

Boosting Trace Cache Performance with NonHead Miss Speculation

Stevan Vlaovic
Sun Microsystems
901 San Antonio Road
Palo Alto, CA 94303-4900
(408) 616-5753

Stevan.Vlaovic@sun.com

Edward S. Davidson
University of Michigan
Advanced Computer Architecture Lab
Ann Arbor, MI 48109-2122
(734) 677-1777

davidson@eecs.umich.edu

ABSTRACT

Trace caches are used to help dynamic branch prediction make multiple predictions in a cycle by embedding some of the predictions in the trace. In this work, we evaluate a trace cache that is capable of delivering a trace consisting of a variable number of instructions via a linked list mechanism. We evaluate several schemes in the context of an x86 processor model that stores decoded instructions. By developing a new classification for trace cache accesses, we are able to target those misses that cause the largest performance loss. We have proposed a hardware speculation technique, called NonHead Miss Speculation, which removes much of the penalty associated with nonhead misses in the eight applications we studied. Performance improvements ranged from 2% to 20%, with an average speedup of around 10% across our application suite.

Categories and Subject Descriptors

Microprogram Design Aids – *optimization*

General Terms

Design, Performance

Key Words

Trace Cache, optimization, x86, branch prediction

1. Introduction

With increasing semiconductor density, we are able to put more resources on a single microprocessor chip than ever before. However, by putting more resources on a chip and demanding ever higher processor clock speeds, the distribution of data becomes an increasingly difficult problem. One method of increasing clock speeds is by increasing the pipeline depth of a processor, thereby reducing the amount of useful work done per stage. Furthermore increasing the pipeline depth increases both the logic complexity of the processor and the cache miss and mispredicted branch penalties when measured in terms of processor cycles or lost instruction headway.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

ICS'02, June 22-26, 2002, New York, New York, USA
Copyright 2002 ACM 1-58113-483-5/02/0006...\$5.00.

For x86 processors, it has been suggested to remove x86 instruction fetch and decode stages from the pipeline to alleviate some of the impact of increased pipeline depth. Recent x86 processors decompose x86 instructions into smaller, more RISC-like operations called μ ops. A trace cache that caches decoded x86 instructions (μ ops) along recently predicted paths of execution can be used to remove the penalty of decoding x86 instructions while removing much of the burden from the branch prediction hardware. In this work, we measure the effects of such a trace cache on the performance of modern x86 processors running some common applications.

Traditionally, trace caches have had lines sizes equal to the processor issue width; in such configurations the trace cache addresses the multiple predictions per cycle required when issuing more than a few instructions (e.g. 16 instructions). In the context of this paper, the trace cache not only addresses the multiple predictions per cycle constraint, but also the additional decoding overhead associated with recent x86 processors. To facilitate these goals, we evaluate a trace cache design that does not have fixed sized (e.g. issue width) traces, but rather has variable length traces that depend on the run time behavior of the program. A trace may consist of multiple segments, each of which is the size of one cache line. This trace cache model has trace segments connected to each other via pointers, analogous to a linked list in software design. This design is based upon the work presented in [3]. Since this type of trace cache has not been evaluated previously, it is our goal to evaluate and improve upon this type of trace cache design.

2. Related Work

The first structure that could be labeled a precursor to the trace cache was described in [4] which proposes a fill unit to compact a fetch block's worth of instructions into an entry in a Decoded Instruction Cache (DIC). A hit in the DIC results in a larger atomic unit of work than would be possible if the individual instructions were fetched and decoded singularly. Yeh, Marr, and Patt[11] then consider a fetch mechanism that provides high bandwidth by predicting multiple branch target addresses every cycle. They extend the Branch Target Buffer (BTB) to create a Branch Address Cache (BAC) which allows multiple predictions to provide the starting addresses of the next several predicted targets. These addresses are then fed into a highly interleaved instruction cache to fetch several basic blocks per cycle. In [1], Dutta and Franklin build on the BAC concept, but instead of making multiple predictions per cycle, they make a single path prediction. This eases timing constraints and makes for a less complex BAC.

The trace cache idea was patented in 1994 by Peleg and Wieser[6]. Rotenberg, Bennett, and Smith[7] further refine the trace cache concept and present a thorough comparison between the trace cache scheme and several other hardware assisted high bandwidth fetch schemes, such as the BAC. They found that the trace cache outperformed all of the previously proposed hardware assisted schemes, both in terms of overall performance and latency. Their machine model had an optimal backend, and a 128 KByte instruction cache with a 4 KByte trace cache, and boosted average performance by 28% over the SPECint92 and Instruction Benchmark Suite (IBS). In their study, they proposed several extensions to the original trace cache idea, such as caching multiple paths starting from the same basic block, partial matching of traces, alternative indexing methods, judicious trace selection (prioritized based on trace usefulness), and even a victim trace cache.

Friendly, Patel, and Patt[2] further refine the trace cache idea with the inclusion of partial matching and inactive issue. Partial matching provided a 12% improvement over the SPECint95 benchmarks, while inactive issue (issuing instructions from a trace regardless of whether they match a prediction) increased performance by 15%. Patel[5] further investigated path associativity, branch promotion, and trace packing which resulted in an overall performance improvement of 14% over an aggressive Icache on the SPECint95 benchmarks. A main difference in philosophy between Rotenberg [7] and Patel [5] is that Rotenberg viewed the trace cache as an assist to the conventional instruction cache, while Patel viewed the trace cache as the main instruction fetch mechanism with the conventional instruction cache acting as an assist.

3. Applications

We have chosen eight popular Windows NT applications to help evaluate our x86 trace cache designs: Id's Doom, FileMaker Pro 5.0, Microsoft Explorer 5.0, Microsoft Visual Studio 5.0, Netscape 6.0, RealPlayer 8.0, Winamp 2.72, and Winzip 8.0. Doom is one of the early first-person type combat games and is available as shareware. The run of Doom included recording a session of a Doom game, and then replaying it on the simulator. FileMaker Pro 5.0 is a database application that allows users to easily share their data over the internet. This run consisted of performing multiple searches and sorts on a 3,000 entry database. Explorer 5.0 is Microsoft's web browser; our input is a set of three .htm pages. The first is the CNN web page, the second is an ESPN web page, and the third is University of Michigan's EECS homepage. Microsoft Visual Studio 5.0 (MsDev) is a code development environment, with 5.0 being the previous release. Our run of Visual Studio involved the compilation of **go** from the SPEC95 benchmark suite. Netscape 6.0 is another popular web browser; the same web pages that were loaded on Explorer were also used for Netscape. RealPlayer 8.0 is a video player that can be used to play a number of different video formats with the second episode of SouthPark being used as its data input. Winamp 2.72 is the latest release of a popular mp3 player; its input was "Cool Down Daddy" by Jellyroll. Winzip 8.0 is a compression and decompression engine that can handle multiple formats. Our run of Winzip entailed compressing the source files of **go** (from SPEC95). Table 1 highlights some dynamic characteristics of the applications. In the Instructions per Branch column, L_{OO}P and REP instructions are included as branches.

