The Block-based Trace Cache

Bryan Black, Bohuslav Rychlik, and John Paul Shen
Department of Electrical and Computer Engineering
Carnegie Mellon University
Pittsburgh, PA 15213
{black,bohuslav,shen}@ece.cmu.edu

Abstract

The trace cache is a recently proposed solution to achieving high instruction fetch bandwidth by buffering and reusing dynamic instruction traces. This work presents a new block-based trace cache implementation that can achieve higher IPC performance with more efficient storage of traces. Instead of explicitly storing instructions of a trace, pointers to blocks constituting a trace are stored in a much smaller trace table. The block-based trace cache renames fetch addresses at the basic block level and stores aligned blocks in a block cache. Traces are constructed by accessing the replicated block cache using block pointers from the trace table. Performance potential of the block-based trace cache is quantified and compared with perfect branch prediction and perfect fetch schemes. Comparing to the conventional trace cache, the block-based design can achieve higher IPC, with less impact on cycle time.

Results: Using the SPECint95 benchmarks, a 16-wide realistic design of a block-based trace cache can improve performance 75% over a baseline design and to within 7% of a baseline design with perfect branch prediction. With idealized trace prediction, it is shown the block-based trace cache with an 1K-entry block cache achieves the same performance of the conventional trace cache with 32K entries.

1   Introduction

Most technologists anticipate the continuation of Moore's law of increasing chip density and complexity for another 10 years. However, existing superscalar techniques for harvesting instruction-level parallelism (ILP) are encountering strong diminishing returns. In order to justify building wider superscalar processors, new microarchitecture techniques capable of achieving significantly higher IPC (average instructions executed per cycle) for ordinary programs are essential.

1.1   High Bandwidth Instruction Fetching

A critical challenge to achieving high IPC is supplying enough useful instructions for execution. High-bandwidth instruction fetching must address several problems: First, the machine must perform multiple branch predictions in every cycle. Second, the fetch engine must be able to fetch from multiple non-contiguous addresses in every cycle, reaching beyond the taken branch. Third, misaligned instructions, from the multiple fetch groups, must be collapsed into the fetch buffer.

1.2   Previous Related Work

The three challenges in high-bandwidth instruction fetching: multiple-branch prediction, multiple fetch groups, and instruction alignment and collapsing, have been addressed in previous studies. These previous works fall into two general categories, those that use an enhanced instruction cache [3][16][18][19] and those that use a trace cache [5][8][14][15]. The fundamental difference between them is where in the pipeline instructions are aligned and collapsed. Both perform multiple-branch prediction the cycle before instruction fetch, and update the predictor at completion. Techniques that use an enhanced instruction cache support fetch of non-contiguous blocks with a multi-ported, multi-banked, or multiple copies of the instruction cache. This leads to multiple fetch groups that must be aligned and collapsed at fetch time, which can increase the fetch latency. The conventional trace cache performs all instruction alignment and collapsing at completion time. At completion time a fill unit constructs traces of instructions to be stored in the trace cache. The conventional trace cache optimizes the fetch latency by shifting the complexity of multiple-branch prediction, multiple fetch groups, and instruction alignment and collapsing to the completion time.

This work proposes a block-based trace cache that is able to achieve higher IPC performance while requiring less

trace storage capacity. It also facilitates more flexible trace prediction schemes; for example, partial matching [5] can be easily and naturally implemented. Furthermore, it has the potential of enabling clean implementation of other dynamic optimizations, such as multi-path execution, dynamic predication and dynamic multithreading.

The block-based trace cache employs a number of novel features. Instead of explicitly storing instructions of a trace, pointers to blocks constituting a trace are stored in a much smaller trace table. The block-based trace cache renames fetch addresses at the basic block level and stores aligned blocks in a block cache. Traces are constructed by accessing the replicated block cache using block pointers from the trace table.

