Critical Issues Regarding the Trace Cache Fetch Mechanism

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

Technical Report

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

ABSTRACT

In order to meet the demands of wider issue processors, fetch mechanisms will need to fetch multiple basic blocks per cycle. The trace cache supplies several basic blocks each cycle by storing logically contiguous instructions in physically contiguous storage. When a particular basic block is requested, the trace cache can potentially respond with the requested block along with several blocks that followed it when the block was last encountered.

In this technical report, we examine some critical features of a trace cache mechanism designed for a 16-wide issue processor and evaluate their effects on performance. We examine features such as cache associativity, storage partitioning, branch predictor design, instruction cache design, and fill unit design. We compare the performance of our trace cache mechanism with that of the design presented by Rotenberg et al [19] and show a 23% improvement in performance. In our final analysis, we compare our trace cache mechanism with an aggressive single basic block fetch mechanism and show that the trace cache mechanism attains a 24% improvement in performance.

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

1 Introduction

A wide issue processor can be divided into two architectural entities: a fetch engine that supplies instruction bandwidth and an execution engine that consumes it. In a dynamically scheduled processor, the fetch engine issues instructions into an instruction window from which ready instructions are executed by the execution engine. The division is similar in a statically scheduled machine with the exception that the hardware instruction window is not required.

The width of the fetch engine places an upper bound on the fetch bandwidth and thus the execution bandwidth of the processor, an observation commonly known as the Flynn Bottleneck. Maximizing the effective fetch bandwidth is not a straightforward task. Control mispredictions, cache misses, and partial fetches all reduce the effective fetch bandwidth.

When a control instruction is mispredicted, the fetch engine spends cycles fetching instructions which will later be discarded. Many sophisticated techniques have been developed to improve the prediction rates for conditional branches and new techniques such as predicated execution and multi-path execution have been explored to hedge against a complete loss of fetch cycles when a branch is mispredicted. Instruction cache misses cause the fetch engine to stall until the missing request is serviced. Techniques such as instruction prefetching, two-level caching, victim buffers, and set prediction all help to reduce the loss in fetch bandwidth due to cache misses.

Partial fetches, however, prevent the fetch engine from producing a full issue even when an access on the correct execution path hits in the instruction cache. Due to the physical layout of instruction caches, branches, jumps, subroutine calls and returns cause the remainder of fetched cache lines to be discarded. With an instruction cache, it is difficult to fetch both a branch and it's target in the same cycle if the branch is taken. Many previous studies have determined that the average basic block size for integer programs is around five instructions, indicating that fetch engines that cannot fetch beyond a control instruction in a single cycle can provide at around five instructions per cycle. Fetching beyond conditional branches that are not taken provides a boost of approximately two instruction per cycle [19]. While branch mispredictions and cache misses result in many cycles during which no useful instructions are fetched, they occur relatively infrequently. The loss due to partial fetches will be incurred almost every cycle instructions are fetched from the cache. The partial fetch problem is more significant for machines capable of issuing more than 6 instructions each cycle; they require a fetch engine that fetches multiple basic blocks per cycle. The focus of this technical report is the evaluation of a technique that improves the overall fetch bandwidth by fetching multiple basic blocks each cycle — the trace cache.

The trace cache is a new paradigm for caching instructions and directly deals with lost bandwidth due to partial fetches. The trace cache stores logically contiguous instruction sequences in physically contiguous storage, a concept first proposed by Melvin [15]. Like the instruction cache, the trace cache is accessed using the starting address of the next block of instructions. (The fetch engine divides very large basic blocks into fetch-width sized chunks, which we refer to as blocks and loosely treat as equivalent to basic blocks.) Unlike the instruction cache, a line of the trace cache contains blocks as they appear in execution order, as opposed to the static order determined by the compiler. Two adjacent instructions in a trace cache line need not be at adjacent addresses in the executable image. A trace cache line stores a *segment* of the dynamic instruction stream or trace, up to **n** instructions containing up to **m** conditional branches.

Trace cache lines are constructed by the fill unit (see figure 1). As a sequence of instructions is issued to the instruction window, it is latched into the fill unit. The fill unit will attempt to maximize the size of the segment by combining newly arriving instructions with instructions latched from previous cycles. The fill unit *finalizes* a segment when it can be expanded no further, for example when the **mth** branch is latched. Finalized segments are enqueued, and subsequently written into the trace cache.

The trace cache works in concert with a multiple branch predictor. The predictor must be able to predict as many branches each cycle as the trace cache is capable of supplying. Furthermore, the accuracy of the multiple branch predictor must not be significantly lower than a corresponding single branch predictor; otherwise gains in fetch bandwidth made by increasing the fetch size will be offset by more discarded fetches.

The primary objective of the trace cache is to increase fetch bandwidth by increasing the effective fetch rate — the average number of correct instructions fetched on fetch cycles which hit in the cache. This metric gives a direct indication of the losses due to partial fetches. By overcoming the basic block limitation, the trace cache can supply more instructions per cycle than a conventional fetch engine. A secondary objective is to make the reprocessing of instructions easier. Since instructions are written into the trace cache after being decoded, information can be stored along with the instructions minimizing the amount of work that needs to be done when the instructions are executed again.

In this technical report, we examine some of the key characteristics of the trace cache fetch mechanism such as size, associativity, fill strategies, and interactions with a multiple branch predictor. We present performance data on a wide set of fetch mechanism organizations and attempt to isolate those parameters that strongly affect performance. We also extend and test some of the ideas mentioned by Rotenberg et al [19] in their study of trace caches.

2 Related Work

The loss in fetch bandwidth due to partial fetches affects all superscalar processors. Since control instructions occur every fifth instruction, for processors that have a maximum execution bandwidth of five or fewer instructions per cycle, the problem is primarily one of encountering a cache line boundary before the full number of instructions are retrieved. While troublesome, this problem is overcome with straightforward techniques such as fetching two adjacent cache lines and realigning the instructions [11].

For processors capable of executing six or more instructions per cycle, the need to fetch beyond control instructions arises. As the focus of our current research in uniprocessor design has concentrated on wide issue machines (eg. 16 wide), the design of high-bandwidth fetch engines is extremely important.

Cache organizations for simultaneously fetching multiple basic blocks have been studied by Yeh et al [27], Conte et al [4] and Seznec et al [20]. By multiporting the instruction cache and/or the branch target buffer (BTB) and generating multiple fetch addresses and branch predictions per cycle, these schemes are theoretically able to overcome the single basic block fetch bottleneck.

A compile-time approach to solving the fetch bottleneck problem is the Block-Structured ISA [14, 9]. The static form of the program is organized into enlarged atomic units composed of multiple basic blocks by the compiler, which, in that case, plays a central role in attaining higher fetch bandwidth. Similar fetch rate increasing techniques are employed for statically scheduled machines through compiler techniques such as superblock scheduling [10] and trace scheduling [6, 3].

A precursor to the trace cache was first introduced by Melvin, Shebanow and Patt [16]. They proposed the fill unit to compact a basic block's worth of instructions into an entry in a decoded instruction cache. A hit in the decoded instruction cache results in a larger atomic unit of work than would be possible if the individual instructions were fetched and decoded one per cycle. In [15] the performance implications of the fill unit are discussed and the idea of dynamically combining basic blocks into larger "execution atomic units" (EAUs) to further increase the fetch bandwidth is first proposed. Two other extensions of the original schemes were presented by Smotherman and Franklin. In [7], they applied the original fill unit ideas to dynamically create VLIW instructions out of RISC-type operations. They reworked the fill unit finalization strategy by restricting the type of instruction dependencies allowed in a fill unit line and by filling both paths beyond a conditional branch. In [22], they demonstrated how a fill unit could help overcome the decoder bottleneck of a Pentium Pro type processor.

In addition to work on instruction caching techniques, there has also been serious investigation into predicting multiple branches per cycle. Fetching multiple blocks and predicting multiple branches go hand-in-hand. Dutta and Franklin [5] proposed subgraph oriented branch prediction mechanisms which use local subgraph history (akin to per-address history) to form a prediction. In addition to proposing a caching structure, Seznec et al [20] also presented a multiple branch predictor capable of predicting two branches per cycle. Recently, Wallace and Bagherzadeh extended Seznec's scheme [25].

The trace cache concept was investigated by Rotenberg et al [19]. They presented a thorough comparison between the trace cache scheme to the current hardware-based high-bandwidth fetch schemes, clearly showing the advantage of using a trace cache, both in performance and latency. They also present a multiple branch predictor capable of making three predictions per cycle. They, however, relegated the trace cache to a supporting role in the fetch engine, showing that in a machine a with a 128KB instruction cache and a 4KB trace cache, performance is boosted by 28% on the Instruction Benchmark Suite (IBS) and the SPECint92 benchmarks over a single block fetch mechanism with a GAg single branch predictor.

