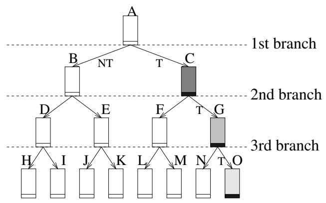

Figure 6. BAC stores subgraphs of the CFG.

Though conceptually the design has two pipeline stages, possibly one or more additional pipeline stages are implied by having the complicated alignment network. The alignment network must (1) interchange the cache lines from numerous banks (with more than two banks, the permutations grow quickly), and (2) collapse the basic blocks together, eliminating unused intervening instructions. Though not discussed in [16], logic like the collapsing buffer [1] discussed in the next section will be needed to do this.

3.2. Collapsing buffer

The instruction fetch mechanism proposed by Conte, Mills, Menezes, Patel [1] is illustrated in Figure 7. It is composed of (1) an interleaved instruction cache, (2) an interleaved branch target buffer (BTB), (3) a multiple branch predictor, (4) special logic after the BTB, and (5) an interchange and alignment network featuring a *collapsing buffer*.

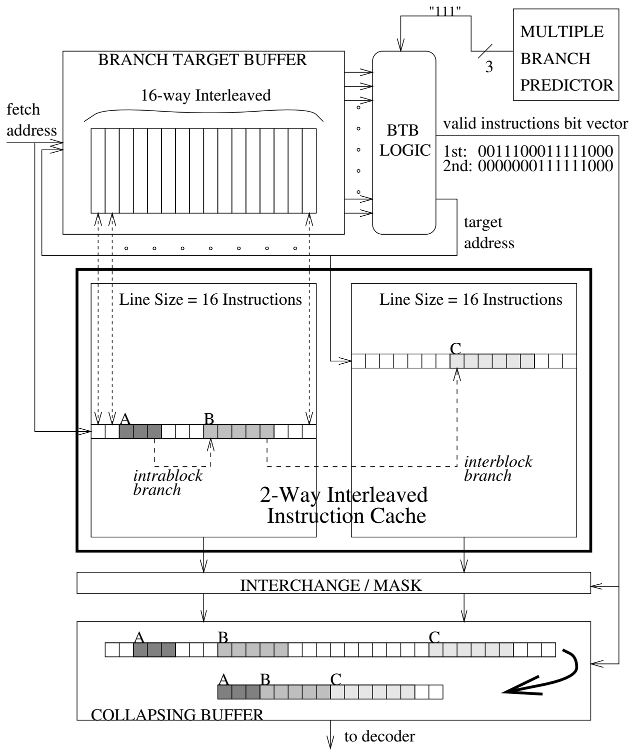

Figure 7. The collapsing buffer approach.

The hardware is similar to the core fetch unit of the trace cache (described in Section 3), but has two important distinctions. First, the BTB logic is capable of detecting *intrablock* branches – short hops within a cache line. Second, a single fetch goes through two BTB accesses. As will be described below, this allows fetching beyond one taken *interblock* branch – a branch out of the cache line. In both cases, the collapsing buffer uses control information generated by the BTB logic to merge noncontiguous basic blocks.

Figure 7 illustrates how three noncontiguous basic blocks labelled A, B, and C are fetched. The fetch address A accesses the interleaved BTB. The BTB indicates that there are two branches in the cache line, one at instruction 5 with target address B, the other at instruction 13 with target address C. Based on this branch information and branch predictions from the predictor, the BTB logic indicates which instructions in the fetched line are valid and produces the next basic block address, C.

The initial BTB lookup produces (1) a bit vector indicating the predicted valid instructions in the cache line (instructions from basic blocks A and B), and (2) the predicted target address C of basic block B. The fetch address A and target address C are then used to fetch two nonconsecutive cache lines from the interleaved instruction cache. This can be done only if the cache lines are in different banks. In parallel with this instruction cache access, the BTB is accessed again, using the target address C. This second, serialized lookup determines which instructions are valid in the second cache line and produces the next fetch address (the predicted successor of basic block C).