Figure 1 and Figure 2 illustrate the basic difference between the conventional and the block-based trace caches. Both designs perform next trace prediction in the cycle prior to the fetch cycle. The conventional design employs a sophisticated path-based next trace predictor [8] to produce the next trace identifier which is used in the fetch cycle to access the trace cache. Similarly, the block-based trace cache makes the next trace prediction by accessing the trace table which outputs the identifiers of the blocks in the predicted trace. These block identifiers are used in the fetch cycle to access the replicated block cache. Section 3.1.3 will show the accessing of the replicated block cache does not increase the fetch latency of the block-based design.
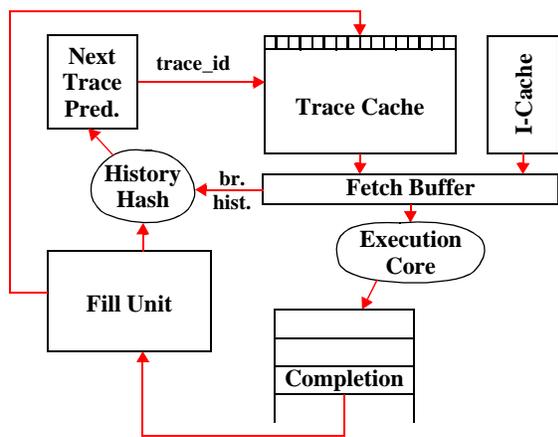

Figure 1 - The conventional trace cache.

This paper starts with a discussion about fetch address renaming in Section 2, then uses Section 3 to detail the implementation and design trade-offs of the block-based trace cache. Section 4 discusses the simulation methodology for this work, and Section 5 explores the design space quantitatively. Performance potential of the block-based trace cache is quantified and compared with that of perfect branch prediction and perfect fetch schemes in Section 6. It is shown that the block-based trace cache achieves IPC higher than

that of perfect branch prediction and approaching that of perfect fetch if an idealized trace predictor is used. Section 7 compares the block-based trace cache to the conventional trace cache implementation, and demonstrates a much higher IPC when trace storage capacity is limited.

2   Fetch Address Renaming

The implementation of the block-based trace cache is based on the concept of *fetch address renaming*. This section presents the motivation for renaming fetch addresses, suggests possible options for the renamed entity, and argues that the basic block is a convenient and effective entity for renaming.

2.1   Motivation for Renaming

Traditionally the effective address of an instruction is used for its fetching. Such addresses contain many bits. An instruction cache big enough to support the full decoding of these bits is impractical and not necessary. Typically a subset of the fetch address bits are used to index into the instruction cache which results in potential aliasing. The aliasing problem is solved by storing a tag in each cache line and using tag compare to ensure the correct group of instructions is being fetched.

Fetch addresses can be renamed or translated to use fewer bits. If a unique renamed pointer can be assigned to each fetch address then aliasing can't occur and tag compare is not necessary. This can reduce the latency of instruction fetch. Fetch address renaming requires a table to translate instruction fetch addresses to their renamed pointers. Such a rename table, maintained at completion time, effectively moves the complexity and latency of associative search and tag compare from instruction fetch time to completion time. Since the rename table is of finite size, there can be capacity misses in the rename table. A replacement policy is needed for the rename table, and an efficient mapping of fetch addresses to the renamed pointers is crucial.

2.2   Possible Renamed Entities

Some form of fetch address renaming is always useful, because it removes the tag compare of cache access during instruction fetching. The key question is what should be the entity for renaming. There are at least three possibilities.

Fetch address renaming can be done at the level of instruction cache lines, as explored by [2][18]. Instead of using the fetch address, the instruction cache line index is used for fetching. This reduces the latency of instruction cache access, with no fragmentation since the entire cache line is renamed as an entity. This renaming scheme assumes the

traditional instruction cache, which may contain non-aligned basic blocks. This will necessitate trace construction at fetch time and a collapsing buffer [3] when multiple non-contiguous blocks are fetched.

It is also possible to perform renaming at the level of traces as suggested by a recent paper on next trace prediction [8]. Such renaming can reduce the latency of the trace cache fetch. The conventional trace cache does not require trace construction at fetch time [5][8][14][15]. However, the trace is not a fundamental unit of operation in terms of program semantics. The definition of a trace can vary and can be somewhat arbitrary depending on the trace construction heuristics used in the fill unit. This makes it more difficult to implement some of the advanced dynamic optimizations and necessitates very good trace selection heuristics.
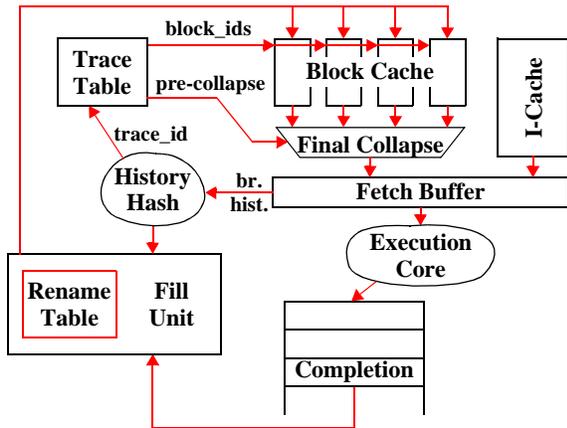

Figure 2 - The block-based trace cache.