3 The Trace Cache Fetch Mechanism

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

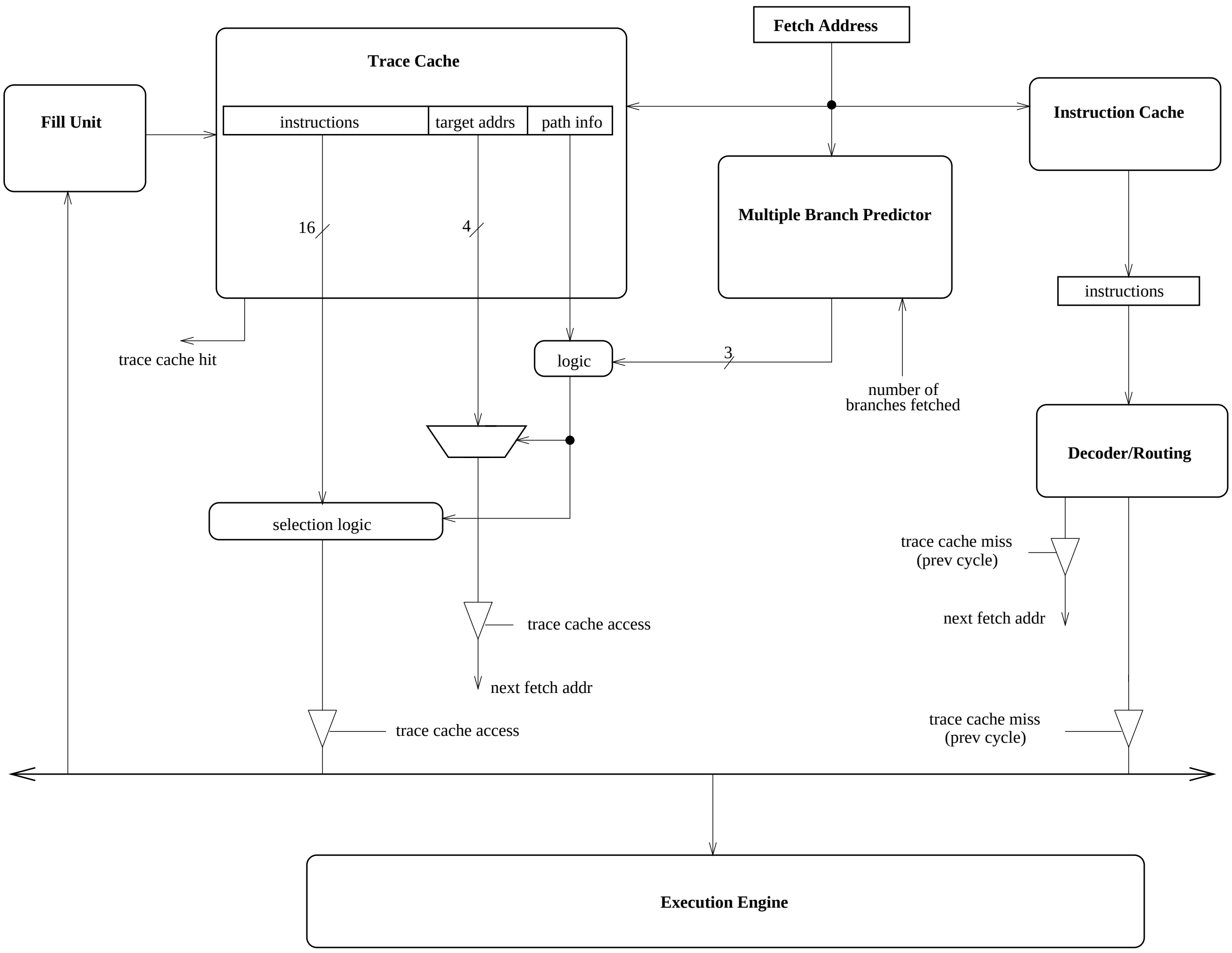

Figure 1: The trace cache fetch mechanism.

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

3.1 The Trace Cache

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

Figure 2 shows an example of a trace cache fetch cycle. The address of block A is presented to the trace cache. The trace cache responds with a hit and drives out the selected segment composed of the blocks A, B, and C. The prediction structures are accessed concurrently with the trace cache. At the end of the cycle, the segment is matched with the prediction. Since the predictor selects B to follow A but D to follow B, only A and B are supplied for execution. This is called partial matching and was briefly discussed by Rotenberg [19]. Its performance implications will be examined in section 5.4.

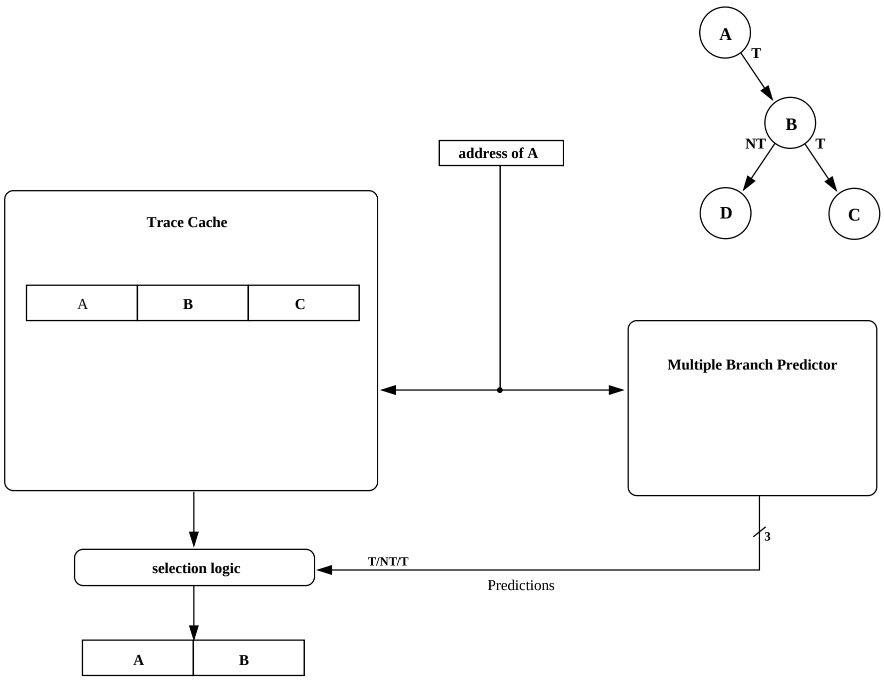

Figure 2: The trace cache and branch predictor are indexed with the address of block A. The inset figure shows the control flow from block A. The predictor selects path ABD. The trace cache only contains ABC. AB is supplied.

The trace cache can store segments of three basic blocks per line. The line is accessed by the address of the first instruction in the first block of the segment. The organization of the trace cache is similar to that of a conventional instruction cache, as the lines may be arranged in an associative manner. A hit is determined by a tag match.

Each line of the trace cache contains:

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

- Four target addresses. With three basic blocks per segment and the ability to fetch partial segments, there are four possible targets to a segment. The four addresses are explicitly stored allowing immediate generation of the next fetch address, even for cases where only a partial segment matches.
- Path information. This field encodes the number and directions of branches in the segment and includes bits to identify whether a segment ends in a branch and whether that branch is a return from subroutine instruction. In the case of a return instruction, the return address stack [12] provides the next fetch address. For reasons mentioned below, this information is stored in the tag store.

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

Instruction dependencies within a segment are pre-analyzed before the segment is stored in the trace cache. The source operand identifiers of each instruction each include a two-bit tag indicating whether the source value is produced by an instruction within the segment or is produced by an instruction issued in a previous cycle. If the value is produced internally, then the tag indicates which block within the segment the producing instruction belongs to. With this information, the rename logic can quickly determine whether the renamed tag for the source operand is supplied by the register alias table (RAT) or can be constructed without a RAT access. The destination operand identifier for each instruction is augmented with a one-bit tag indicating whether its value lives beyond its basic block, i.e. is live-out of its basic block. All values that are live-out are renamed and given a physical register. This allows the checkpoint repair mechanism, which has to create up to three checkpoints per cycle, to recover from a branch misprediction that occurs in the middle of a segment without having to discard the entire segment.

With this additional information, each segment that is retrieved from the trace cache requires minimal renaming before being merged into the instruction window. Only instructions that have a source operand produced by an instruction outside the segment require a lookup in the RAT and only instructions that produce a value that is live-out require a physical register. A complex dependency analysis of 16 instructions does not need to be performed on the fetched segment. Sprangle and Patt [23] demonstrated that pre-analyzed fetch packets require fewer read and write ports to the register renaming structures and the register file.

Finally, instructions can be stored in an order that permits quick issue. Because the segment is pre-analyzed and the dependencies are explicitly marked, the ordering of instructions within the trace cache line carries no significance. Instructions within the cache line can be ordered to mitigate the routing required to send instructions into functional unit reservation stations. Future microarchitectures may arrange the instructions within a segment to reduce the communications delays associated with incomplete bypass networks.

A segment is written into the trace cache only if the trace cache does not contain a larger, subsuming segment. For instance, if segment ABC were resident in the cache, the new segment AB would not be added. However, if ABC were resident in the cache, the segment ABD would overwrite it. To facilitate this, the path information is included in the tag entry of each line. If the tag indicates that a line exists with the same starting block and the path information indicates that the line contains a larger segment along the same path, then the incoming write is not committed to the trace cache. To implement this with nominal performance impact, a second read port is required in the trace cache tag store. The data store requires no additional read ports.

An important design option of the trace cache is that only one segment starting at a particular basic block can be resident in the cache at any given time. The implications of relaxing this constraint are discussed in section 5.3.

3.2 The Fill Unit

The job of the fill unit is to collect instructions as they are issued by the processor and combine them into segments to be written into the trace cache. Conceptually, the instructions are presented to the fill unit as blocks, in the order they were fetched. The fill unit merges the arriving blocks with awaiting blocks latched in a previous cycle. The merge process involves maintaining dependency information and reordering instructions. The process continues until the segment becomes *finalized*, at which point it is enqueued to be written into the trace cache.

Dependency information is maintained by simply recalculating the two-bit source operand dependency tag on each arriving instruction. This tag is changed to reflect whether an awaiting block provides the value.

A segment becomes finalized when

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

Rule 1 is implied by the size of the trace cache line and rule 2 by the number of predictions supplied by the predictor. Increasing these two limits could lead to higher fetch bandwidths. Because their targets vary, return instructions and indirect jumps cause finalization (rule 3). Unconditional branches, however, are replaced by NOPs. Furthermore, subroutine calls do not cause finalization.

Because blocks are being combined in a greedy fashion, there are cases where the fill unit can create multiple copies of basic blocks in the trace cache. Figure 3 shows a simple loop composed of five basic blocks. The fill unit can potentially create five combinations, all of which can be simultaneously resident in the trace cache. This block redundancy may degrade performance of the trace cache mechanism by displacing useful lines with redundant information. The tradeoff here is between higher bandwidth from fetching fuller segments versus bandwidth losses due to increased misses in the trace cache. Similar situations are possible with all blocks where several execution paths merge, such as block B in figure 3.

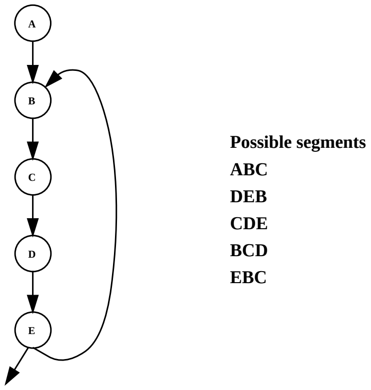

Figure 3: If the fill unit is able to create three-block segments for this path through a loop, then all five possible segments will be created and stored in the trace cache.

The fill unit strategy treats basic blocks as atomic entities. A basic block is not split across two segments unless the block is larger than 16 instructions. In addition to exacerbating the block redundancy problem, splitting a basic block creates an additional block that may tax other structures such as the branch predictor.