Table 1. Application Trace Characteristics

	Insts (x10 ⁶)	μops/ Inst	Insts/ Store	Insts/ Load	Insts/ Branch
Doom	388	1.52	4.61	2.64	5.40
Explorer	396	1.50	4.53	2.55	4.77
FileMaker	447	1.49	4.61	2.85	5.04
MsDev	469	1.46	4.94	2.63	4.07
Netscape	377	1.52	4.53	2.65	4.33
RealPlayer	522	1.44	5.44	2.76	4.63
Winamp	456	1.47	4.11	2.31	6.88
Winzip	302	1.44	5.74	3.05	5.63
Average	420	1.48	4.81	2.68	5.09

4. Machine Model

The trace cache in our machine model is based upon the linked list style of trace cache described in [3]. It stores sequences of μops (including taken branches) as sequential strings of μops, thereby eliminating the x86 instruction fetch and decode latencies when a stored sequence of μops is reexecuted. Our model's trace cache can hold 12K μops; x86 instruction address references within this cache are adjusted to point to the address of the target μop in the trace cache. This instruction address mapping effectively creates a new address space for the translated code that is local to the CPU core and is not externally visible, i.e. it is not architecturally visible in the x86 ISA. The goal of this L1 instruction cache is to remove x86 instruction fetch and decode from the main execution loop.

The trace cache serves as the primary (L1) instruction cache and delivers up to 4 μops per clock to the out-of-order execution logic. Only when there is a trace cache miss does the processor fetch and decode x86 instructions that it obtains by accessing the unified L2 cache.

The decoding of x86 instructions into μops is difficult, since the instructions have a variable number of bytes and have many different fields that can be interpreted in numerous ways. When a branch is mispredicted, the recovery time is shorter if the machine does not have to redecode the instructions from the correct target location. By using the trace cache to store μops of previously decoded x86 instructions, we can reduce branch misprediction penalties by bypassing the x86 decoders whenever the correct target instructions's μops reside at the head of a trace in the trace cache.

In addition to storing the decoded x86 instructions, the trace cache also assembles the decoded x86 instructions into sequences of μops called traces. One trace may span several basic blocks; the predicted path of program execution is assembled into one sequence of μops that are stored in the trace cache as a linked list of trace cache lines. A trace cache line can hold up to 6 μops and a single trace can contain up to 64 lines, but will terminate early whenever a call, a return, or an indirect branch is encountered. The capacity of the trace cache is 12 K-μops, i.e. 2048 trace cache lines.

There are two distinct modes of trace cache operation: *trace cache execution mode*, and *trace segment building mode*. A trace consists of a head, zero or more body segments (trace cache lines), and a tail. These segments are assembled in a doubly linked list, with the head segment comprising the first trace cache line of the list, the body segments linked in between, and the tail segment being the last trace cache line of a given trace. If a trace has only one segment, then the head and the tail point to the same segment, and there are no body segments. When an x86 instruction is searched for in the trace cache, the search is performed *only* on the first instruction address of the trace heads; if the μ ops for that instruction are resident in the trace cache, but reside only in a body segment, then the trace cache search will miss.

The TAXI infrastructure[9] provides a cycle accurate simulator for applications that run on x86 processors. We use this infrastructure, with the baseline configuration parameters shown in Table 2, to gather our results.

Table 2. Baseline configuration parameters

Parameter	Model 4
Physical Registers	128-INT, 128-FP
Streaming Buffer Size	64 bytes
Decode width	64 bytes
# Complex Decoders	1
# Simple Decoders	0
Decoded Instruction Queue	8 μ ops
μ op Decode Width	8 μ ops
μ op Issue Width	4 μ ops
μ op Commit Width	4 μ ops
Functional Units	4 IntALU, 1 IntMult/Div, 4 FP ALU, 1 FPMult/Div, 2 MemPorts
ROB	126 μ op entries
Branch Predictor	2 level, 4 K-entry
BTB	4 K-entry, 4-way
Return Address Stack	32 entries
Trace Cache BTB	512 entries, 4-way
Mispredict penalty	Variable (no wrongpath execution)
L1 D-cache	8 KB 4-way
L1 I-cache (Trace Cache)	12 K- μ ops, 4-way
L2 Unified	256KB 8-way, 7 cycle latency
TLB size	32 Instruction/64 Data
Memory	110 cycle latency

The number of architected registers is the standard x86 register definition (8 general purpose registers and 8 floating point stack registers, plus control registers), but the number of physical registers in our model is 128 integer and 128 floating point.

In our model, if there is a miss to the trace cache, the L2 cache is accessed. On an L2 cache hit, the cache line is brought into a buffer where the cache line is parsed into x86 instructions. Once the instruction boundaries are determined, the instructions are decoded using the x86 decoders (our baseline model has just 1 decoder, as seen in Table 2). Once decoded, the resulting μ ops are placed in the decoded instruction queue (DIQ). Note that the trace cache also places μ ops in the DIQ, but avoids the x86 instruction fetch and decode. Except for the trace cache BTB, the branch prediction parameters in Table 2 refer to the branch prediction mechanism in the x86 instruction fetch and decode stages. A block diagram is shown in Figure 1.

Since the x86 fetch and decode portion of the pipeline is before the trace cache (i.e. its L1 instruction cache), the argument could be made for grouping the L2 access and the x86 fetch and decode into an ordinary L1 instruction miss handler. A trace cache miss would then simply incur the time required to service an L1 miss, rather than the additional overhead of including a longer pipeline. However, this would obscure the dynamics of the front end of the processor. To attribute the various penalties associated with trace cache misses appropriately, our model has two pipelines, one with twenty stages that is used when the trace cache is in trace execution mode, and another that is twenty stages plus the stages required for x86 fetch and decode for use when the trace cache is in trace segment building mode.

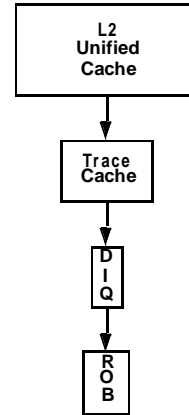


Figure 1. Model Block Diagram

On a trace cache hit, a μ op will see a twenty stage pipeline. However, on a trace cache miss, an X86 instruction will have to be fetched from the L2 cache, the instruction boundaries marked, and the instruction aligned, decoded, and placed in the decoded instruction queue. In our model, we add nine stages for doing this, we place them before the trace cache and activate them when in trace segment building mode. With 9 added stages, the μ ops of an instruction that misses in the trace cache would see 29 pipeline stages before completion.