This work proposes renaming at the basic block level, because the basic block is a logical unit of program execution [6][11]. Similar to other forms of fetch address renaming, basic block renaming reduces the latency of accessing the cache during instruction fetch. Renaming at the level of basic blocks can facilitate dynamic optimizations such as partial match of traces, dynamic predication, multi-path execution and dynamic multithreading. It can also ease the implementation complexity of register renaming and the reorder buffer. However, renaming at the basic block level can potentially introduce more fragmentation and replication. It also requires trace construction and instruction collapsing at fetch time. These problems are addressed in Section 5.

3   Block-based Trace Cache

The machine organization of the block-based trace cache is similar to that of the conventional trace cache [5][8][14][15]. Both support the same superscalar execution core, shown in Figure 1 and Figure 2. The key differ-
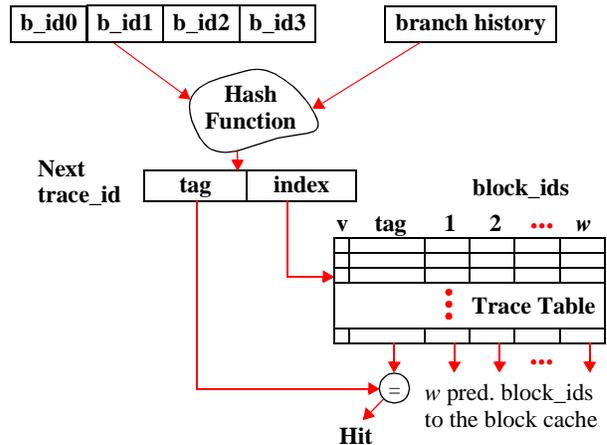

Figure 3 - Logical diagram of the next trace predictor and the trace table.

ence is the storage of traces. The conventional design shifts most of the complexity and latency of instruction fetching to completion time using a fill unit. The block-based design brings back a portion of that complexity to improve the efficiency of trace storage and the flexibility of trace construction. The block-based trace cache has four major components: the *trace table*, the *block cache*, the *rename table* and the *fill unit*. Instead of storing the instructions from multiple basic blocks in a contiguous trace, the block-based design stores aligned basic blocks separately in the block cache. The trace table stores the renamed pointers (called block_ids) to these basic blocks for trace construction. The block-based trace cache shifts the potential static-to-dynamic explosion of the number of traces to the much smaller trace table. One disadvantage of the block-based trace cache is the addition of the final collapse MUX in the fetch stage. It is shown later that the final collapse MUX if implemented carefully will add little fetch latency.

The next three subsections present details of the trace table, the block cache, and the rename table. Since the fill unit controls the update of each of these three components, the interaction between the fill unit and each of the components is discussed in each subsection.

3.1   Trace Table

The trace table can be viewed as storing a short-hand representation of traces since each of its entries contains the block-ids of a trace. The trace table can also be viewed as part of the next trace predictor. The next trace predictor uses block-id execution history and branch history bits to generate the predicted trace_id, which is used to access the trace table. The entry of the trace table indexed by the predicted trace_id stores the block_ids or pointers to the blocks constituting the predicted trace; see Figure 3. A trace table line consists of a valid bit, a tag, and the w block_ids of the pre-

dicted trace. These block_ids are used to access the replicated block cache during the fetch cycle. The trace table also provides pre-generated steering bits for the final collapse MUX (Section 3.1.3).

3.1.1 Next trace prediction

The next trace_id is generated using a hashing function to reduce branch history bits and previously executed block_ids to the predicted trace_id. The input to the hashing function can be based solely on the last block_id and h bits of global branch history (gshare [10]), or a combination of previous block_ids and branch history bits (next trace predictor of [8]). Since the fetch unit can track block_id history and branch history, potentially speculative previous block_ids and branch history bits can be used by the hashing function. As shown in [9], speculative branch history can be corrupted due to mispredictions; a simple recovery and cleanup mechanism is necessary. The associativity and size of the trace table are explored in Section 5. However, detailed exploration of the huge design space for the next trace predictor is left for future work.