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

The fill unit can collect blocks as they are issued into the instruction window or as they are retired. If blocks are collected at retire time, segments due to speculative execution are not added to the cache, potentially reducing misses. On the other hand, creating segments on the wrong path may generate segments that may be useful later. The additional lag time in generating segments after retirement might also have negative effect on performance: the first few iterations of a tight loop will miss until the first iteration is retired and written into the trace cache. The effects of this design issue are examined in section 5.5.

3.3 The Branch Predictor

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

Two level adaptive branch prediction has been demonstrated to achieve high prediction accuracy over a wide set of applications [28]. In a two level scheme, the first level of history records the outcomes of the most recently executed branches. The second level of history records the most likely outcome when a particular pattern in the first level history is encountered.

The first level history can be maintained *per address* by storing the most recent outcomes separately for each branch or *globally* by storing the most recent outcomes together for all branches. This history is maintained in a single *history register* for the global scheme or in multiple history registers for the per address scheme. The second level history is maintained as a table, called the *pattern history table* (PHT).

To make three predictions per cycle, we expand each PHT entry from a single two-bit counter into seven two-bit counters. Figure 4 shows how the seven counters are used to supply three predictions per cycle. The first two-bit counter supplies the prediction for the first branch and is used to select which of two two-bit counters supplies the prediction for the second branch. Both predictions are used to select one of four two-bit counters to supply the prediction for the third branch. All three predictions are made with a single access to the pattern history table. To circumvent critical path problems in accessing the PHT, the selection of two-bit counters can be done when the branches corresponding to a prediction have resolved and the next set of predictions can be stored directly within the entry. This adds two bits to the cost of each entry.

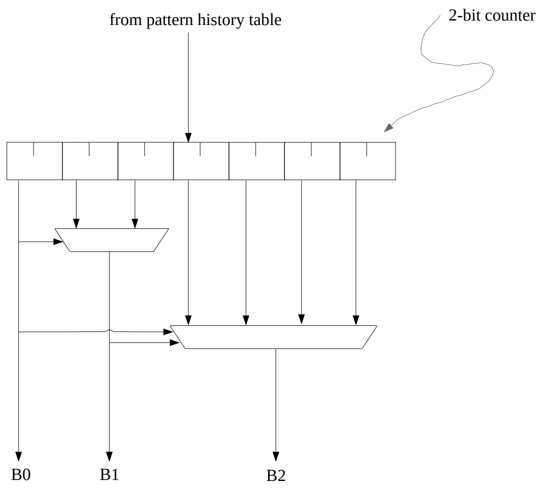

Figure 4: Pattern history table entry. Seven two-bit counters are used to provide three predictions. B0 is the prediction for the first branch, B1 is the prediction of the second and B2 is for the third.

Although other methods for indexing the PHT are examined in section 5.8, the baseline configuration for the trace cache predictor uses the gshare scheme outlined by McFarling [13]. The global branch history is XORed with the current fetch address, forming an index into the PHT. Gshare has been shown to be effective in reducing negative interference in commonly accessed PHT entries, such as entries accessed by strongly biased branches. McFarling has demonstrated the increased accuracy of gshare over other global history based prediction schemes.

The extra selection done after the PHT is indexed adds two extra bits of branch history. Thus such a predictor indexed with 15 bits of history actually utilizes 17 bits of history. However, this extra history is utilized only if the trace cache line being fetched contains three conditional branches. In the worst pathological case, if the trace cache only contains segments composed of single blocks, only the first two bit counter in each PHT entry is utilized. In the best case, if each trace cache line that is accessed contains three conditional branches, the PHT entries will be fully utilized.

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

3.4 The Instruction Cache

A conventional instruction cache supports the trace cache by supplying instructions when the trace cache does not contain the requested segment. If hitting in the trace cache is the frequently occurring case, then the supporting fetch mechanism need not be enhanced for higher bandwidth. The instruction cache can get away with supplying up to one basic block each cycle.

Both the trace cache and instruction cache are accessed concurrently with the same address. If the access misses in the trace cache, instructions are supplied by the instruction cache after a delay of one cycle. This delay allows for a shorter latency of trace cache access. Essentially, we optimize for the case that the fetch will hit in the trace cache. In the event of a miss in the trace cache and a hit in the instruction cache, an extra cycle of latency is added to the fetch.

The latency through the instruction cache is likely to be less than or similar to the latency of the trace cache. The delay cycle may be used to decode the instructions fetched from the instruction cache and calculate the next fetch address, as shown in figure 1. For this reason, no branch target buffer is required for the trace cache fetch mechanism.

The instruction cache has two read ports to allow adjacent cache lines to be retrieved each cycle. By fetching two cache lines and realigning instructions, the fetch mechanism overcomes fetch breaks due to cache line boundaries. The need for this optimization will be examined in section 5.9.

3.5 The Fetch Cycle

At the beginning of the cycle the fetch address, determined in the previous cycle, is presented to both the trace cache and the instruction cache. The fetch address is also used by the branch predictor, along with global branch history, to index into the pattern history table. Some time into the cycle, the trace cache will respond with a hit and a segment of instructions or a miss. The branch predictor will respond with three predictions. The instruction cache will produce two cache lines, along with a hit/miss signal.

In the case of a trace cache hit, the branch predictions are used to select which part of the segment to supply to the machine. Logic at the output of the trace cache attempts to match the path information stored in the selected trace cache line with the current prediction. The predictions also select which of the four possible fetch addresses to use for the following cycle. If the fetched segment ends in a return, the next fetch address is supplied by the return address stack. The branch predictor's history register is updated, shifting in the appropriate predictions for the fetched branches.

Selecting the proper next fetch address and updating the global history register are the critical operations to complete in the fetch cycle. If a partial segment matches, unwanted instructions can be invalidated in the next stage of processing.

In the case of a trace cache miss and an instruction cache hit, up to a single basic block is supplied at the end of the next cycle. Because the fetch path is optimized for a trace cache hit, detecting and adjusting for a trace cache miss requires an extra cycle. Since the instruction cache access is complete at the end of the first cycle, the second cycle is spent decoding the fetched instructions.

If both the trace cache and instruction cache respond with a miss, then a request for the missing instruction cache line is made to the second level cache. The fetch mechanism stalls until the missing line arrives.

4 Experimental Setup

The trace cache was added as the fetch mechanism of a 16-wide issue processor implementing the HPS model of execution [17]. To isolate the effects of the fetch engine, many of the bottlenecks in the execution core have been removed. The data cache is ideal (all load requests require one cycle of address calculation and one cycle to access the data) and the 16 functional units are uniform and capable of all operations. The latencies for different operations are listed in Table 1. The instruction window is 16x32 instructions. The HPS model uses checkpointing to recover from branch mispredictions and exceptions. The execution engine is capable of creating three checkpoints each cycle. All instructions undergo four stages of processing: fetch, issue, schedule, execute. Instructions fetched from the instruction cache undergo at least one extra stage of decoding. Figure 5 shows a block diagram of our pipeline model. All stages take at least one cycle.

Operation	Latency
INT ops	1
LOAD	2
STORE	2
FMUL	3
FADD	3
FDIV	8

Table 1: Functional unit latencies.

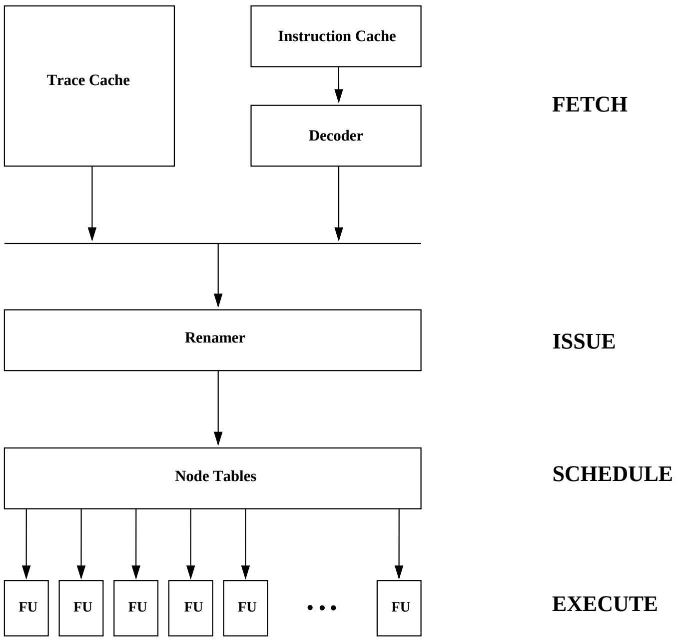

Figure 5: Trace Cache/HPS pipeline diagram.

The Trace Cache/HPS machine was simulated on an executable driven simulator which models wrong path effects. The simulator operates on executables of the HPS baseline ISA. The instruction set is an enhanced subset of the MC88100 ISA. All experiments were performed on the SPECint95 benchmarks. Table 2 lists the benchmarks and the input sets [1]. All simulations were run until completion.

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

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

The benchmarks go and gcc are relevant for fetch mechanism issues. Both contain a large number of dynamic basic blocks and have many conditional branches that are difficult to predict.

For all simulations performed, the return address stack was assumed to be ideal.

The latency for accessing the ideal second level cache is 10 cycles.

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

5 Critical Issues

To evaluate the various trace cache design options available, we started with the following baseline configuration: a 128KB trace cache which can be accessed in a single cycle, and a 4KB dual-ported instruction cache with a 64B line size. Fetches from the instruction cache undergo one delay cycle, during which the next fetch address is generated. Both caches are 4-way set associative. The fill unit collects blocks as they are issued into the execution core. The branch predictor keeps 15 bits of history and uses gshare to index the pattern history table. Using this configuration as the baseline model, we performed experiments to determine the impact of varying key fetch engine parameters. In addition, to demonstrate the effectiveness of the trace cache fetch mechanism, we compare its performance to the performance of a similarly sized single block fetch mechanism with an aggressive hybrid branch predictor.

5.1 Trace Cache associativity

The first set of experiments deals with determining the effects of set associativity on trace cache performance. Figure 6 shows the performance of our baseline trace cache fetch mechanism composed of a 128KB trace cache and a 4KB instruction cache as the trace cache associativity is varied from 1 to 32.