The line replacement policy in the trace cache may also greatly impact performance. Normally, LRU replacement is used in conventional set-associative caches; however, since body and tail segments are not looked up via their addresses (they are linked to previous segments via pointers), they have the flexibility of being placed anywhere in the cache. The head of a trace must be placed in the set to which it maps so that it can be accessed by address. LRU replacement, as in a conventional cache, is used for allocating a head segment. A different policy, called *tail*

replacement, is used for allocating body and tail segments: a set is chosen as for head segments, but the trace containing the LRU element of this set is then followed to its tail, and the line containing that tail segment is replaced. We tried other schemes such as a more nearly global LRU scheme and a random replacement policy, but our tail replacement scheme performed best overall.

5. Trace Cache Access Classification

Figure 2 classifies the possible outcomes of a trace cache (TC) access. In order to characterize the operation of the trace cache, this classification delineates the various possibilities that might constitute a "trace cache miss." The process begins with a *BTB Access*. If the BTB returns the correct address, then this address is used to index the trace cache, called a *TC Head Lookup*, which either hits or misses. Note that a BTB hit can produce the wrong address and still hit in the trace cache, which is clearly undesirable. A trace head hit results in μ ops being inserted into the decoded instruction queue (DIQ); a trace head miss will eventually result in invoking the conventional x86 fetch and decode pipeline if the BTB provided the correct address, or in a new head lookup if the BTB provided the wrong address. If the correct execution path diverges before the end of the complete trace stored in the trace cache, then we will have a *nonhead miss*. Nonhead misses, as shown below, significantly degrade performance.

Only after a TC Head Lookup misses does the trace cache transition from *trace cache execution mode* (i.e. fetching μ ops from the trace cache) to *trace segment building mode* (i.e. building traces and storing them into the trace cache), and then commence to search the L2 for the missing instruction. A line with the instruction is then loaded from the unified L2 cache, the instruction boundaries are marked, the instruction is aligned for the one decoder, and finally the instruction is decoded. A trace cache miss can thus be very costly relative to an L1 instruction miss in a processor that doesn't have a trace cache.

Figure 2 shows that a trace cache access can be classified as one of eight types which have different ramifications on overall performance. For instance, a TC body miss (nonhead miss) followed by a TC head miss will incur the μ op misprediction penalty, which averages 17.2 cycles over our 8 applications, and will then have to go through the 29 stage pipeline before the instruction execution is complete. Obviously, we want to eliminate as many misses as possible, but since certain types of misses have a higher cost, we should focus primary effort on those that contribute the larger run time penalties. Figure 2 shows the 8 cases of the TC access classification, namely:

- 1) BTB hit with correct address followed by a TC head hit
- 2) BTB hit with correct address followed by a TC head miss
- 3) BTB hit with wrong address followed by a TC head hit
- 4) BTB hit with wrong address followed by a TC head miss
- 5) Nonhead miss followed by a TC head hit
- 6) Nonhead miss followed by a TC head miss
- 7) BTB miss followed by a TC head hit
- 8) BTB miss followed by a TC head miss

Each of the 8 cases is concerned with the results of an initial event as well as the following TC head lookup. For instance, it is useful

to know how many nonhead misses are followed by trace cache head hits; the penalties are calculated differently for each case.

The *TC Head Lookup* returns a hit in Cases 1, 3, 5, and 7, a miss in Cases 2, 4, 6, and 8. Which of these cases applies depends on how the trace cache arrived at the *TC Head Lookup* state. For instance nonhead misses result in Case 5 or 6, depending on whether the subsequent TC Head Lookup is a hit or miss, respectively. BTB hits and misses are decomposed similarly, as indicated in Figure 2.

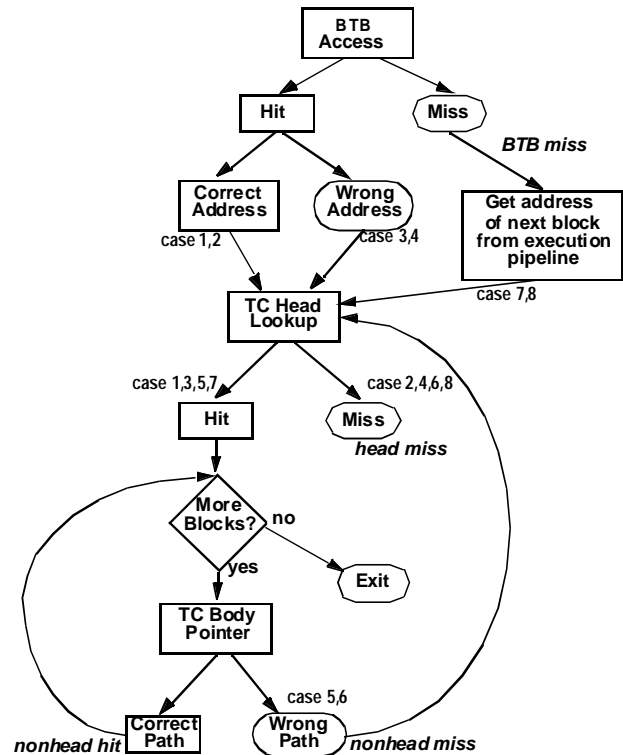


Figure 2. Trace Cache Access Classification

Case 1 1 cycle	Case 3 (17+1) cycles	Case 5 (17+1) cycles	Case 7 (17+1) cycles	Head Lookup Hit
Case 2 (1 + x86 Fetch) cycles	Case 4 (17+1+ x86 Fetch) cycles	Case 6 (17+1+ x86 Fetch) cycles	Case 8 (17+1+ x86 Fetch) cycles	
BTB Hit Correct Address	BTB Hit Wrong Address	Nonhead Miss	BTB miss	

Figure 3. TC Access Classification Cost Model

The penalty associated with each case is shown in Figure 3. Three different penalty terms appear in Figure 3. Each case costs 1 cycle for the trace head lookup. Next is the 17 cycles charged for a μ op misprediction, which occurs whenever the address of the next instruction must be fetched from the execution pipeline once it becomes available. This term appears in cases 3 through 8. The last term is "x86 Fetch" representing a variable penalty that occurs whenever there is a miss on a TC head lookup. The penalty associated with a trace head miss is a lookup in the L2 cache, x86 instruction decode, and finally trace cache fill. Note that in addition to these operations, the execution pipeline for an instruction that must be fetched from the

L2 cache is 29 stages, rather than the 20 stage pipeline seen by an instruction that is found in the trace cache.

As seen in Figures 2 and 3, Case 5 occurs when there is a nonhead miss (a miss at some trace segment other than the head), followed by a subsequent trace cache hit when this segment is looked up as a head. This means that an instruction along the correct execution path that diverges from the trace segment path is later identified as a trace cache head; the penalty is the μ op misprediction penalty of roughly 17 cycles. A nonhead miss can arise because of two different reasons. The first is due to a mispredicted branch within the trace, and can be thought of as a nonhead misprediction. The second reason is more subtle; if the segment within the trace has been replaced by a segment of some other trace, then this too will cause a nonhead miss.