3.1.2 Trace table read.

In the cycle prior to the fetch cycle, block_id history and branch history are hashed to produce the predicted next trace_id which is used to index into the trace table. The sequence of block_ids retrieved from the trace table represents the predicted trace. The predicted sequence of block_ids are then used, in the fetch cycle, to access the block cache to produce the predicted trace of instructions. The trace table and the block cache are accessed in two separate cycles in a pipelined fashion. A miss in the trace table access invokes access from the instruction cache.

3.1.3 Trace table fill

The trace table is filled at completion time as blocks of instructions are completed. As the fill unit constructs a trace of executed blocks it writes back the block_ids to the trace table. Only one writeback to the trace table is supported per cycle. The fill unit also performs pre-collapsing by generating a set of steering bits based on the actual number of instructions in each block of the trace. These steering bits are stored in the trace table along with the block_ids and are used at fetch time to control the final collapse MUX. This reduces the collapsing latency incurred at fetch time to the data propagation delay of a MUX or a shallow MUX tree.

3.2 Block Cache

The block cache stores aligned instruction blocks. To support multiple simultaneous accesses, the block cache is replicated. In the current implementation all copies are written with the same content. (Through more complex block renaming, future implementations can make more efficient use of the replicated block cache.) Figure 4 illustrates the replicated block cache and shows the details of one of the block cache copies. Real basic blocks from a program have varying sizes. This work assumes that the block cache has fixed-size entries. A logical basic block that exceeds this size is partitioned into multiple physical blocks. The block cache stores the N most recently used blocks. Each line of the block cache stores up to b instructions, along with the fetch address of the first instruction. The block cache is a direct mapped cache with no tag compare needed at fetch time. Replacement is controlled by the fill unit and the rename table (Section 3.3). The most appropriate block size (b) and replication count (w) of the block cache are examined in Section 5.
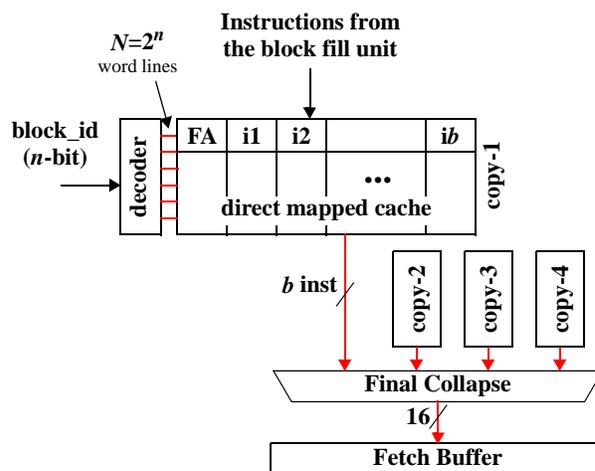

Figure 4 - Logical diagram of the replicated block cache, with final collapse MUX.

3.2.1 Block cache read

The block cache is read at fetch time. The trace table (Section 3.1) provides w block_ids to index the w copies of the block cache. The block cache copies are accessed in parallel each providing up to b instructions. Due to renaming of the fetch address of each block, a block_id uniquely identifies a particular block. Consequently, no aliasing can occur and no tag compare is needed even though the block cache is direct mapped. A final collapse MUX, controlled by steering bits retrieved from the trace table, is used to construct the predicted trace of instructions. After the instructions are loaded into the fetch buffer, instruction execution is identical to the standard out-of-order machine.

The fetch address (FA) of each block that is stored in a block cache line is used during branch execution for trace prediction validation and misprediction recovery. Since the fetch address of each block is stored in the block cache, misprediction can be determined at the block level. If the predicted trace is only partly correct, partial matching [5] is

easily implemented. Once a misprediction is detected, correct fetching from the instruction cache can be performed in the next cycle.

3.2.2 Block cache fill

The block cache is filled at completion time by the fill unit. The fill unit constructs physical blocks from logical basic blocks and inserts them into the block cache. Block construction is accomplished by a fill buffer. The fill buffer captures instructions in order as they complete. The end of a block occurs when a branch or other control flow instruction is encountered or when the physical block size (b) is reached. Constructing physical blocks from logical basic blocks produces fragmentation when the logical basic block size is not an integral multiple of the physical block size. This internal fragmentation can cause a performance loss, and is studied in Section 5.1.

Once a block boundary is detected, the fill unit renames the block with a block_id and writes it to the block cache in the entry indexed by that block_id. To prevent wasteful insertion and replacement in the block cache, the fill unit detects already cached blocks and does not attempt to insert them into the block cache a second time. It is possible that during block construction a completing block may contain instructions that belong to an already renamed block; the fill unit will only rename the portion of the completing block that has not been seen before.