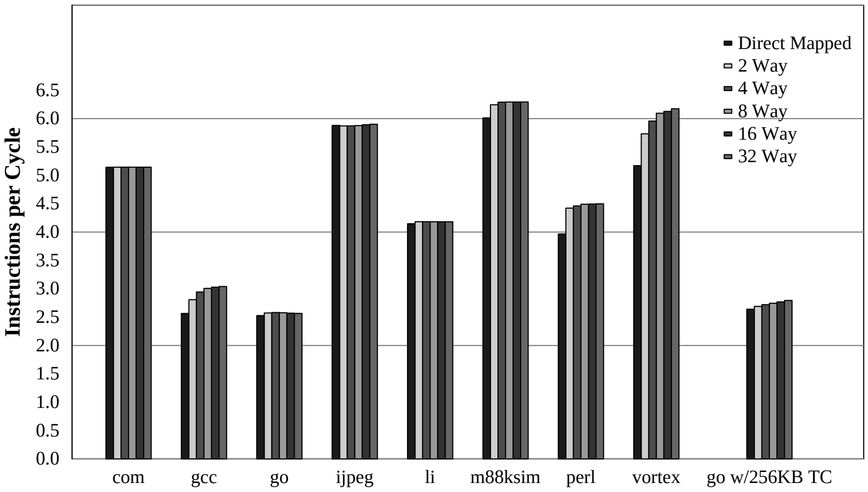

Figure 6: Associativity. The performance in instructions completed per cycle is shown graphically for trace caches of associativities ranging from 1 to 32. For all configurations, the trace cache size is 128KB. A 4KB instruction cache is the supporting structure. An additional plot shows the performance of go with a 256KB trace cache.

Many of the SPECint95 benchmarks, such as compress, li, and m88ksim have small instruction working sets, that fit entirely within a 128KB trace cache, thus varying associativity has very little impact. For benchmarks such as gcc, perl, vortex, going from a direct-mapped configuration to a set-associative one shows a dramatic improvement. The improvement in performance of a 4-way set-associative trace cache over a direct-mapped trace cache is 15% for gcc, 12% for perl, and 15% for vortex.

The benchmark go displays erratic behavior. Although it is not apparent from the diagram, its performance rises and falls slightly as the trace cache associativity is increased. This benchmark has a large instruction working set and suffers from a significant number of capacity misses with a 128KB trace cache. If the size of the cache is doubled and the same experiment is performed on go, then the performance trends are more in line with expectations, as shown by the relevant data plotted in figure 6.

Adding associativity to a trace cache improves performance significantly as seen on gcc, perl, and vortex. With a direct-mapped 128KB trace cache, blocks which are 2048 instructions apart will map to the same line. Since branch targets tend to exhibit spatial locality [24], direct-mapped trace caches suffer a significant penalty due to conflict misses.

Implementing associativity will increase the cache access latency. However, several recent techniques such as set prediction [29, 8] and remap caches [18] allow caches to exhibit both lower conflict misses and lower access times. Such techniques can be applied to the trace cache mechanism to improve performance without incurring an increase in cycle time.

5.2 Relative sizes

As stated in section 3.5, the trace cache and instruction cache are accessed in parallel with the next fetch address. At the end of the access cycle, the trace cache will respond with either a hit and a segment of instructions will be latched into the instruction latch or it will respond with a miss. If it responds with a miss and the instruction cache responds with a hit, then a block of instructions fetched from the instruction cache will be latched into the instruction latch at the end of the subsequent cycle. The extra cycle of delay allows for two things: first, the fetch path can be optimized for a trace cache hit and, at the end of the access cycle, only instructions from the trace cache are latched into the instruction latch, which helps reduce the fetch latency. Second, the next fetch address for the block fetched from the instruction cache can be generated during this extra cycle. No extra storage structure is required to maintain branch target information for blocks fetched from the instruction cache.

The purpose of this experiment is to determine the partitioning of storage between trace cache and instruction cache for the trace cache fetch mechanism. Since we established (as we will show in section 5.9) that the extra cycle of latency in accessing the instruction cache minimally affects performance of our baseline configuration and since we suspect that this extra cycle will more significantly impact other configurations with a relatively larger instruction cache, we do not model the additional latency for this experiment. Both structures are capable of supplying instructions at the end of an access cycle.

For these experiments, the instruction cache is dual ported and capable of fetching up to one block of instructions each cycle. Since it is accessed in a single cycle, a branch target buffer is required to generate the next fetch address for accesses serviced by the instruction cache. For these experiments, we modeled an ideal BTB.

Figure 7 shows the performance of several fetch mechanisms with various cache sizes. The results are not surprising: it is better to hit frequently in the cache that will supply more instructions per cycle. A fetch mechanism consisting of 128KB of trace cache provides a 5% performance increase on gcc, a 43% increase on m88ksim, a 15% increase on perl, and a 12% increase on vortex over a fetch mechanism consisting of a 4KB trace cache. If the trace cache and instruction cache are both the same size, the performance is slightly better overall than that of the 128KB trace cache mechanism. However, one should note that the instruction cache is dual read-ported in both tag and data stores and a larger branch target buffer would be required to enable single cycle access.

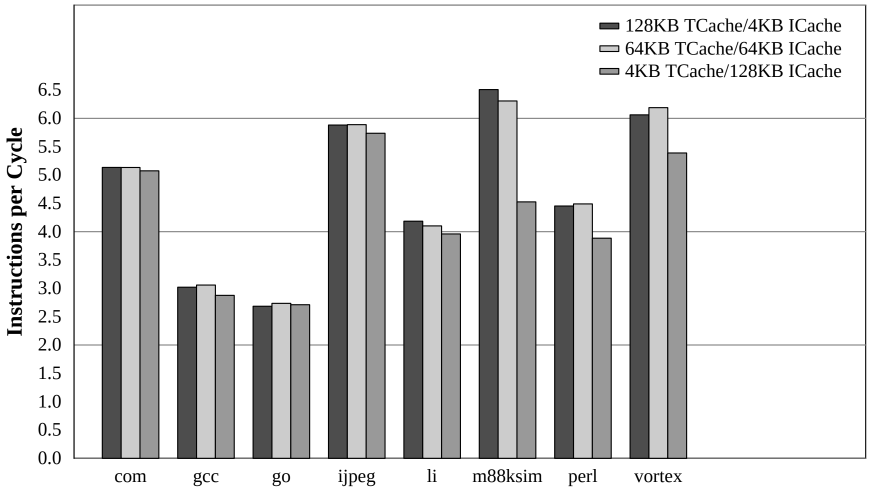

Figure 7: The performance of 128KB fetch mechanisms. The size of the trace cache plays a dominant role in the trace cache fetch mechanism.

Figures 8 and 9 show the results for 64KB and 256KB fetch mechanisms. The results are similar across all three sizes.

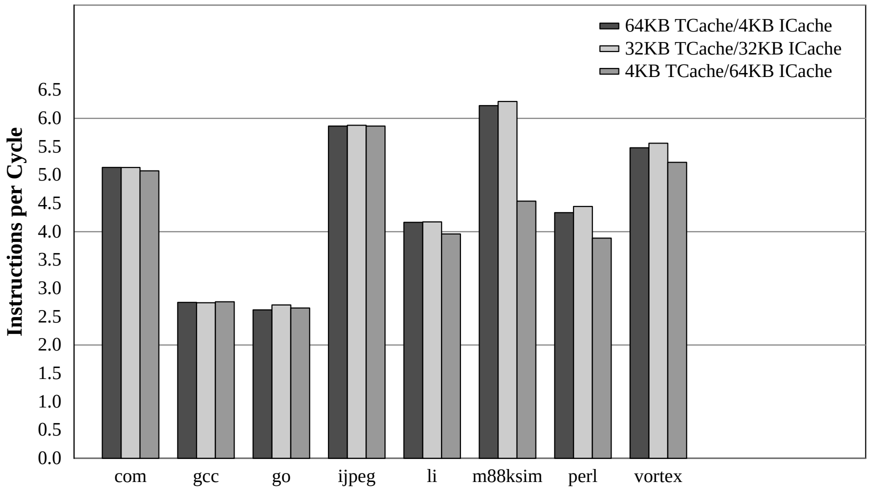

Figure 8: The performance of 64KB fetch mechanisms.

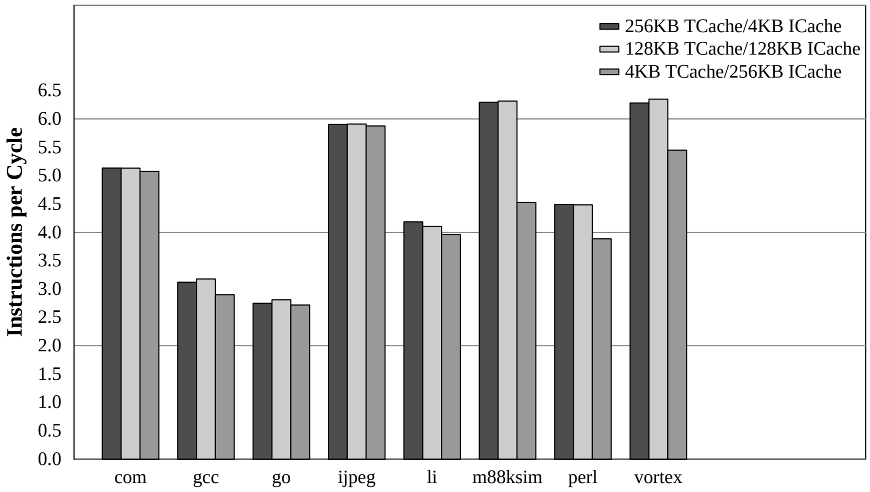

Figure 9: The performance of 256KB fetch mechanisms.

5.3 Path Associativity

Path associativity [19] relaxes the constraint that different segments starting from the same basic block cannot be stored in the trace cache at the same time. Path associativity allows segments ABC and ABD to reside concurrently in the cache whereas a non-path associative trace cache allows only one segment starting at A to be resident in the trace cache at any instance in time. A path associative trace cache datapath is shown in Figure 10.