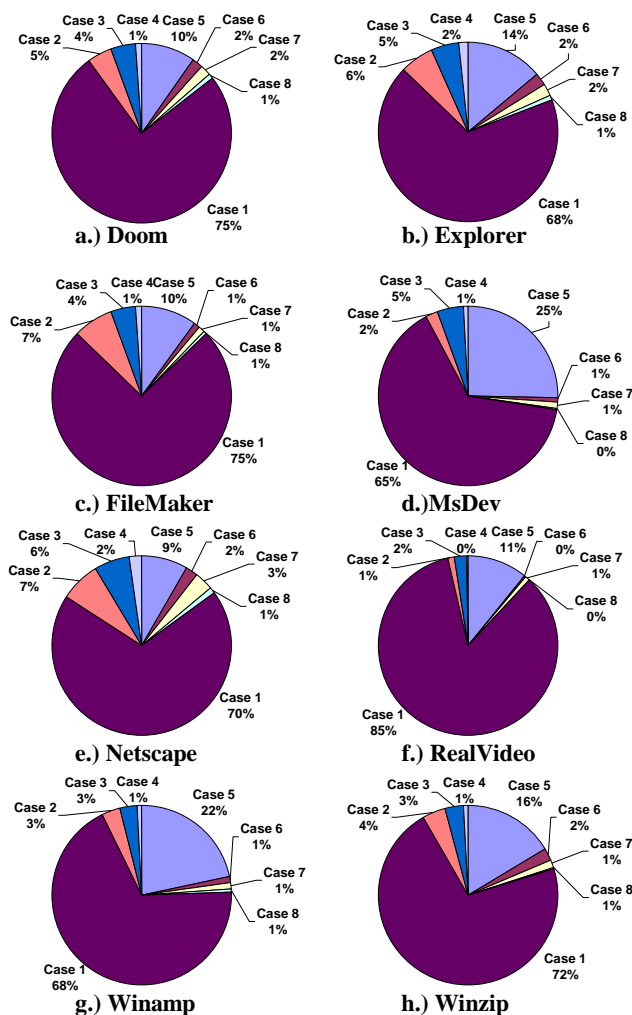


Figure 4. Trace cache head lookup classification breakdown across all eight Win32 Applications

Case 6 also involves a nonhead miss, but the eventual correct head lookup causes a trace cache miss. This case (as well as cases 4 and 8) has the highest penalty: first, it takes 17 cycles to determine that there is a nonhead miss, then 1 cycle to perform a head lookup, followed by access of the L2 cache, which requires 7

cycles. So a total of 25 cycles are required before even inserting this instruction into the 29 stage pipeline, assuming an L2 cache hit. Note that the correct address is generated only after the processor determines that there is a μ op misprediction.

For Cases 3, 4, 5, and 6, a real processor would perform wrong path execution until the correct address of the next block arrives from the execution pipeline. Since TAXI does not perform wrong path execution, the instruction (or μ op) fetch simply stalls until the correct next block address arrives, which took an average of 17.2 stall cycles in our application simulations. Therefore, for our simulation, we only need penalties for Cases 1, 2, 7 and 8 (Cases 3 and 5 fold into case 7, and cases 4 and 6 fold into case 8).

Clearly Case 1 is the most desirable; we stay in the trace cache execution stage and keep fetching μ ops down the correct execution path. Case 2 is not desirable, as we must start fetching from the L2 cache after the head miss and must then use the full 29 stage pipeline which includes the x86 fetch and decode stages. In Case 2, as the BTB hit gives us the correct target address in one cycle, the processor can then immediately issue the fetch request to the L2 cache. Thus there is no μ op misprediction penalty.

Case 7 (likewise 3 and 5) is clearly undesirable as the BTB misses (or provides the wrong address to the trace cache), although after the target instruction's address is determined from the processor pipeline, it does hit in the trace cache. For example, in Case 3 μ op fetch stalls until the branch is eventually resolved in the execution stage, at which point the branch misprediction is detected and the trace cache is presented with the correct address; the penalty for this case is the μ op misprediction penalty, roughly 17 cycles.

Case 8 (likewise 4 and 6) is worse than Case 7; the BTB misses (or provides the wrong address) as above, but once the correct address is determined, the head lookup misses in the trace cache. After that miss, the conventional x86 fetch and decode pipeline is enabled and the L2 cache is probed with the predicted address generated by the front end predictor.

With this classification in mind, the distribution of trace cache head lookups among the eight cases above was measured for each application. The results are shown in the pie charts of Figure 4. A clear majority of the accesses are Case 1 (BTB hit with the correct address, followed by a trace cache hit), varying from 85% for RealVideo to 65% for MsDev which is good, but shows room for improvement. Summing up the percentages for Case 2, Case 4, Case 6, and Case 8 yields the total percentage of head accesses that result in a trace cache miss when the correct head is looked up, which varies from 1% for RealVideo to 12% for Netscape. To limit the occurrences of these cases (2, 4, 6, and 8), we might try making the trace cache larger, or we could conserve space by organizing the traces in a different manner to avoid replication. Although we have not directly measured the amount of replication (copies of the same block in different traces) in the trace cache, other studies have indicated that it can be a significant factor [5][7].

Figure 4 shows that a significant percentage of the accesses to the trace cache are nonhead misses (Cases 5, 6). Interestingly, the number of nonhead misses that subsequently miss as trace heads (Case 6) constitutes no more than 2% of all head lookups in any of the applications studied, thus a large majority of the nonhead misses subsequently hit in the trace cache as trace heads, once the correct address is produced. For MsDev 25% of its trace cache

head lookups are Case 5, Winamp has 22%, and Winzip has 16%; Netscape has only 9%, but in every application Case 5 is the second most common case (after Case 1).

6. Nonhead miss handling

In most of the applications we studied, a majority of all misses (i.e. head accesses, excluding Case 1) are nonhead misses (cases 5 and 6). We therefore decided to target nonhead miss accesses for optimization. The bars in Figure 5 labeled *Opt NHMS* show the impact of removing the entire penalty associated with nonhead misses. In Netscape, optimal nonhead miss handling is only about 3% faster; however, recall that Netscape also had the lowest percentage of nonhead misses (only 11%). In MsDev, Winamp, and Winzip, which have a significantly higher percentage of nonhead misses, the performance improvement yielded by *Opt NHMS* is 20%, 15% and 10%, respectively. Only 11% of RealVideo’s head accesses are nonhead misses, but Case 1 and Case 5 together comprise 96% of all head accesses to the trace cache which allows RealVideo to achieve the second highest improvement at 16%. Although the variance is somewhat high, the average improvement over all the applications is around 12%. By eliminating most (or all) of the nonhead miss penalty, we could achieve a significant performance improvement over the baseline for at least some important applications.