3.3   Rename Table

The rename table implements the fetch address renaming of basic blocks (Section 2.2). The rename table is a set-associative mapping of instruction fetch addresses to block_ids. Figure 5 illustrates an eight entry 2-way set associative rename table. Each entry corresponds to a particular block_id number. The index portion of the fetch address and the LRU replacement policy determine which entry a newly renamed fetch address will be mapped to. The block_id of the newly allocated entry is returned as the renamed value.

The rename table may use any associativity scheme. However the number of entries (N) is equal to the number of entries of the block cache because each entry of the set-associative rename table renames one entry of the block cache.

3.3.1   Rename table read

The rename table is used at fetch time to detect if the requested fetch address corresponds to a block already stored in the block cache. Concurrent with the instruction cache access, the fetch unit accesses the rename table using the fetch address. If the rename table access returns that the current fetch group is already renamed, the returned block_id

is recorded in the fetch history and used in the next cycle to access the block cache. The fetch unit maintains a speculative history of block_ids fetched and the associated branch history bits, which are hashed to perform next trace prediction.

To reduce rename table bandwidth requirements we limit the rename table to one read access per cycle. Hence, during an instruction cache access, the fetch unit is only allowed to fetch at most one basic block. To facilitate this a single predecode bit in the instruction cache can be used to mark each instruction as branch or non-branch.
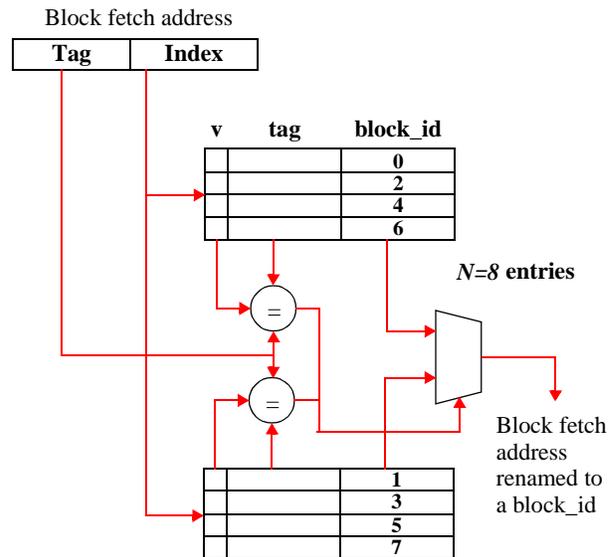

Figure 5 - Example implementation of the rename table (8 entries, 2-way set associative).

3.3.2   Rename table fill

At completion time the fill unit updates the rename table. When a new block of instructions is constructed, the rename table is written with the fetch address of the first instruction in the block. The renamed block_id is returned and used to update the block cache entry. To limit the number of ports required in the rename table, only one block of instructions can be renamed in a single cycle. This does not limit completion bandwidth to one block, because blocks that are already renamed are not written to the rename table again, and therefore do not require rename table access at completion time. When an existing block_id is renamed, traces already stored in the trace table that include this block_id become incorrect. Since it is difficult to search the trace table to invalidate these traces, stale block_ids are detected during execution and treated as mispredictions; see Section 3.2.1.

4   Experimental Methodology

All the experimental data reported in this paper are generated by a full-functional performance simulator that is built based on a validated PowerPC 604 simulation model [1], which is based on published reports [4][7][17] and accurately models all key features of the microarchitecture.

4.1   Machine Model

To focus the current study on instruction fetching and to highlight the impact of instruction availability on machine performance, the PowerPC 604 microarchitecture is extended to remove resource constraints, and widened to utilize more instruction bandwidth.

A centralized reservation station with 512 entries and unlimited out-of-order issue bandwidth is assumed. This effectively limits the instruction window to 512 instructions. An unlimited number of functional units is also assumed. The instruction fetch, dispatch, and completion bandwidth is increased to 16 instructions per cycle. The memory hierarchy is fully modeled with a perfect main memory, a 32KB Level-1 I-cache, a 32KB Level-1 D-cache, and a 256KB unified Level-2 cache. Access latencies are 1, 3, and 100 cycles for the L1, L2 caches and the main memory, respectively. On-chip implementation of the L2 is assumed. An unlimited load miss queue and an unlimited store queue handle all load and store execution. The store queue performs data forwarding, and load/store instructions execute out-of-order if no address aliasing is detected.[1]

All register and memory data dependencies are enforced. Instruction execution latencies can be found in [7], and accurately reflect the PowerPC 604. The potential bottlenecks of this machine are the data flow limit due to true data dependencies and instruction availability.