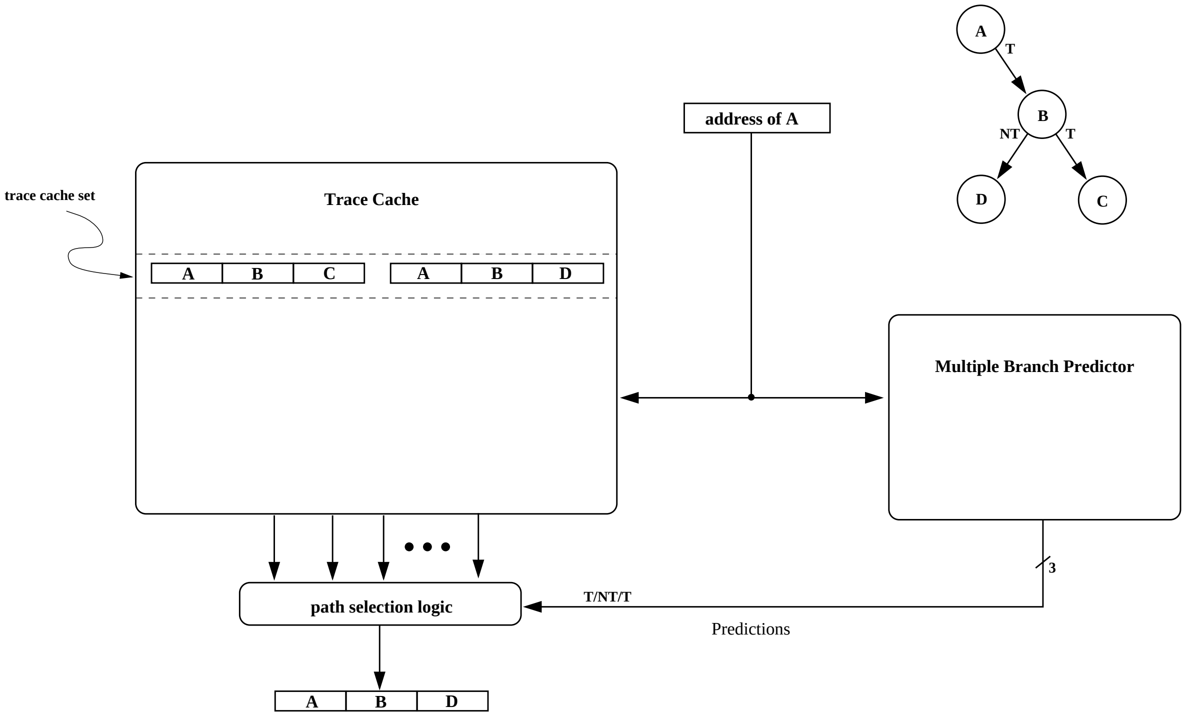

Figure 10: The trace cache drives out all segments in the set. The prediction is used to select the longest matching segment.

All segments starting at the same address are stored in the same set of the cache. In the same manner as the non-path-associative case, the fetch address is used to index into the trace cache and a tag match is performed to find the matching line. The tag matching processes of a path associative cache and a non-path associative cache have a slight but crucial difference. Both require a tag match of the upper bits of address and both require a path match with the branch prediction to select the proper next fetch address. However, the path associative cache requires that the longest matching path be determined before the matching line is selected. Finding the longest matching path requires completing the address match first. Only after the address match is complete, can the longest matching path and thus the next fetch address and instructions to supply, be selected. It is likely that the line selection time for the path associative cache will be longer, possibly impacting fetch latency.

The data plotted in Figure 11 indicate that path associativity enhances performance by 4% on go, 2% on compress, and 2% on m88ksim compared to the baseline configuration. Adding path associativity increases the number of segments that map into a particular set; thus additional misses may occur due to increased set conflicts. This is the reason gcc and vortex suffer a small loss of performance in the path associative case. Increasing the degree of associativity of the trace cache from 4-way to 8-way further increases the performance of gcc by 3%. Of course, a portion of this performance increase is due to the increase in associativity. The data indicates that the gains from path associativity are marginal and will be offset if the additional selection complexity results in a multi-cycle cache access or a longer cycle time.

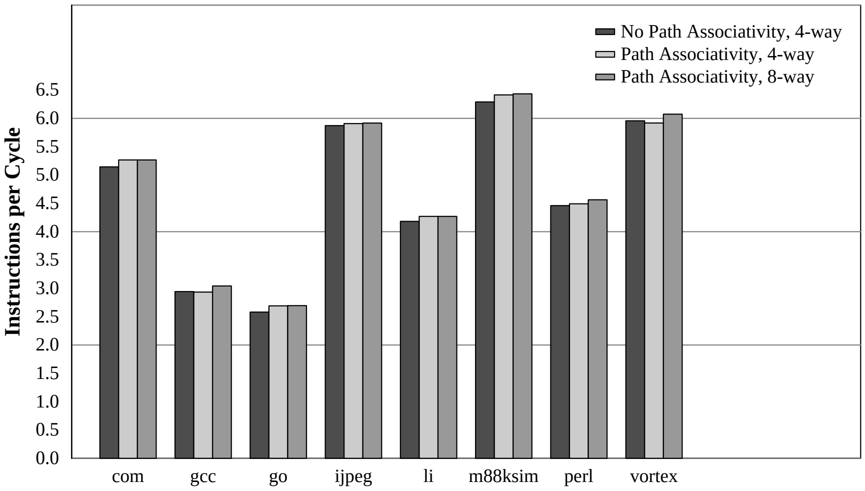

Figure 11: Performance of Path Associativity. Performance presented in instructions per cycle as measured on the SPECint95 benchmarks.

Path associativity tends to be overshadowed by the ability to partially match a segment in the trace cache. If the predictor speculates that the next path is ABD and a non-path associative trace cache contains ABC, only the segment AB will be fetched. In the path associative case, the cache will be able to supply the full three block segment ABC or ABD, depending on the prediction, if both segments are resident in the cache. Because partial matching allows for some instructions to be returned in the case where the segment does not match the full predicted path, path associativity could be an important factor on performance for trace caches that do not partially match segments.

5.4 Partial matching

Each fetch cycle, the predictions made by the branch predictor are used to select which blocks within the accessed segment will be issued to the core. This process is referred to as partial matching [19]. Alternatively, the trace cache can be designed to signal a hit only if all the blocks within the selected segment match. Figure 12 shows the performance difference between a trace cache that implements partial matching and one that does not. The average performance improvement is 25%.

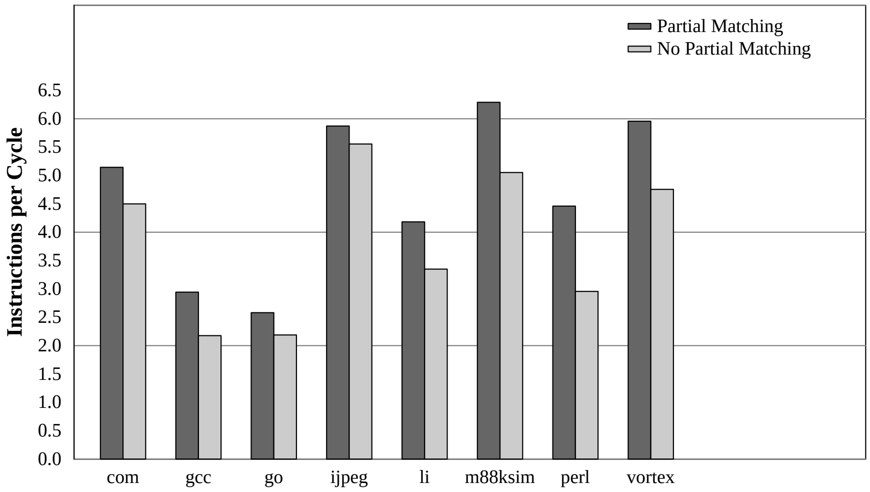

Figure 12: Partial Matching. The performance difference between a trace cache that partially matches segments and one that does not is around 25%

5.5 Block collection

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

Figure 13 shows that the differences in performance between the two schemes are slight. The fill unit collecting instructions at issue time provides increased traffic to the trace cache because segments are collected while executing on a wrong execution path. In some cases this prefetches useful segments, but in other cases it evicts useful segments from the trace cache.

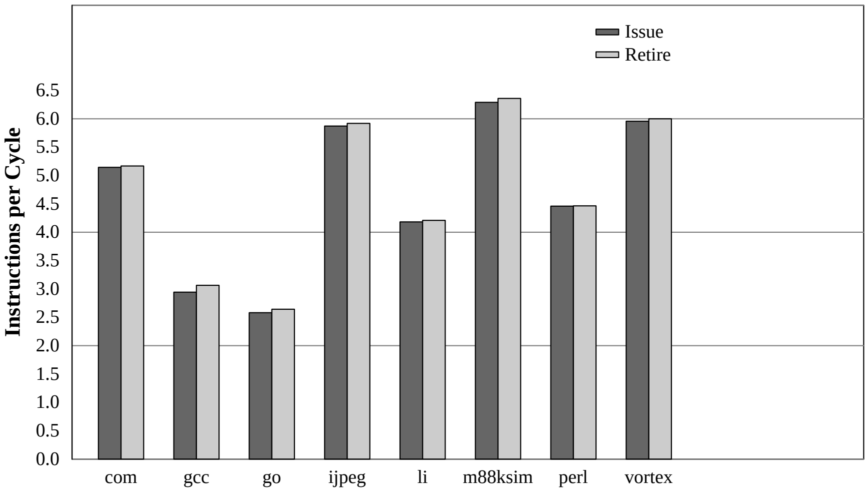

Figure 13: Issue vs. Retire. This plot shows that collecting instructions at issue time is not very different from collecting instructions at retire time.

A fill unit that collects at retirement time only writes segments from the correct execution path to the trace cache. However, it suffers from an increased latency between the initial fetch of a block and its collection into a segment and subsequent storage into the trace cache. While this can potentially impact the first few iterations of a tight loop, which will be fetched from the instruction cache until the first iteration retires, we will show in section 5.6 that this is not an important performance factor.

Although fill units that collect at retirement time have a slight advantage over those that collect at issue time, collecting segments at retirement time requires a greater hardware investment. Implementing an issue time fill unit is straight-forward, requiring blocks to be latched into the node tables and fill unit concurrently. For retirement time collection either the fill unit must contain enough storage to maintain a copy of the instructions in the execution window or blocks of instructions must be driven from the node table into the fill unit as they retire. In the former case checkpoints are added to the fill unit as they are issued but are only eligible for merging into larger segments after they retire. In the latter case wiring must be added from the node table to the fill unit so that the checkpoints may be entered as they retire.