When the two cache lines have been read from the cache, they pass through masking and interchange logic and the collapsing buffer (which merges the instructions), all controlled by bit vectors produced by the two passes through the BTB. After this step, the properly ordered and merged instructions are captured in the instruction latches to be fed to the decoders.

This scheme has several disadvantages. First, the fetch line and successor line must reside in different cache banks. Bank conflicts can be reduced by adding more banks, but this requires a more complicated, higher latency interchange switch. Second, this scheme does not scale well for interblock branches; supporting additional interblock branches requires as many additional BTB accesses, all serialized. Third, the BTB logic requires a serial chain of n address comparators to detect intrablock branches, where n is the number of BTB banks. Most seriously, however, is that this fetch mechanism adds a significant amount of logic both before and after the instruction cache. The instruction fetch pipeline is likely to have three stages: (1) initial BTB lookup and BTB logic, (2) instruction cache access and second BTB lookup, and (3) interchange switch, masking, and collapsing buffer. The collapsing buffer takes only a single stage if implemented as a bus-based crossbar [1].

4. Simulation methodology

4.1. Processor model

Our simulation model follows the basic structure shown in Figure 1 – a fetch engine and an execute engine decoupled via instruction issue buffers. Various fetch engines – trace cache, branch address cache, and collapsing buffer – are modeled in detail. The processor execution part of the model is constrained only by true data dependences. *We assume unlimited hardware resources – any instructions in the instruction buffers that have their data available may issue.* This is done to place as much demand on the fetch unit as possible, making instruction fetch the performance bottleneck. In effect, unlimited register renaming and full dynamic instruction issue are assumed. Loads and stores are assumed to have oracle address disambiguation – loads and stores wait for previous stores only if there is a true address conflict. Also, the data cache always hits. The only hardware limitations imposed are the maximum size of the instruction buffer and the degree of superscalar dispatch. In all simulations, the size of the instruction buffer is 2048 useful instructions and the maximum fetch/dispatch bandwidth is 16 instructions per cycle. In summary, the amount of ILP exploited is limited by 5 factors:

- *maximum fetch/dispatch rate (16/cycle)*
- *maximum size of instruction window (2048)*
- *true data dependences in the program*
- *operation latencies*
- *performance of the fetch engine*

It is the last factor that we are interested in and which will vary between simulations.

The instruction pipeline is composed of 4 phases: fetch, dispatch, issue, and execution. The latency of the fetch phase is varied according to implementation, and the dispatch latency is fixed at 1 cycle. If all operands are available at or immediately after dispatch, instruction issue takes only 1 cycle; otherwise issue is delayed until operands arrive. Because of unlimited resources and unlimited register renaming, issue never stalls due to structural or register WAR/WAW hazards. After issue, execution takes a certain number of cycles based on the operation. Operation latencies are similar to those of the MIPS R10000 processor.

4.2. Workload

The six integer SPEC92 benchmarks and six benchmarks from the *Instruction Benchmark Suite* (IBS) [14] are used to evaluate the performance of the various fetch mechanisms. SPEC92 floating-point results can be found in [12].

The SPEC92 benchmarks were compiled on a Sun SPARCstation 10/30 using "gcc -O4 -static -fschedule-insns -fschedule-insns2". SPARC instruction traces were generated using *the Quick Profiler and Tracer* (QPT) [7] and then fed into the trace-driven SPARC processor simulator. The SPEC92 benchmarks were simulated to completion.

The IBS benchmarks are MIPS traces obtained via a logic analyzer connected to the CPU of a DECstation 3100. These benchmarks are a better test of instruction fetch performance than SPEC92 [14]. For one thing, a significant fraction of the traces are kernel and X-server references, increasing instruction path lengths. To simulate the IBS traces, we developed a trace-driven MIPS processor simulator similar to the SPARC one.