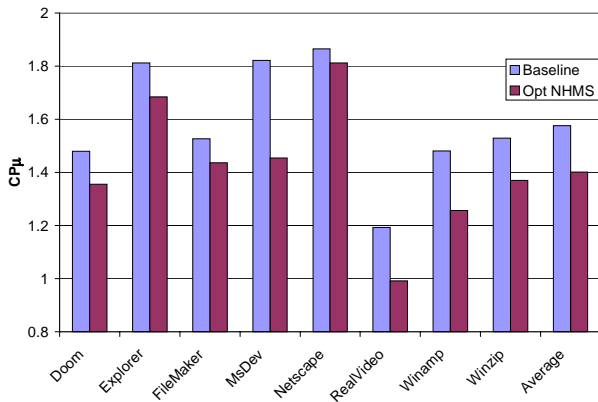


Figure 5. Comparison of baseline performance to a baseline configuration with optimal nonhead miss speculation

In this section we propose a new prediction mechanism to help alleviate the nonhead miss penalty of the trace cache. This mechanism employs a variation on conventional branch prediction hardware. It is similar to other branch prediction mechanisms, but is made to differ since our requirements are somewhat different. Note that predicting a nonhead miss when there is in fact no nonhead miss will incur a significant penalty over the baseline, which correctly continues along the trace. In particular, whenever the trace cache provides a correct multiblock trace for execution (i.e. no penalty), but the new prediction mechanism predicts a nonhead miss somewhere within that trace, it incurs the 17 cycle μ op misprediction penalty plus the 1 cycle head lookup, and if that head lookup misses, an even larger penalty will be incurred by causing an unnecessary transition to trace segment building mode.

By contrast, whenever the prediction mechanism fails to predict a nonhead miss when there is one, it simply does not improve on the baseline penalty; both systems will incur the same 17 cycle μ op misprediction penalty, plus a 1 cycle head lookup and a

possible transition to trace segment building mode. This failure does, however, represent a lost opportunity to improve performance.

With this bias in mind, we designed a speculative nonhead miss prediction mechanism to alleviate nonhead miss penalties. The trace cache and DIQ portion of our baseline processor model’s datapath is shown on the left side of Figure 6. Our nonhead miss prediction mechanism, sketched in the dashed box on the right, is comprised of two parts: its *Nonhead Miss Predictor* does the actual prediction, and its *Nonhead Miss Target Buffer* supplies the target address.

The mechanism works by passing the EIP of every instruction to be accessed (except those at the end of the trace) to the predictor. This address is then converted to an index into a table of 3-bit saturating counters, as described in Section 6.1. If the accessed counter indicates that this instruction will result in a nonhead miss, then the EIP is passed to the Nonhead Miss Target Buffer (NMTB) for target address prediction, as described in Section 6.2. The NMTB is not searched unless the nonhead miss predictor predicts that a miss will occur. Currently the EIP of every instruction (except those at the end of a trace) is submitted to the Nonhead Miss Predictor for checking, but this constraint could potentially be relaxed, since a very high percentage of the nonhead misses occur on a very small set of addresses. For example, in MsDev only about 2% of all nonhead miss addresses (static count) account for 90% of all nonhead misses (dynamic count).

6.1 Nonhead Miss Predictor

The Nonhead Miss Predictor in Figure 6 predicts whether or not the instruction at the given address (EIP) will cause a nonhead miss. The Nonhead Miss Predictor comprises an indexing scheme and a table of saturating counters, as shown in Figure 7. Recall that if the predictor wrongly predicts a nonhead miss (i.e. the next block in the current trace in the baseline trace cache is in the correct path), then additional overhead is incurred over the baseline model; however, if the prediction predicts that there is no nonhead miss when in reality there is one, then no additional overhead is incurred (we simply miss an opportunity to gain performance over the baseline). For this reason, the predictor should be biased toward the conservative side when predicting nonhead misses.

To determine the number of bits to use in the saturating counters, we simulated an infinite size table of n-bit saturating counters, with the EIP as the selection mechanism (i.e. each instruction had an n-bit counter in the table). Initially when designing the Nonhead Miss Predictor we started out with 2-bit saturating counters (with counter state 2 and 3 predicting taken), but these predicted too many nonhead misses when in fact the instruction stream followed the execution trace (i.e. too many false positives). We subsequently tried 4, 5, 6, and even 7-bit saturating counters (again with the highest two counter states predicting taken), each of these virtually eliminated false positives, but unfortunately did not predict enough of the true nonhead misses and thus failed to capture a reasonable fraction of the performance improvement potential indicated by *Opt NHMS* in Figure 5. We found that 3-bit saturating counters worked best overall.

After choosing 3-bit saturating counters, a suitable indexing mechanism for identifying which 3-bit counter to use had to be

found. Initially, just the EIP of the instruction was used to index a fixed sized (64 K-entry) table, but this provided too many conflicts; a more complex scheme had to be determined. A number of different selection mechanisms were tested, but the one that proved to provide the best overall performance, shown in Figure 7, used a 16-bit nonhead history register XORed with the lower 16-bits of the EIP, while the upper 16-bits of the EIP are passed through.