5.6 Fill Unit Latency

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

The purpose of this experiment is to determine the sensitivity of the trace cache fetch mechanism to fill unit latency. Figure 14 shows the results of varying the number of cycles from the arrival of the terminal block to the point it is written into the trace cache. The results show that a fill unit with a 10-cycle latency has a negligible loss in performance over a single-cycle fill unit.

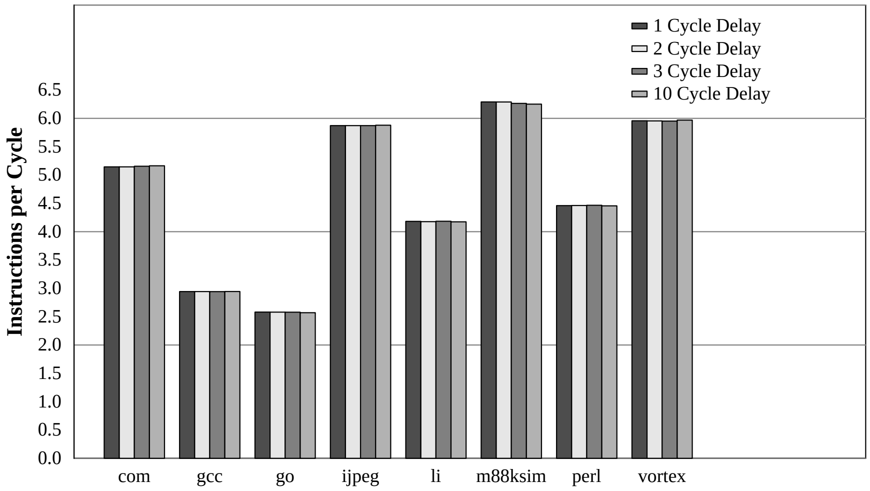

Figure 14: Fill Unit Latency is not a major influence on performance.

5.7 Non-atomic blocks

One obvious fill unit optimization is the possibility of filling out the trace cache lines by allowing the fill unit to finalize segments on instruction boundaries rather than basic block boundaries. Relaxing this restriction allows the fill unit to store the first portion of a basic block as the last set of instructions in a segment. The remaining instructions are then used as the first portion of the next segment created by the fill unit. Although this lets the fill unit create longer segments — the finalization rules are now 16 instructions, three conditional branches, or an indirect jump, return, or serializing instruction — it may also have negative effects on other factors which influence fetch engine performance.

Figure 15 compares the performance of a trace cache in which blocks are not treated atomically by the fill unit and the baseline case where blocks are are treated atomically. Although breaking blocks provides a slight performance advantage for compress, li, m88ksim, and perl, performance is worse on gcc, go, ijpeg, and vortex. The IPC of go drops more than 6.5%.

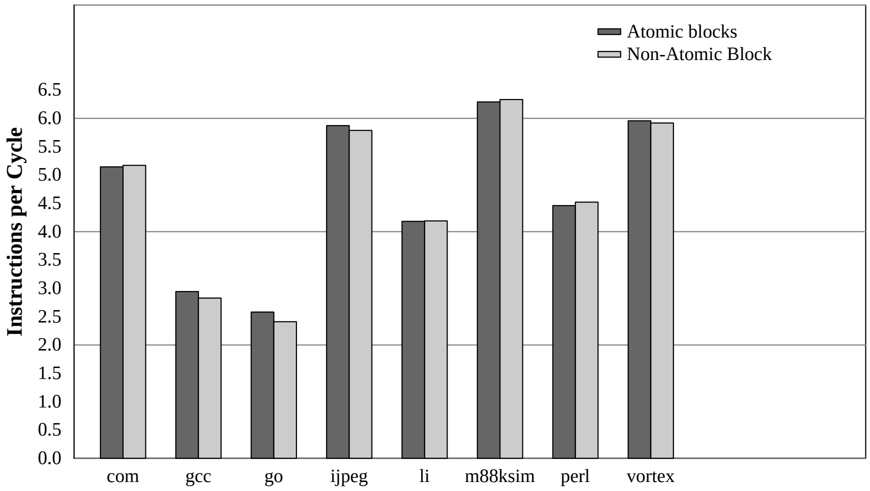

Figure 15: Perfomance impact of not treating basic blocks atomically in the fill unit. Allowing the fill unit to split a block across two segments results in a loss of performance.

To understand these results we first must look at how this change affects the effective fetch rate. Table 3 shows the average number of valid instructions provided by the fetch engine every cycle it supplies correct path instructions to the execution core. This metric provides a direct measure of how partial fetches are affecting performance. With the exceptions of go and li, allowing the fill unit to break blocks slightly increases the number of instructions fetched on average. This increase is not as significant as one would expect because partial matching allows the branch predictor to select a portion of a fetched line and because a misprediction on a branch in the middle of a segment causes the subsequent instructions in the segment to be later discarded (thus they do not contribute to the effective fetch rate). However, this slight increase does not indicate a corresponding increase in performance.

Benchmark	Atomic	Non-atomic
compress	10.565	10.605
gcc	8.395	8.465
go	8.656	8.471
ijpeg	11.839	12.236
li	9.337	9.315
m88ksim	10.259	10.501
perl	9.373	9.570
vortex	10.734	11.253

Table 3: The effective fetch rates for atomic (baseline) and non-atomic treatment of basic blocks by the fill unit.

If we look at the other factors which reduce fetch bandwidth, we see that by not treating blocks atomically, we aggravate them. Allowing the fill unit to break blocks creates more dynamic basic blocks (ie. unique addresses from which instructions are fetched during execution). An increased number of blocks exacerbates the redundancy problem described in section 3.2 and increases demand for space in the trace cache. Breaking a block also aggravates the branch predictor by adding another address from which the same branch must be predicted. Table 4 shows the trace cache miss rates for fetches along the correct path. The added pressure on the trace cache drives the miss rate up dramatically. Table 5 shows the impact that these additional fetch addresses have upon the branch predictor. Also, the additional bandwidth gets into the core only one cycle early and may not have a significant impact on the overall performance of a dynamically scheduled machine.

Benchmark	Atomic	Non-atomic
compress	0.014%	0.030%
gcc	7.608%	9.761%
go	13.187%	19.359%
ijpeg	0.408%	1.394%
li	0.087%	0.222%
m88ksim	0.872%	0.479%
perl	0.615%	1.185%
vortex	2.677%	3.711%

Table 4: The trace cache miss rates for atomic (baseline) and non-atomic treatment of basic blocks by the fill unit.

Benchmark	Atomic	Non-atomic
compress	8.002%	8.060%
gcc	9.015%	9.636%
go	18.218%	19.570%
ijpeg	7.479%	7.602%
li	4.620%	4.712%
m88ksim	1.500%	1.490%
perl	2.264%	2.240%
vortex	1.019%	1.104%

Table 5: The conditional branch misprediction rates for atomic (baseline) and non-atomic treatment of basic blocks by the fill unit.

Thus while allowing the fill unit to break blocks does increase the average number of instructions the trace cache stores per line, this gain is offset by losses due to more mispredictions and cache misses.

5.8 Branch Predictor issues

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

In this section we examine several techniques for indexing the pattern history table, whose entries are described in section 3.3. In our baseline configuration, we chose to use the gshare indexing scheme, where the global history register is hashed (XORed, in our case) with the next fetch address to form the PHT index. Here, we explore other indexing options.

In figure 16, the conditional branch misprediction rates for the go benchmark are shown for four different PHT index methods. The first method is to use only bits of the next fetch address. This technique is similar to the 2-bit counter technique [21] used for single-level branch predictors. The second method is to use only bits of the global history, or GAg. The third method is to use a combination of the next fetch address and the global history, or GAs [2]. Finally, we include the gshare indexing scheme used in our baseline configuration.

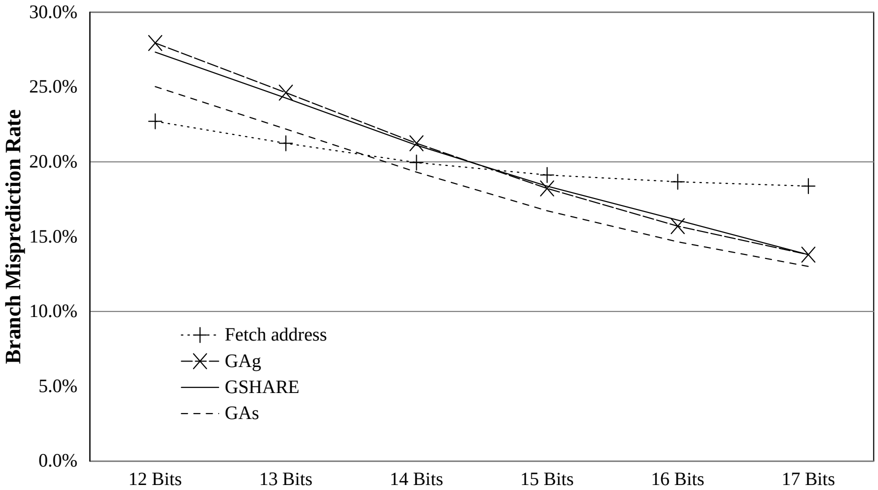

Figure 16: Branch misprediction rates on go for various PHT indexing schemes. The x-axis corresponds to the number of bits used to index the PHT

The results show that for go the GAs scheme outperforms the other three when the size of the predictor is 16K entries or larger. The fetch address scheme, which uses no global history, stops gaining accuracy as the number of entries starts to exceed the number of different dynamic branches encountered in the benchmark.

[2] Via a separate set of experiments, we arrived at a GAs scheme which uses three bits of address concatenated with the global history.

In figure 17, using a PHT with 32K entries (thus up to 15 bits of global history), we measured the performance (in instructions per cycle) of the four different indexing methods on the seven benchmarks. The results show that gshare and GAs attain the best performance, slightly beating the GAg scheme. The gshare technique outperforms the fetch address index by 18% on compress, 22% on li, 42% on m88ksim, and 16% on perl.