4.2   Benchmarks

The benchmark set used is the SPECint95 suite, compiled by gcc 2.7.2. To reduce simulation time, we use small input files and limit run length to 200 million instructions for each benchmark, totaling 1.6 billion instructions. All user library calls are modeled, though system calls are not.

---

[1] The authors are not proposing this as a realistic machine design, but a machine model that focuses the performance bottleneck on instruction fetch while enforcing register and memory data dependencies and using realistic trace prediction.

5   Design Space Exploration

This section explores the design space of the block-based trace cache. The block size (b), block cache replication (w), block cache entry count or size (N), rename table associativity, and trace table size and associativity are explored. We attempt to systematically narrow the design space search via a series of experiments and analyses.

5.1   Block Cache Fragmentation (b)

Fragmentation of the instruction fetch group is an issue for all trace cache implementations. Since the block-based trace cache can incur fragmentation at the block level, there is potential for greater trace-level fragmentation as compared to a conventional trace cache. The optimal block size (b) for the block cache should minimize the total fragmentation. To study the fragmentation for different block sizes, the block-based trace cache is simulated with realistic branch prediction and a fixed number of block cache entries, while varying the block size (b) and block cache replication (w). The maximum potential fetch bandwidth is equal to b x w and is fixed at 16 instructions (except for b=6 and w=3). Fragmentation is measured in terms of the utilization of the fetch buffer which is related to the actual fetch bandwidth achieved. Figure 6 shows the effects of varying the block size (b) on the fetch buffer utilization.

For both b=16 and b=8 there is significant drop-off of fetch buffer utilization for some benchmarks indicating significant fragmentation. For two of the benchmarks b=2 is the best choice. b=4 is best for one benchmark and b=6 is best for two. The key observation is that the fetch buffer utilizations for b=2, 4, and 6 are all quite close for most of the benchmarks. b=2 is not desirable due to high replication (i.e. w=8). Based on harmonic means b=6 is the best choice with b=4 a close second. Both remain as potential candidates for optimal b.
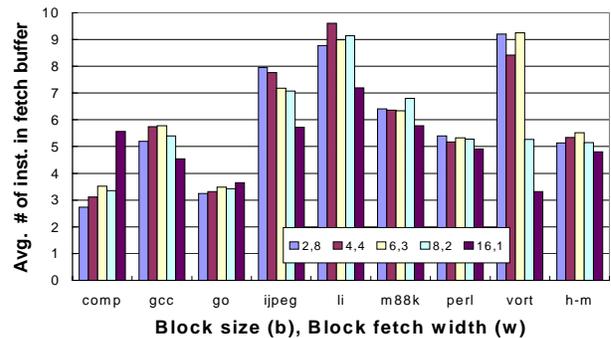

Figure 6 - Block cache fragmentation analysis. Block size vs. block cache replication trade-off. (N=4096)

5.2 Block Cache Replication (*w*)

By increasing the number of blocks fetched in a cycle, the size of the total fetch window can be greater than the size of the fetch buffer to compensate for the fragmentation in each block. Figure 4 illustrates the use of *w* copies of the block cache to increase the total available instructions, where *w* can be greater than 16/*b*. The final collapsing MUX removes empty entries within the individual blocks, and yields a total fetch group that fits into the fetch buffer.