4.3. Performance metric

For measuring performance we use instructions completed per cycle (IPC), which is a direct measure of performance and is almost certainly the measure that counts most. The harmonic mean is used to average the performance of benchmarks.

5. Results

Table 2 summarizes the trace cache (TC), collapsing buffer (CB), and branch address cache (BAC) fetch unit parameters used in all experiments. In the sections which follow, results for the three high bandwidth implementations are compared.

As a base case for comparison, results are also presented for conventional instruction fetching. The core fetch unit of the trace cache (described in Section 2.1) is used as the base case. We will call the base case "sequential" (SEQ), since only sequential instructions can be fetched in a given cycle. To demonstrate the effect of branch throughput, two variations of SEQ are simulated: SEQ.1 is limited to one basic block per cycle, and SEQ.3 can fetch up to three contiguous basic blocks. Simulation parameters for SEQ are the same as those for TC in Table 2, but with the trace cache not present (and 1 or 3 branch predictions per cycle).

SIMULATION PARAMETER		INSTR SUPPLY MECHANISM		
		TC	CB	BAC
instruction fetch limit		16 instructions per cycle		
Multiple Branch Predictor	BHR	14 bits		
	PHT	2^{14} 2-bit counters (4 KB storage)		
	# pred/cyc	up to 3 predictions each cycle		
Instr Cache	size	128 KB		
	assoc	direct mapped		
	interleave	2-way	2-way	8-way
	line size	16 instr	16 instr	4 instr
	prefetch	none	none	3 lines
	miss penalty	10 cycles		
Ret Stack	depth	unlimited		
Branch Target Buffer	size	1K entries	1K entries	n/a
	assoc	dir map	dir map	
	interleave	16-way	16-way	
Trace Cache	size	64 entries	n/a	
	assoc	dir map		
	line size	16 instr		
	# conc fills	1		
Branch Address Cache	size	n/a		1K entries
	assoc			dir map
	# conc fills			1

Table 2. Fetch unit configurations.

The results are split into two sets. The first set assumes all fetch units have a latency of 1 cycle, in order to demonstrate each mechanism's ability to deliver bandwidth performance. The second set shows what happens when the extra pipe stages implied by CB and BAC are actually simulated.

5.1. Single-cycle fetch latency

The first set of results, the two graphs in Figure 8, assumes a fetch unit latency of 1 cycle for all schemes. This is done to isolate the ability of the fetch mechanisms to supply instruction bandwidth.

The first observation is that SEQ.3 gives a substantial performance boost over SEQ.1. The graph in Figure 9 shows that fetching past multiple not-taken branches each cycle yields performance improvement above 7% for all of the SPEC benchmarks. Over half of the SPEC benchmarks show a 17% or better performance improvement. Four of the IBS benchmarks show a 7% or greater improvement.

The second observation is that for both SPEC and IBS workloads, fetching past taken branches is a big win. Adding the TC function to the SEQ.3 mechanism yields about as much performance improvement as extending SEQ.1 to multiple not-taken branches per cycle.

The graph in Figure 10 shows the performance improvement that BAC, CB, and TC yield over SEQ.3 (SEQ.3 is used as the base instead of SEQ.1 because it is aggressive, yet not much more complex than SEQ.1). One might expect that under the single-cycle fetch latency assumption, the three approaches would perform similarly. This is the case for much of the IBS workload, with TC always performing as well or better than the other two schemes.

For the SPEC workload, however, TC enjoys a noticeable lead over CB. This is most likely because the original collapsing buffer was not designed to handle backward taken intrablock branches [1], whereas the TC can handle any arbitrary trace.

For the majority of the benchmarks, BAC performs worst of the three, but this is particularly noticeable in the SPEC runs. There are two reasons for this. First, instruction cache bank conflicts are the primary performance loss for BAC. Data in [12] shows that BAC is comparable to TC if bank conflicts are ignored. Second, the BAC treats basic blocks as atomic units. As a result, a BAC entry will provide only as many basic block addresses as will fit within the 16 instruction fetch limit. Given hits in both the TC and BAC, the BAC can never supply more instructions than the TC.