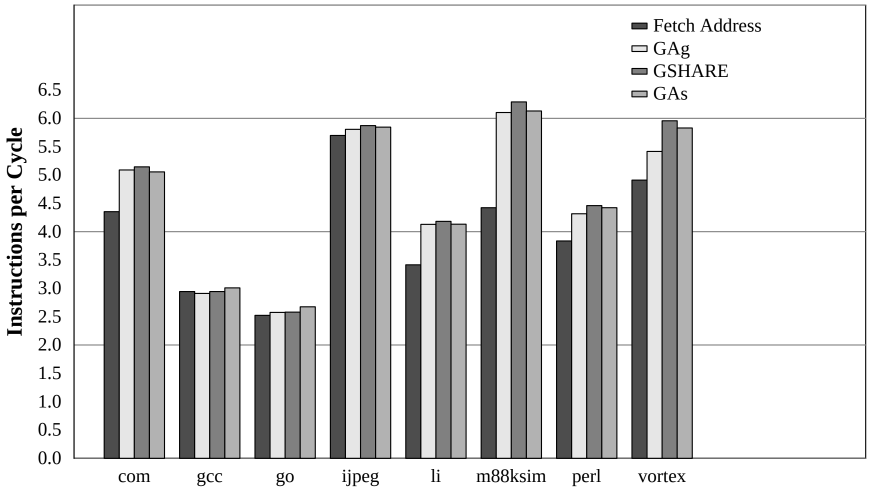

Figure 17: PHT indexing. A performance comparison of the four PHT indexing schemes discussed. In all cases the PHT has 32K entries.

5.9 Instruction Cache issues

Since we have relegated the instruction cache to a supporting role in a processor with a trace cache fetch mechanism, the design of the instruction cache should be revisited. All of the previous experiments have assumed that the instruction cache is dual ported. The addition of a second read port somewhat mitigates the partial fetch problem. By fetching two consecutive 64B lines from the instruction cache only a change of control flow will terminate a fetch before the full fetch width is utilized. However, this optimization is a costly addition to the fetch mechanism.

Figure 18 shows the results of removing the second read port from the instruction cache. There is very little impact on the performance if the instruction cache is scaled back in this way. Li shows the largest drop, losing slightly more than 2%. The performance on a number of the benchmarks actually increases as we go from a dual-ported to a single-ported instruction cache. This is due to the effect that fetching smaller lengths of instructions has on the fill unit. When a block of instructions fetched from the single-ported instruction cache is terminated because the end of the cache line is reached, the fill unit treats it atomically. This allows the fill unit to handle what would otherwise be a larger contiguous block of instructions as two separate entities. In this way a limited form of treating basic blocks non-atomically is beneficial on these few benchmarks.

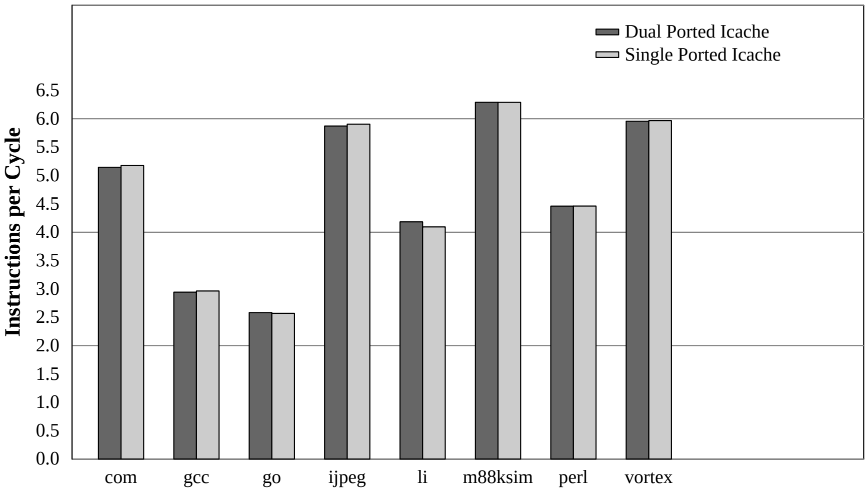

Figure 18: The effects of dual-porting the instruction cache.

A second area in which the decreased importance of the instruction cache may allow for a reduction of resources is in decoding instructions from the instruction cache. Because the instruction cache is accessed less frequently, we can reduce the hardware costs of the decoder and dependency analyzer by allowing them to take multiple cycles. Figure 19 shows the results of varying the number of non-pipelined delay cycles in accesses to the instruction cache from zero to three. Clearly, for the majority of the benchmarks, adding this latency has very little impact. The effects are significant, however, on gcc, go and to a lesser extent vortex, all of which have a significant number of trace cache misses. Between a zero-cycle and a three-cycle delay there is a 6.5%, an 11% and a 3% performance drop, respectively. A zero-cycle delay represents an ideal situation (equivalent to the configurations used in section 5.2) and our baseline configuration has a one-cycle delay. The performance difference between the ideal and our baseline is 3% for gcc, 4% for go, 3% for m88ksim, and 2% for vortex.

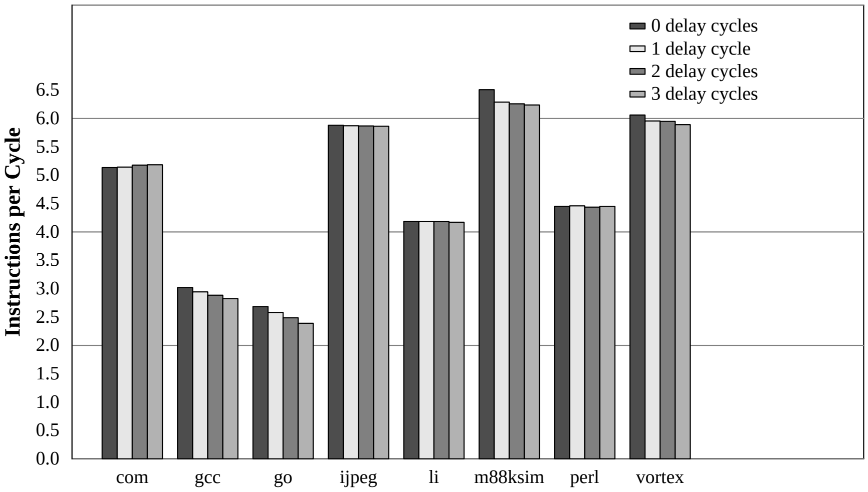

Figure 19: The effects of instruction cache delay on overall performance.

5.10 Comparison to previous work

Figure 20 compares the performance of our baseline configuration to the trace cache model investigated by Rotenberg et al. The large differences in performance, which amount to 23% on average, can be attributed to three major differences between our trace cache model and the Wisconsin configuration we simulated. The most obvious of these is the relative sizes of the trace cache and the instruction cache. Their configuration used a 4KB trace cache and a 128KB instruction cache. Although having a larger instruction cache allows the overall storage to be more efficiently utilized, our experiments have shown that this benefit is outweighed by the increased effective fetch rate of a large trace cache design. One difference between the two configurations that would have been improper to ignore is the associativity of the caches. As our baseline contains 4-way set associative caches, we chose to simulate the Wisconsin configuration using 4-way set associativity also. The second important difference between the two design points is our use of partial segment matching. Although our previous experiments have shown that this is a crucial aspect for large trace cache designs, it may be of less importance to fetch mechanisms with smaller trace caches.

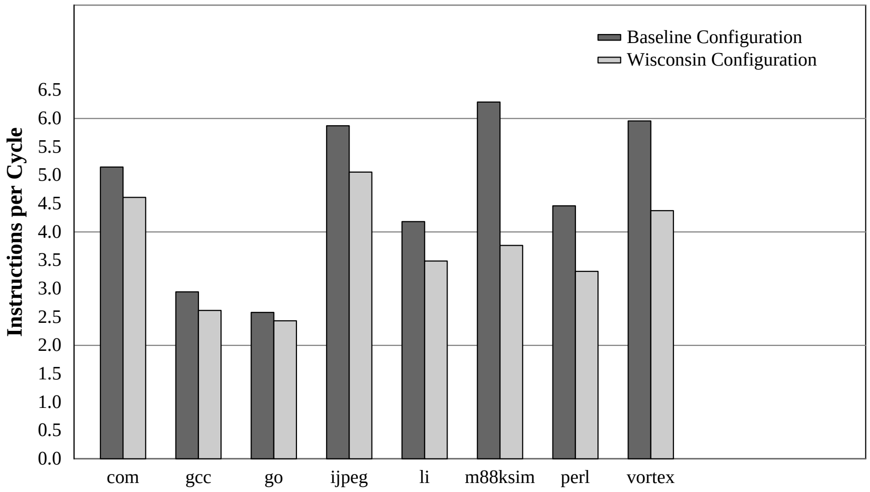

Figure 20: Comparison of the configuration proposed by Rotenberg et al and our baseline model.

The final difference between the two configurations is the branch predictor. Table 6 lists the misprediction rates of the two predictors. Rotenberg et al. implemented a GAg predictor, modified to perform three predictions per cycle. This predictor requires three read ports and three write ports. The predictor we have presented requires only a single read and a single write port, while providing a significant improvement in prediction accuracy.

Benchmark	TraceCache	Wisconsin
compress	8.002%	8.903%
gcc	9.015%	11.890%
go	18.218%	21.850%
ijpeg	7.479%	12.118%
li	4.620%	6.964%
m88ksim	1.500%	3.835%
perl	2.264%	5.893%
vortex	1.019%	4.531%

Table 6: The conditional branch misprediction rates for our proposed multiple branch predictor and the one proposed by Rotenberg et al

5.11 Comparison to an aggressive single block mechanism

The experiments thus far have evaluated various design options of the trace cache fetch mechanism. An assessment of the mechanism would not be complete without a comparison to the current dominant techniques for fetch engine design. Rotenberg et al. [19] presented a thorough comparison of the trace cache's performance on the SPECint92 and IBS benchmarks to a few of the hardware-based multiple block fetch techniques mentioned in section 2. Here we present a comparison of our baseline configuration with an aggressive single block fetch mechanism.

The components of the single block mechanism we modeled are each approximately the same size and access complexity as their trace cache counterparts in our baseline configuration. The single block mechanism consists of a single cycle, 128KB, 4-way set associative, dual-ported instruction cache capable of fetching up to 16 instructions or until the first control flow instruction each cycle. The next fetch address is generated with an ideal branch target buffer. The single branch predictor is a hybrid predictor, consisting of two components: a 15-bit PAs predictor and a 15-bit gshare predictor. The selection between the components is done by a 15-bit gshare-style selector. Combining a per-address predictor with a gshare predictor was first proposed by McFarling [13]. Using a two-level mechanism to select between the two was proposed by Chang et al [2]. A similar version of this predictor is implemented in the DECChip 21264 [8].