Realistic branch prediction is used in these simulations to factor in the effect of decreasing trace prediction accuracy as *w* increases. A block cache size of *N*=4096 entries is used for each copy of the block cache. In Figure 7 *w,* the number of block cache copies, is varied. Notice that the fetch buffer utilization does not necessarily increase for a larger number of block cache copies. This is due to the increasing misprediction rate for predicting longer sequences of blocks; see Section 5.4. For some benchmarks *w*=16 is suffering significant misprediction penalty. Figure 7 shows that for most benchmarks, *w*=5 and *w*=6 are the best choices when *b*=4. For *b*=6, the best choices are *w*=4 and *w*=5. In both cases, the two choices are very close. In an effort to minimize replication (*w*), for the remainder of our study we select a block size of *b*=6 and *w*=4 copies of the block cache as our optimal replicated block cache organization.
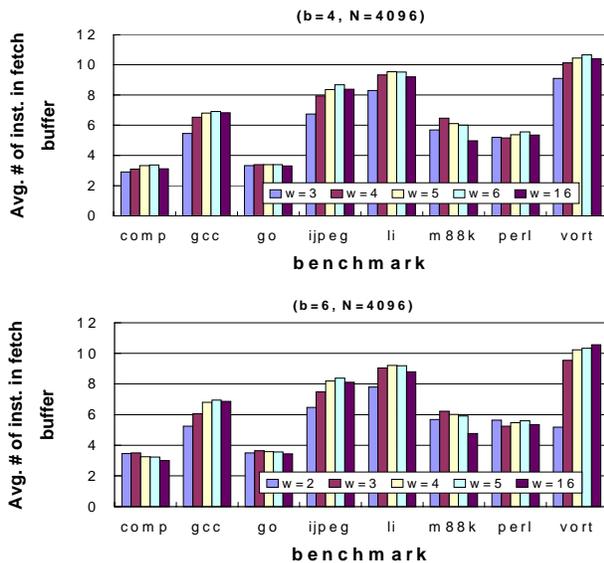

Figure 7 - Block cache replication analysis; exploration of (*w*).

5.3 Rename Table Associativity

The number of entries in the rename table *(N)* is equal to the number of entries in the block cache, which is explored in Section 6. For this analysis a block size of *b*=6, a block

cache replication of *w*=4, and a block cache size of *N*=256 entries are used. The number of block cache entries is intentionally reduced to emphasize the effects of associativity in smaller block cache sizes. To remove the effects of branch prediction a perfect predictor is used. Figure 8 shows the hit rates for the rename table with associativity of 1, 2, 4, 8, and full. For most of the benchmarks, the hit rate seems to level off at an associativity of 4 or 8. For our study a rename table associativity of 4 is used.
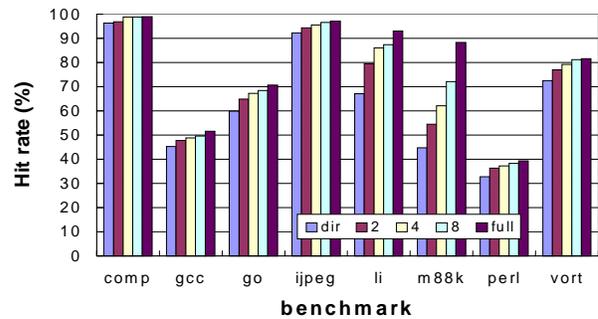

Figure 8 - Rename table associativity analysis. (b=6, w=4, N=256)

5.4 Trace Table

The first design feature associated with the trace table is the hashing function used in the next trace predictor. As discussed earlier, exploring the large design space for the best hashing function is outside the scope of this work. For this study only one hashing function is considered. The hashing function is chosen after a brief preliminary analysis. The next (predicted) trace_id is the concatenation of the last block_id, the last 3 branch directions, and the block_id fetched 3 blocks earlier. This hashing function is capturing a combination of path and branch history. The two remaining design parameters of the trace table are size and associativity. Figure 9 and Figure 10 measure the hit rate of the
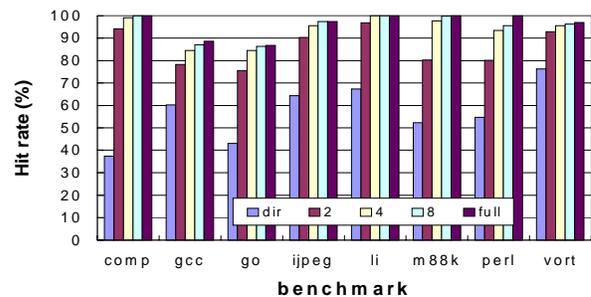

Figure 9 - Associativity analysis of the trace table. (b=6, w=4, N=4096)

trace table, with block cache parameters: *b*=6, *w*=4, and *N*=4096. Based on the results in these figures a trace table

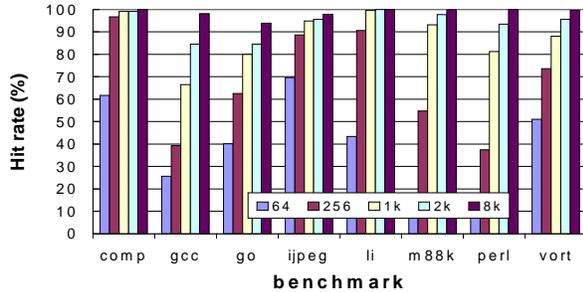with 4-way set associativity and a total of 8192 entries is a good choice.