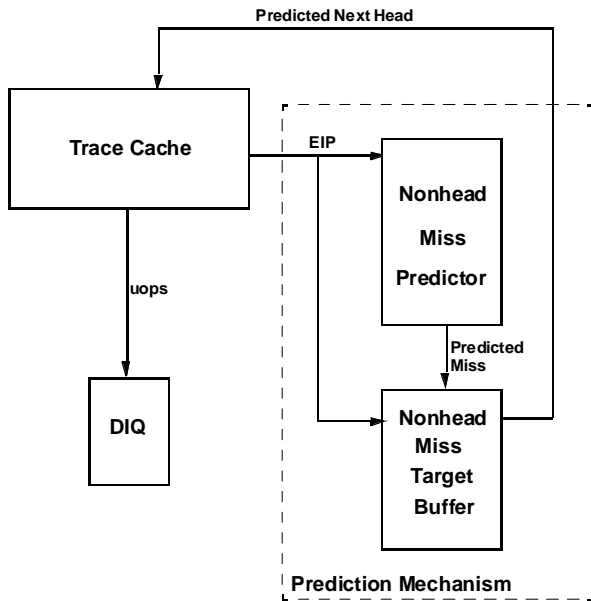


Figure 6. Nonhead miss prediction mechanism

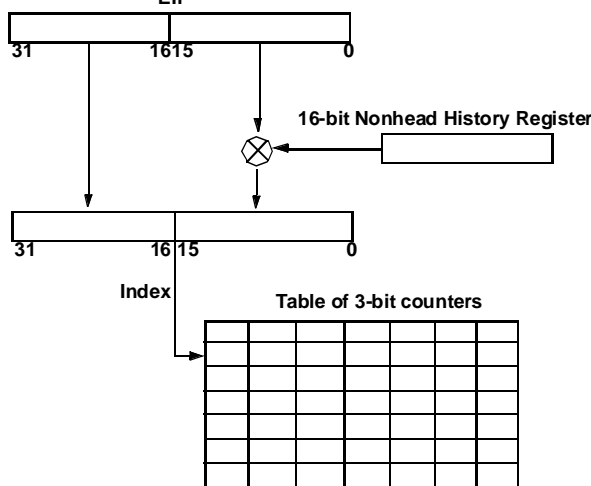


Figure 7. Nonhead Miss Predictor Organization

For each application, with a 64 K-entry table of 3-bit counters, Table 3 shows the percentage of nonhead misses that are correctly predicted (true positives), and the percent that are not predicted (false negatives, which amount to incorrect predictions of trace execution). Note that these add up to 100%, i.e. all the nonhead misses. The last column shows incorrect predictions of nonhead misses (false positives, for which the execution actually follows the cached trace). The percentages in this column are calculated two ways, first as a percentage of actual nonhead misses (as for the other columns), and secondly (in parentheses) as a percentage

of the total number of instruction addresses (EIPs) that are used to index the predictor.

The predicted nonhead misses (true positives) are the cases where we improve on the baseline; the not predicted nonhead misses (false negatives) incur no extra penalty relative to the baseline, but miss out on a chance to improve performance; the nonhead hits that are predicted to miss (false positives) incur a penalty relative to the baseline. We chose to use 64K counters since they provided good coverage without excessive hardware cost.

In MsDev, for example, our nonhead miss predictor correctly classifies 85.8% of the nonhead misses as nonhead misses, but the remaining 14.2% of the nonhead misses are incorrectly predicted as hits (i.e. as staying on the path in the execution trace); the number of predicted nonhead misses that are not actual nonhead misses are only 2.14% of the total actual nonhead misses, or 0.22% of all instructions that are used to index to the predictor. Our predictor tends to perform better on applications with more nonhead misses in their traces, yielding both a higher percentage of correctly predicted misses and a lower percentage of nonhead hits that are predicted to miss.

Table 3. Percentage accuracy of the nonhead miss predictor (64 K-entry, 3-bit saturating counters)

Application	Nonhead Misses		Nonhead Hits
	Predicted	Not Predicted	Predicted to Miss
Doom	65.9	34.1	5.28 (0.27)
Explorer	65.2	34.8	4.67 (0.24)
FileMaker	68.3	31.7	6.16 (0.24)
MsDev	85.8	14.2	2.14 (0.22)
Netscape	46.2	53.8	7.17 (0.28)
RealVideo	88.6	11.4	2.36 (0.13)
Winamp	83.1	16.9	2.29 (0.16)
Winzip	76.9	23.1	3.56 (0.19)
Average	72.5	27.5	4.20 (0.22)

6.2 Nonhead miss target buffer

In the previous section we discussed the mechanism used to predict which instructions will be nonhead misses. In this section, we discuss a mechanism for predicting the address of the instruction that first diverges from the path provided by the trace cache. The address of the last instruction that lies on the trace provided by the trace cache is used as an index; the target is the address of the instruction that is predicted to be a nonhead miss. This predicted nonhead miss address is required to access the trace cache as a subsequent head lookup. This mechanism is analogous to the BTB in traditional branch prediction, which provides the target address of a predicted branch. Initially, we used a table capable of storing multiple targets (i.e. nonhead miss addresses) per index, but this proved to be excessive since the majority of nonhead misses for a given EIP were to a single target (i.e. there is often only one nonhead miss target address for a given EIP in a given trace in the trace cache). For our experiments, we used a 4K entry Nonhead Miss Target Buffer

(NMTB), whose hit rates for each application are shown in Table 4.

Table 4 shows that the hit rates of our 4 K-entry NMTB are very high, and the applications that have the most accesses (most predicted nonhead misses) to the NMTB also have the highest hit rates. The hit rates in Table 4 show that a simple last address seen mechanism is sufficient to capture the behavior of the nonhead misses; a more complex scheme (e.g. caching multiple targets) is not necessary.

Table 4. Target instruction hit rate in a 4K entry NMTB

Application	Hit rate
Doom	97.7
Explorer	98.0
FileMaker	98.3
MsDev	99.2
Netscape	96.3
RealVideo	99.4
Winamp	99.3
Winzip	98.7
Average	98.4

7. Application Performance with NHMS

To assess NHMS performance, we added the 64K entry nonhead miss predictor to our baseline configuration and initially assumed a perfect NMTB; this model is called *64k pred*. To assess the effect of a practical NMTB, we created a model called *64k pred/4k NMTB*, in which the perfect NMTB of *64k pred* is replaced with a 4K entry NMTB. The resulting application performance is shown in Figure 8. The first two bars are the same as in Figure 5. The 64K entry predictor adds an average of only 0.023 CPμ (1.67%) over the optimal case. We see that *64k pred* CPμ lies only 13% of the way from perfect miss speculation (*Opt NHMS*) to the baseline. On average the 64K entry predictor reduces the baseline CPμ by 10%.

Replacing a perfect NMTB with the 4K entry NMTB adds negligible CPμ to *64k pred*. The 4K entry BTB adds an average of only 0.001 CPμ (0.07%) to the CPμ of *64k pred*. Furthermore we see that *64k pred/4k NMTB* CPμ lies only 20% of the way from the perfect miss speculation (*Opt NHMS*) to the baseline.

The *64k pred/4k NMTB* implementation of our NonHead Miss Speculation technique reduces the baseline CPμ by an average of 10% (vs. about 12% for perfect NonHead Miss Speculation), thus it eliminates most of the nonhead miss performance penalty. This number, however, varies widely among applications from only a 2% CPμ reduction for Netscape, to MsDev at 20%, RealVideo at 15%, and Winamp 14%. Note, however, that Netscape does not appear to suffer much from nonhead misses; even *Opt NHMS* reduces its CPμ by only 3%. Another way to measure of effectiveness of NHMS is to measure the percent reduction in the nonhead miss penalty. A graph of these percentages is shown in Figure 9. Note that the applications that remove the largest fractions of the nonhead miss penalty also have the most nonhead misses. Our NHMS implementation removes 95% of the nonhead

miss penalty in MsDev, 89% in Winamp, and 88% in RealVideo; it does least well on Doom at 67% and Netscape at 62%.