To summarize the major results, TC performs on average 15% better than SEQ.3 for the SPEC workload and 9% better for the IBS workload – overall, a 12% performance gain. Compared to SEQ.1, TC improves performance by 37% for SPEC, 20% for IBS, and 28% for the combined workload.

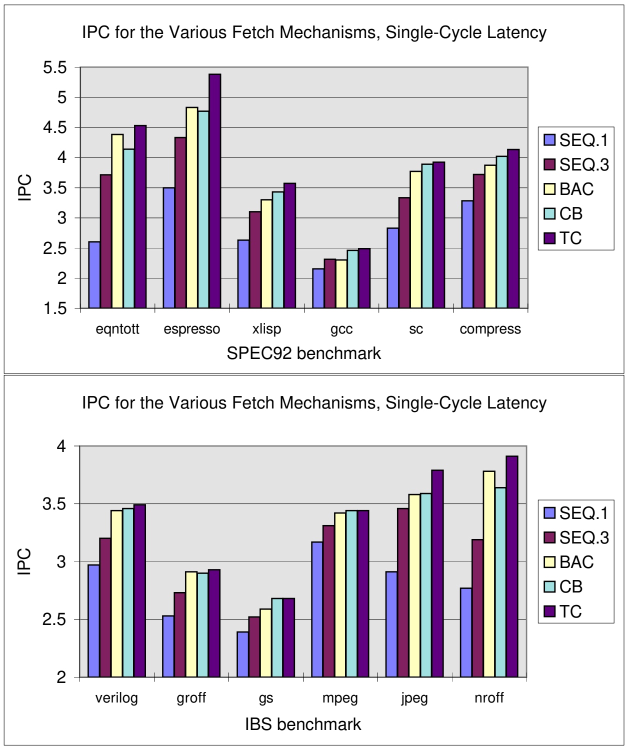

Figure 8. IPC results (fetch latency = 1 cycle).

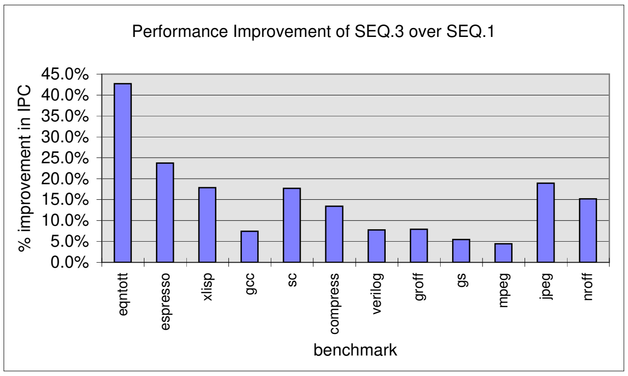

Figure 9. Improvement of SEQ.3 over SEQ.1.

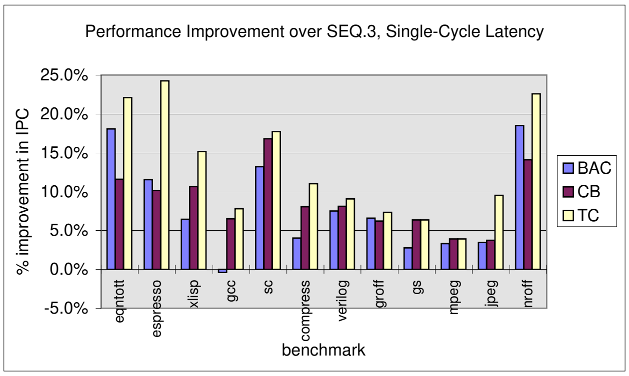

Figure 10. Improvement over SEQ.3.

5.2. The effect of latency

The effect of fetch unit latency is quantified by the graph in Figure 11. Since both CB and BAC add stages before and after the instruction cache, we give the performance of these schemes for fetch latencies of 2 and 3 cycles. CB and BAC fall well below the performance of TC. For all but 4 of the benchmarks, BAC with a latency of 2 cycles performs worse than SEQ.3. Likewise, for most of the benchmarks, CB with a latency of 3 cycles performs worse than SEQ.3.