While the trace cache predictor is roughly twice the size (64KB) of the single branch predictor (32KB), the access times are roughly equivalent. Both predictors have 32K entries in their pattern history tables. The trace cache predictor has 16 bits in each entry, whereas the single branch predictor has three tables, each of two bits. The magnitudes of and differences in the widths of these entries make the access times very similar [26].

Figure 21 compares the performance of the trace cache and single block fetch mechanisms. The trace cache mechanism outperforms the single block scheme across all but one of the benchmarks simulated. On the benchmark compress, the difference is 22%, on gcc 5%, on ijpeg 20%, on li 39%, on m88ksim 51%, on perl 32%, and on vortex 30%. On go, there is a 5% degradation in performance. On average, the trace cache delivers 24% higher performance than an aggressive single block mechanism.

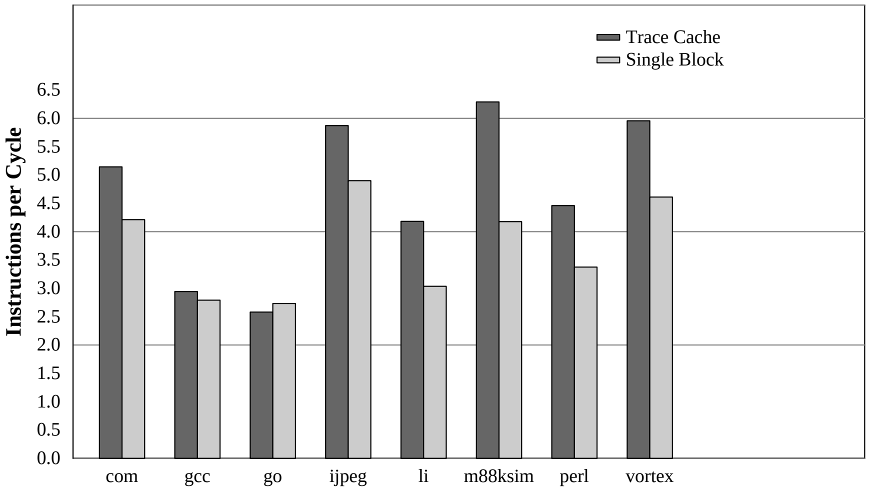

Figure 21: The performance of the baseline trace cache fetch mechanism, against an agressive single block fetch mechanism.

The large boost in performance of the trace cache mechanism comes from a significant increase in average effective fetch rate. Table 7 shows the effective fetch rates for both mechanisms; the average increase is 92%, indicating that the trace cache delivers almost two blocks per cycle.

Benchmark	TraceCache	SingleBlock
compress	10.565	5.792
gcc	8.395	4.453
go	8.656	5.544
ijpeg	11.839	7.010
m88ksim	10.259	4.933
li	9.337	4.157
perl	9.373	4.583
vortex	10.734	5.185

Table 7: The effective fetch rates for the trace cache mechanism and the single block mechanism.

While the increased fetch rate improves the performance of the trace cache mechanism, the losses due to branch mispredictions and, to a lesser extent, cache misses degrade performance. This explains the loss in performance on the go benchmark. Table 8 lists the conditional branch misprediction rates for the multiple and single branch predictors. The technology for predicting a single branch in one cycle is more mature than the technology for predicting multiple branches in one cycle. As the techniques and lessons learned with single branch predictors, techniques such as reducing negative interference [1] and combining branch predictors [13], are integrated into multiple branch predictors, the performance of the trace cache mechanism will continue to increase.

Benchmark	TraceCache	SingleBlock
compress	8.002%	5.273%
gcc	9.015%	5.361%
go	18.218%	12.965%
ijpeg	7.479%	6.076%
li	4.620%	3.131%
m88ksim	1.500%	1.166%
perl	2.264%	1.001%
vortex	1.019%	0.559%

Table 8: The conditional branch misprediction rates for the multiple branch predictor and the single block hybrid branch predictor

6 Conclusions

In this paper we have examined some of the critical design parameters of the trace cache fetch mechanism. The trace cache supplies multiple basic blocks of instructions each cycle by storing logically contiguous instruction sequences in physically contiguous storage — a concept first proposed by Melvin et al [15].

We have shown that a large trace cache assisted by a small instruction cache outperforms a small trace cache acting as an assist to a large instruction cache. With an instruction storage capacity of 132KB, a trace cache of 128KB with a an instruction cache of 4KB, outperforms the reverse configuration by 5% on gcc, 43% on m88ksim, 15% on perl, and 12% on vortex. Furthermore, since accesses to the instruction cache are infrequent with a trace cache of this size, the instruction cache can be designed less aggressively. The fetch mechanism can tolerate some latency along the access path to the instruction cache.

Since the heart of the trace cache is its ability to fetch multiple basic blocks each cycle, an effective multiple block branch predictor is critical to its performance. The branch predictor we have presented accurately predicts up to three conditional branches each cycle while requiring only a single read and a single write port. Although this predictor performs respectably, there is still much room for improvement. Multiple branch predictors are still in their infancy compared to techniques for single branch predictors. We expect that many of the techniques used for single branch predictors, such as reducing negative interference and combining predictors, will be successfully applied to multiple branch predictors. As these techniques develop, the performance of the trace cache will certainly increase.

We have also demonstrated the sensitivity of the trace cache to conflict misses, and that the associativity of the trace cache has a large impact upon the performance of several of the SPECint95 benchmarks. The improvement in performance of a 4-way set associative trace cache over a direct mapped trace cache is 15% for gcc and vortex, and 12% for perl. Our path associativity experiments have shown modest improvements of 4% on go and 2% on compress and m88ksim if the trace cache can store multiple segments that start with the same block.

One very important trace cache design option is the ability to partially match a segment. Our experiments have shown that without this feature the average performance across the benchmarks drops 25%.

Other experiments have shown that there is a slight advantage to having the fill unit collect blocks as they are retired rather than at issue time. Furthermore, we have shown that the trace cache is extremely tolerant to latency within the fill unit. This could prove exceptionally useful as the fill unit may be able to do complex runtime analysis and optimizations, allowing the trace cache to store segments of instructions that have been optimized or translated for the specific execution core it is feeding.

In figure 22 we show the performance gained from taking the most important of these aspects and combining them into a single configuration. This configuration differs from our baseline in three respects: the trace cache is path associative, the latency of the instruction cache has been reduced to a single cycle, and the branch predictor uses a GAs indexing scheme. Although they have a limited impact on most of the benchmarks, these changes combine to increase the performance of gcc by 9% and go by 11%.

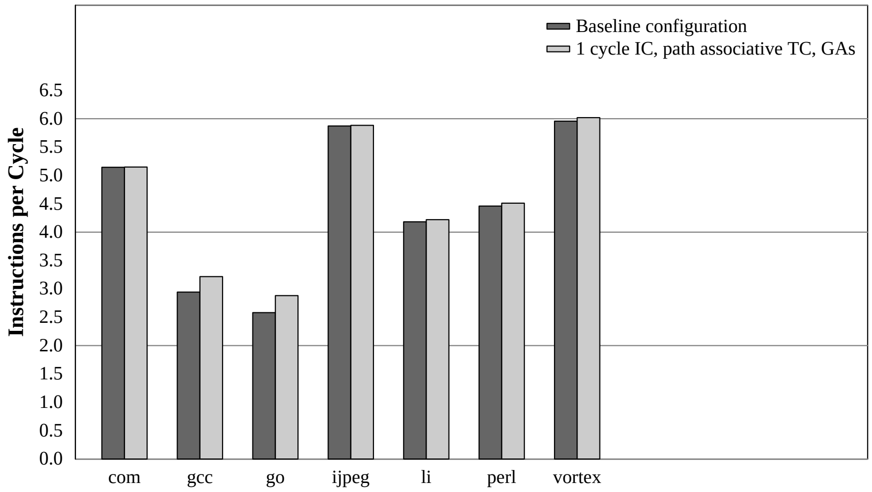

Figure 22: A comparison of our baseline configuration and one in which the trace cache is path associative, the instruction cache has a single-cycle latency, and the branch predictor indexes the PHT with a GAs method.

Finally, when compared with an aggressive single block fetch mechanism, the trace cache attains an average performance increase of 24%. Much of this performance increase comes from the increase in effective fetch rate, which is 92% greater than that of the single block engine.

References

[1] P.-Y. Chang, M. Evers, and Y. N. Patt, "Improving branch prediction accuracy by reducing pattern history table interference," in *Proceedings of the 1996 ACM/IEEE Conference on Parallel Architectures and Compilation Techniques*, 1996.

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

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

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

[5] S. Dutta and M. Franklin, "Control flow prediction with tree-like subgraphs for superscalar processors," in *Proceedings of the 28th Annual ACM/IEEE International Symposium on Microarchitecture*, pp. 258–263, 1995.

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

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

[8] L. Gwennap, "Digital 21264 sets new standard," *Microprocessor Report*, pp. 11 – 16, October 1996.

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

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

[11] *Pentium Processor User's Manual Volume 1: Pentium Processor Data Book*, Intel Corporation, 1993.

[12] D. Kaeli and P. Emma, "Branch history table predictions of moving target branches due to subroutine returns," in *Proceedings of the 18th Annual International Symposium on Computer Architecture*, 1991.

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

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

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

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

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

[18] P. B. Racunas and Y. N. Patt, "Achieving full associativity with direct-mapped access times using a remap cache," Unpublised manuscript, 1997.

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

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

[21] J. E. Smith, “A study of branch prediction strategies,” in *Proceedings of the 8th Annual International Symposium on Computer Architecture*, pp. 135–148, 1981.

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

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

[24] G. S. Tyson, “The effects of predication on branch prediction,” in *Proceedings of the 27th Annual ACM/IEEE International Symposium on Microarchitecture*, pp. 196–206, 1994.

[25] S. Wallace and N. Bagherzadeh, “Multiple branch and block prediction,” in *Proceedings of the 1997 ACM/IEEE Conference on High Performance Computer Architecture*, 1997.

[26] S. Wilton and N. Jouppi, “An enhanced access and cycle time model for on-chip caches,” in *DEC Western Research Lab. Technical Report 93/5*, 1994.

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

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

[29] R. Yung, “Design decisions influencing the ultrasparc’s instruction fetch architecture,” in *Proceedings of the 29th Annual ACM/IEEE International Symposium on Microarchitecture*, 1996.