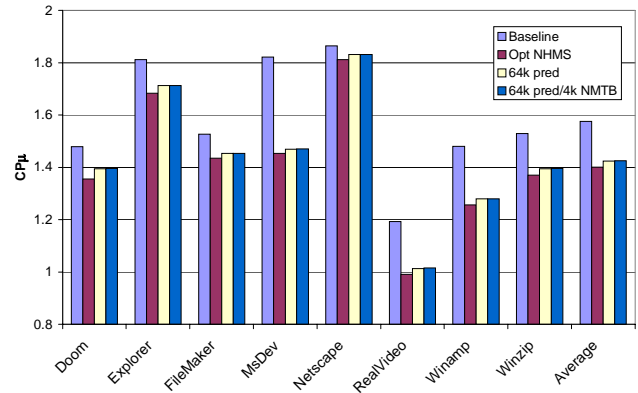


Figure 8. Comparison of baseline performance to that of Opt NHMS, 64K pred, and 64K pred/4K NMTB

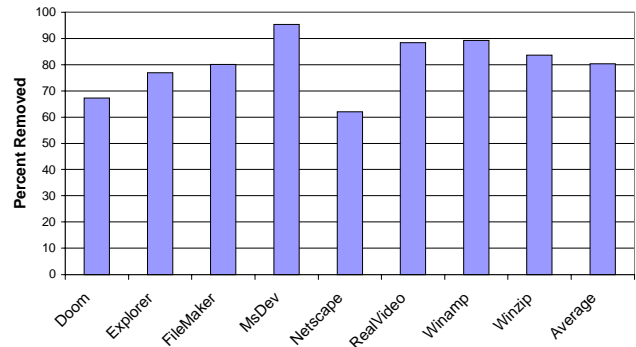


Figure 9. Percent of the nonhead miss penalty removed by NonHead Miss Speculation (64k pred/4k NMTB)

In Figure 10, we sum up all instances of each of the 8 cases over all of the applications, and present the trace cache access classification for the baseline with and without NHMS (*64k pred/4k NMTB*). Across all eight applications, Case 5 has been drastically reduced. For MsDev (not shown), Case 5 has been reduced from 25% to 4%. Other applications do not show such large changes, but even Netscape has a reduction in the percentage of Case 5 from 9% to 4%. In all of the applications, most of the accesses that were labeled Case 5 or Case 6 are now Case 1, and there is hardly any measurable increase in the number of the other (penalty causing) cases. In MsDev, for example, Case 1 (BTB hit followed by a head hit) goes from 65% to 83%. Not all of the applications enjoy such a large improvement, but every application experiences an increase in the percentage of its accesses that are categorized as Case 1.

As seen in Figure 10, NHMS reduces the incidence of Case 5 on average from 14% to 3%, and also increases Case 1 from 74% to 84%. The only other visible change occurs in case 7 where NHMS increases the number of BTB misses that later hit in the trace cache to 2% (from 1% in the baseline). This increase is due to the change in the traces that are stored in the trace cache.

For instance, if NHMS incorrectly predicts a nonhead miss, then the baseline with NHMS exits the current trace and performs a trace head lookup using the address delivered by the NMTB. If

there is no trace in the trace cache with this starting address, then the baseline with NHMS transitions to trace segment build mode. Note that the baseline would simply continue (correctly in this case) down the existing trace. The model with NHMS would, however, build a new trace with the target address of the predicted nonhead miss at its head. Since NHMS predicted a nonhead miss when there was actually a nonhead hit, the original trace with this address in the body would still be in the trace cache, as well as the newly created trace with this address at its head. Furthermore, since we need to replace at least one trace cache line, we have now shortened some trace(s) within the trace cache to create space for this newly created trace. This complex process makes it difficult to pinpoint the reason for the increase in the number of BTB misses followed by trace cache hits (case 7); however, that increase is small.

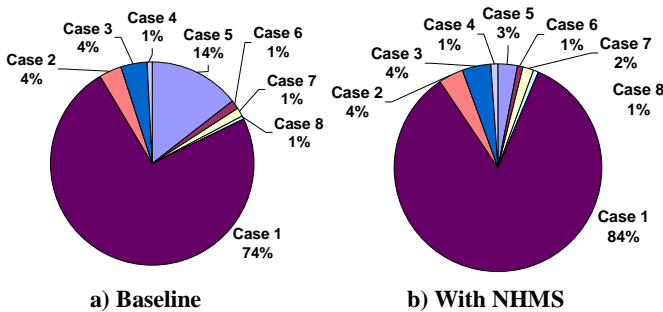


Figure 10. Trace cache access classification breakdowns averaged over all of the eight applications for the baseline with and without NHMS

Our NHMS implementation uses a reasonably modest amount of memory; its 64K entry table of 3-bit counters comprises just 24 KBytes of extra memory. The 4K entries of the NMTB each store a 32-bit nonhead address and a 32-bit target address; thus the entire NMTB is just 32 KBytes. An additional merit of this design is that these structures are off the critical path. In fact, an access to the nonhead miss predictor could be multiple cycles. The quicker the access, the sooner the determination can be made that there is a nonhead miss, and the better the overall benefit will be (and we did assume 1 cycle access in our simulations), but the size of the pipeline in our model leaves room for longer predictor access times. In Section 8.0, we compare NHMS to other techniques that could be used to reduce the nonhead miss penalty.

8. Comparison of NHMS to other schemes

In order to see the effectiveness of nonhead miss speculation, we can compare it to other schemes that also reduce the nonhead miss penalty. Here we introduce a Basic Block cache which caches only decoded basic blocks of the instruction stream. By design, this configuration should have greatly reduced nonhead misses, and should provide a good comparison to NHMS. One might assume that Basic Block cache would not have any nonhead misses, but since the trace cache lines are of finite size, there are instances when a basic block exceeds one trace cache line. Recall from Table 1 that the average basic block size is 5.09 x86 instructions and the average number of μ ops per instruction is 1.48. Multiplying these two numbers yields 7.50 μ ops per basic block on average. Since the number of μ ops per trace cache line in our models is 6, clearly many basic blocks will have multiple trace

cache lines and may get their tail segments replaced, leading to nonhead misses.

Another way to reduce nonhead misses is to access the BTB on every branch even if currently executing within a trace. We call this design a Branch Compare trace cache, since the branches within a trace must be compared to the output of the BTB prediction. With this model whenever the BTB predicts a different address from the next address in the trace, the current trace execution is abandoned and a new trace is searched for by doing a head lookup using the predicted target address. The Branch Compare approach has several implications. First, as with the Basic Block cache, more pressure is put on the BTB, since it must cache a much larger working set of branches, rather than just caching trace heads. Secondly, since a trace cache line can contain up to two branches in our baseline model, the BTB must be able to provide the results of two lookups in a single cycle. The most important distinction between Branch Compare and our baseline is the additional penalty incurred when the BTB makes an incorrect prediction, i.e. the BTB predicts a nonhead miss when in fact the current trace contains the correctly executed path.