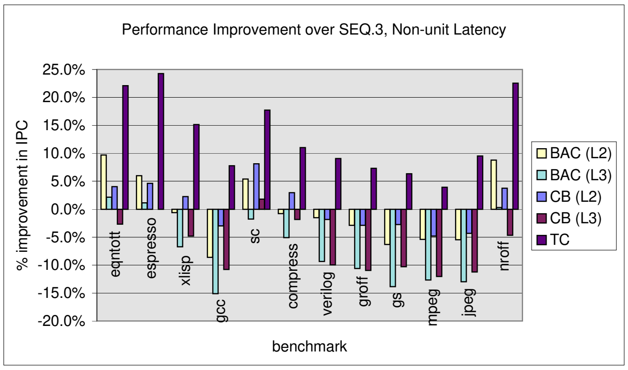

Figure 11. Improvement over SEQ.3 (*L2* and *L3* stand for 2 and 3 cycle latency.)

5.3. Trace cache effectiveness

To determine how effective the trace cache is, we establish an upper bound on its performance and measure how far short it falls from this bound. The bound is established by an "ideal" fetch model, defined as follows: as long as branch outcomes are predicted correctly and instructions hit in the instruction cache, up to 3 basic blocks or 16 instructions – whichever comes first – can be fetched every cycle.

Figure 12 shows that there is still performance to be gained by better instruction fetching. TC falls short of ideal performance due to trace cache and BTB misses. The trace cache used in the previous experiments has only 64 entries and is direct mapped; adding 2/4-way associativity or simply increasing the number of entries will narrow the performance gap between TC and ideal. Figure 12 provides incentive to explore the design space alternatives of Section 2.3 aimed at improving hit rate. To demonstrate the benefit of using a larger trace cache, we include IBS results for a 32 kilobyte, 4-way associative trace cache in the same graph.

Trace cache miss rate can be specified in two ways: in terms of traces (trace miss rate) and in terms of instructions (instruction miss rate). Trace miss rate is the fraction of accesses that do not find a trace present. Instruction miss rate is the fraction of instructions not supplied by the trace cache. Trace miss rate is a more direct measure of trace cache performance because it indicates the fraction of fetch cycles

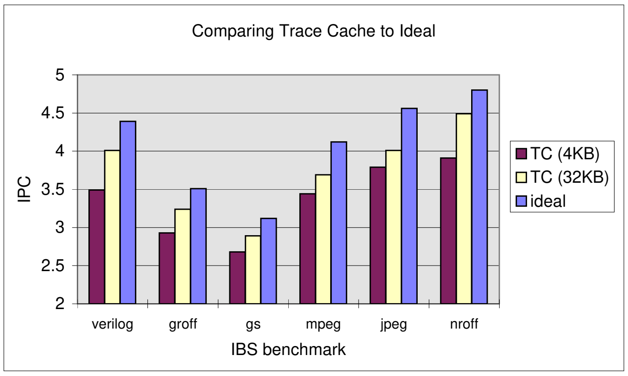

Figure 12. TC performance potential.

that benefit from higher bandwidth. However, instruction miss rate is also reported because it corresponds to cache miss rate in the traditional sense. Both trace miss rate (tmr) and instruction miss rate (imr) are shown in Table 3.

IBS	4 KB, 1-way		32 KB, 4-way		Spec	4 KB, 1-way	
	tmr	imr	tmr	imr		tmr	imr
veri	70%	48%	48%	25%	eqn	26%	8%
groff	76%	61%	60%	38%	esp	32%	14%
gs	76%	58%	60%	39%	xlisp	64%	40%
mpeg	70%	54%	51%	29%	gcc	71%	52%
jpeg	64%	43%	53%	25%	sc	50%	28%
nroff	62%	42%	45%	24%	comp	18%	6%

Table 3. Trace cache miss rates.