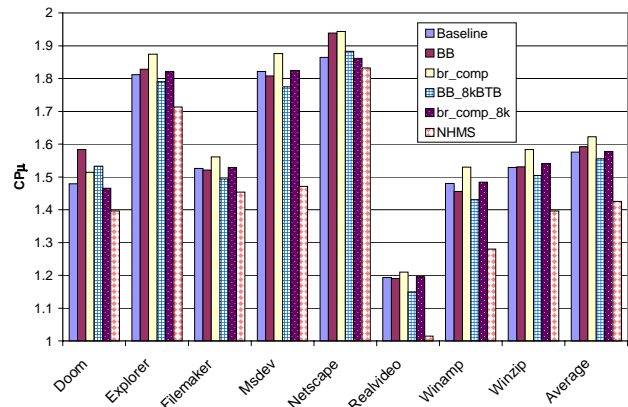


Figure 11. Comparison of Nonhead Miss Speculation (NHMS) to a Basic Block cache (BB) and a Branch Compare trace cache (br_comp)

Figure 11, shows how well NHMS performs relative to the Basic Block cache model and the Branch Compare model. The first bar represents our original model (*Baseline*), second is the Basic Block cache model (*BB*), and the third is the Branch Compare model (*br_comp*). Baseline and Basic Block cache perform similarly overall, with our original Baseline model performing about 1% better on average. Branch Compare, however, is consistently worse than the other two models (except on Doom), and performs about 3% worse than our original model on average.

Since both the BB and Branch Compare caches target nonhead misses, we wanted to directly compare these models to our NHMS cache model. To make a better comparison, the additional memory required for NHMS was added to both Basic Block and Branch Compare. Recall that NHMS requires an extra 56 KBytes of memory to implement its nonhead miss predictor and NMTB. By increasing the size of the Baseline BTB to 8K entries, we utilize an extra 60 KBytes of memory over a 512-entry BTB. The cycle time of such a BTB would most likely be larger than a single cycle, but to provide them the maximum benefit we assumed a single cycle access for both the BB and Branch

Compare caches. The improved results for the Basic Block cache with an 8K entry, 4-way BTB are shown by the fourth bars, labeled *BB_8kBTB*. Similarly, the fifth bars, labeled *br_comp_8k*, show the improved results of adding an 8K entry, 4-way BTB to Branch Compare.

Adding the larger BTB to Basic Block cache, drops the average CPμ from 1.59 to 1.56, roughly a 3% improvement. *BB_8k_btb* is just slightly better than the 1.58 average CPμ for Baseline. Branch Compare does not fare much better. Adding the larger BTB drops its CPμ from 1.62 to 1.58, also roughly a 3% improvement. On average, however, *br_comp_8k* still performs about the same as the Baseline model.

NHMS provides the greatest overall improvement for one main reason. It treats branches that are to be trace heads differently from branches within a trace. In our original model, many of the traces delivered by the trace cache are generally considered to be useful, i.e. they provide the back end with μops that are on the correct execution path. For this reason, it is a good policy to err on the side of conservatism when it comes to predicting that execution will diverge from the current trace once that trace has been started. By requiring all branches (as opposed to just trace heads) to access the BTB, many more conflicts are created, and the prediction accuracy of both trace heads and branches within a trace drops significantly. Our Baseline model avoids this problem by caching (in the BTB associated with its trace cache) only trace heads, thereby reducing the pressure on the BTB. Our Baseline model does have a problem with nonhead misses, but since most of the traces delivered by the trace cache are useful, it still performs better overall than either the BB or Branch Compare caches. NHMS achieves its performance gains by treating nonhead misses differently than ordinary branches or trace heads, since their behavior and the consequences of mispredicting them differs. By leveraging the fact that nonhead misses frequently occur at only one location within a trace, NHMS is able to reduce the average CPμ of our original Baseline model by 10%.

9. Conclusion

The effect of a linked list style trace cache on x86 processor performance is most sensitive to the size of the trace cache and the number of nonhead misses that occur. The Nonhead Miss Speculation (NHMS) mechanism introduced in this paper is comprised of a nonhead miss predictor and a nonhead miss target buffer and reduces the impact of nonhead misses by speculating when a nonhead miss will occur. Since the penalty for incorrectly speculating that a nonhead miss will occur is potentially worse than an actual nonhead miss, our speculation mechanism must be conservative. By removing all of the nonhead misses from the applications, we could achieve an average speedup of almost 12% over the Baseline model. NHMS reduces baseline CPμ by about 10% on average over all the applications studied. For certain applications, our NHMS implementation performs significantly better: a 20% improvement in CPμ over the baseline for MsDev, 15% for RealVideo, and 14% for Winamp.

10. References

- [1] S. Dutta and M. Franklin, "Control Flow Prediction with Tree-like Subgraphs for Superscalar Processors," *Proceedings of the 28th Annual International Symposium on Microarchitecture*, December 1995, pp. 258-263.
- [2] D.H Friendly, S.J. Patel, and Y.N. Patt, "Alternative Fetch and Issue Policies for the Trace Cache Fetch Mechanism," *Proceedings of the 30th Annual International Symposium on Microarchitecture*, pp. 24-33, December 1997.
- [3] R. Krick, G. Hinton, M. Upton, D. Sager, C. Lee, "Trace Based Instruction Caching," U. S. Patent 6,018,786, October 1997.
- [4] S.W. Melvin, M.C. Shebanow, and Y.N. Patt, "Hardware Support for Large Atomic Units in Dynamically Scheduled Machines," *Proceedings of the 21st Annual International Symposium on Microarchitecture*, pp. 60-66, December 1988.
- [5] S.J. Patel, "Trace Cache Design for Wide Issue Superscalar Processors," PhD Dissertation, University of Michigan, 1999.
- [6] A. Peleg and U. Weiser, "Dynamic Flow Instruction Cache Memory Organized Around Trace Segments Independent of Virtual Address Line," U.S. Patent Number 5,381,533, 1994.
- [7] E. Rotenberg, S. Bennett, and J.E. Smith, "Trace Cache: A Low Latency Approach to High Bandwidth Instruction Fetching," *Proceedings of the 29th Annual International Symposium on Microarchitecture*, pp. 24-34, December 1996.
- [8] Standard Performance Evaluation Corporation, *CPU2000 documentation*. <http://www.spec.org/osg/cpu2000/docs>
- [9] S. Vlaovic, "TAXI: Trace Analysis for X86 Interpretation," PhD Dissertation, University of Michigan, 2002.
- [10] T.-Y. Yeh, D.T. Marr, and Y.N. Patt, "Increasing the Instruction Fetch Rate via Multiple Branch Prediction and a Branch Address Cache," *Proceedings of the 7th International Conference on Supercomputing*, pp. 67-76, July 1993.