6. Conclusions

We have shown that it is important to design instruction fetch units capable of fetching past multiple, possibly taken branches each cycle. However, this additional bandwidth should not be achieved at the expense of longer fetch unit latency. The trace cache is successful in satisfying both of these requirements.

While a small trace cache performs well, comparison with the "ideal" noncontiguous instruction fetch model shows the potential for even higher performance. This experiment motivates investigation of larger and/or more complex trace cache designs, such as path associativity, partial matches, judicious trace selection, and victim trace caches.

7. Acknowledgements

This work was supported in part by NSF Grant MIP-9505853 and by the U.S. Army Intelligence Center and Fort Huachuca under Contract DABT63-95-C-0127 and ARPA order no. D346. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the U.S. Army Intelligence Center and Fort Huachuca, or the U.S. Government. Eric Rotenberg is funded by an IBM Graduate Fellowship.

References

[1] T. Conte, K. Menezes, P. Mills, and B. Patel. Optimization of instruction fetch mechanisms for high issue rates. *22nd Intl. Symp. on Computer Architecture*, pp. 333–344, June 1995.

[2] S. Dutta and M. Franklin. Control flow prediction with tree-like subgraphs for superscalar processors. *28th Intl. Symp. on Microarchitecture*, pp. 258–263, Nov 1995.

[3] M. Franklin and M. Smotherman. A fill-unit approach to multiple instruction issue. *27th Intl. Symp. on Microarchitecture*, pp. 162–171, Nov 1994.

[4] G. F. Grohoski. Machine organization of the ibm rs/6000 processor. *IBM Journal of R&D*, 34(1):37–58, Jan 1990.

[5] N. Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. *17th Intl. Symp. on Computer Architecture*, pp. 364–373, May 1990.

[6] D. Kaeli and P. Emma. Branch history table prediction of moving target branches due to subroutine returns. *18th Intl. Symp. on Computer Architecture*, pp. 34–42, May 1991.

[7] J. Larus. Efficient program tracing. *IEEE Computer*, 26(5):52–61, May 1993.

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

[9] J. Losq. Generalized history table for branch prediction. *IBM Technical Disclosure Bulletin*, 25(1):99–101, June 1982.

[10] S. Melvin, M. Shebanow, and Y. Patt. Hardware support for large atomic units in dynamically scheduled machines. *21st Intl. Symp. on Microarchitecture*, pp. 60–66, Dec 1988.

[11] S.-T. Pan, K. So, and J. T. Rahmeh. Improving the accuracy of dynamic branch prediction using branch correlation. *5th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, pp. 76–84, Oct 1992.

[12] E. Rotenberg, S. Bennett, and J. Smith. Trace cache: a low latency approach to high bandwidth instruction fetching. Tech Report 1310, CS Dept., Univ. of Wisc. - Madison, 1996.

[13] J. E. Smith. A study of branch prediction strategies. *8th Symp. on Computer Architecture*, pp. 135–148, May 1981.

[14] R. Uhlig, D. Nagle, T. Mudge, S. Sechrest, and J. Emer. Instruction fetching: Coping with code bloat. *22nd Intl. Symp. on Computer Architecture*, pp. 345–356, June 1995.

[15] T.-Y. Yeh. *Two-level Adaptive Branch Prediction and Instruction Fetch Mechanisms for High Performance Superscalar Processors*. PhD thesis, EECS Dept., University of Michigan - Ann Arbor, 1993.

[16] T.-Y. Yeh, D. T. Marr, and Y. N. Patt. Increasing the instruction fetch rate via multiple branch prediction and a branch address cache. *7th Intl. Conf. on Supercomputing*, pp. 67–76, July 1993.

[17] T.-Y. Yeh and Y. N. Patt. A comprehensive instruction fetch mechanism for a processor supporting speculative execution. *25th Intl. Symp. on Microarchitecture*, pp. 129–139, Dec 1992.