Figure 10 - Size analysis of the trace table. (b=6, 4=4, N=4096, assoc=4)

Figure 11 shows the effectiveness of the resultant next trace predictor described above. The bars show the percentage of cycles at least 1, 2, 3, or 4 blocks are correctly predicted. The prediction rate starts quite high for the first block and quickly drops off as more blocks are predicted. The no prediction part of the stacked bars includes the number of cycles no prediction is made because there is no space in the fetch buffer for the predicted block. This fast diminishing return in accuracy for predicting multiple blocks of a trace, is due to the compounding effects of incorrect predictions, and shows the importance of keeping the replication of the block cache low.
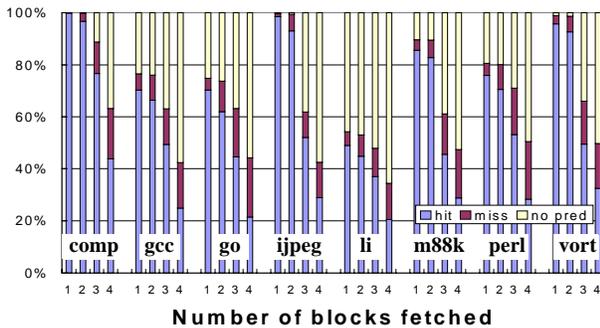

Figure 11 - Rate blocks are fetched from the block cache. (b=6, w=4, N=4096)

6   Block-based Trace Cache Performance

We evaluate the performance of the block-based trace cache by comparing its IPC to: 1) the 16-wide baseline PowerPC 604 design (base), 2) the baseline machine with perfect branch prediction (perfect_branch), and 3) the baseline machine with perfect fetch (perfect_fetch). Perfect_branch predicts all conditional branch directions with 100% accuracy but can only fetch up to the first taken branch instruction or I-cache line boundary. Perfect_fetch

can perfectly predict and fetch beyond any number of branches and cross I-cache line boundaries. It is limited by the number of free entries in the fetch buffer. Figure 12 compares perfect_fetch (top straight line), perfect_branch (middle straight line), and base (bottom straight line) to a block-based trace cache with perfect trace prediction (perfect_block), and to a block-based trace cache with realistic trace prediction (real_block). The block cache size (N) is varied from 16 to 16384 with a 4-way set-associative rename table. The block size is b=6 and the block cache replication is w=4. Perfect_block predicts the next block successfully if it resides in the block cache. Real_block uses the hashing function and the next trace predictor described in Section 3.1.1, with a trace table size of 8192 at 4-way set-associative.

Looking at the results in Figure 12, for each benchmark the IPC curve of perfect_block (higher curve) always exceeds that of perfect_branch given enough entries in the block cache (256-512 entries for most benchmarks) and approaches the performance of perfect_fetch when the number of entries reaches 16K. This suggests the block cache is adequately capturing the working set of these benchmarks. For four of the eight benchmarks, real_block IPC (lower curve) actually exceeds that of perfect_branch with only a 2K-entry block cache. This suggests that perfectly predicting just the first taken branch is not enough. Being able to make multiple-branch predictions and fetch from multiple predicted targets is very beneficial. *Go* is the most challenging benchmark. Its perfect_fetch IPC (~12) is quite high, which suggests shorter data dependency chains than one with lower IPC. However, the IPC achieved by real_block is relatively low (~2.6) even with a block cache of 16K entries. This is an indication of the unpredictable branching behavior of *go* and the lack of a small number of dominant traces. On the other hand *gcc,* with perfect_fetch IPC of ~8, contains longer data dependency chains, while demonstrating a more repetitive trace behavior with a real_block IPC of ~4.0.

The final graph of Figure 12 presents the harmonic mean of the eight benchmarks. The results indicate that the block-based trace cache can attain an average IPC of 3.95 for the benchmark suite with a block cache of only N=4096 entries and a realistic next trace predictor (real_block). This represents a 68% improvement over the baseline design and is within 12% of perfect branch prediction performance. Increasing N to 16384 the block-based trace cache outperforms the baseline by 75% and comes within 7% of perfect_branch, achieving 4.11 IPC. This can be increased to 7.6 IPC if a perfect next trace predictor (perfect_block) is employed. This suggests there is significant headroom for better next trace predictor designs.

Figure 12 also explains how the IPC is being lost for the different regions of the plot. On average this benchmark
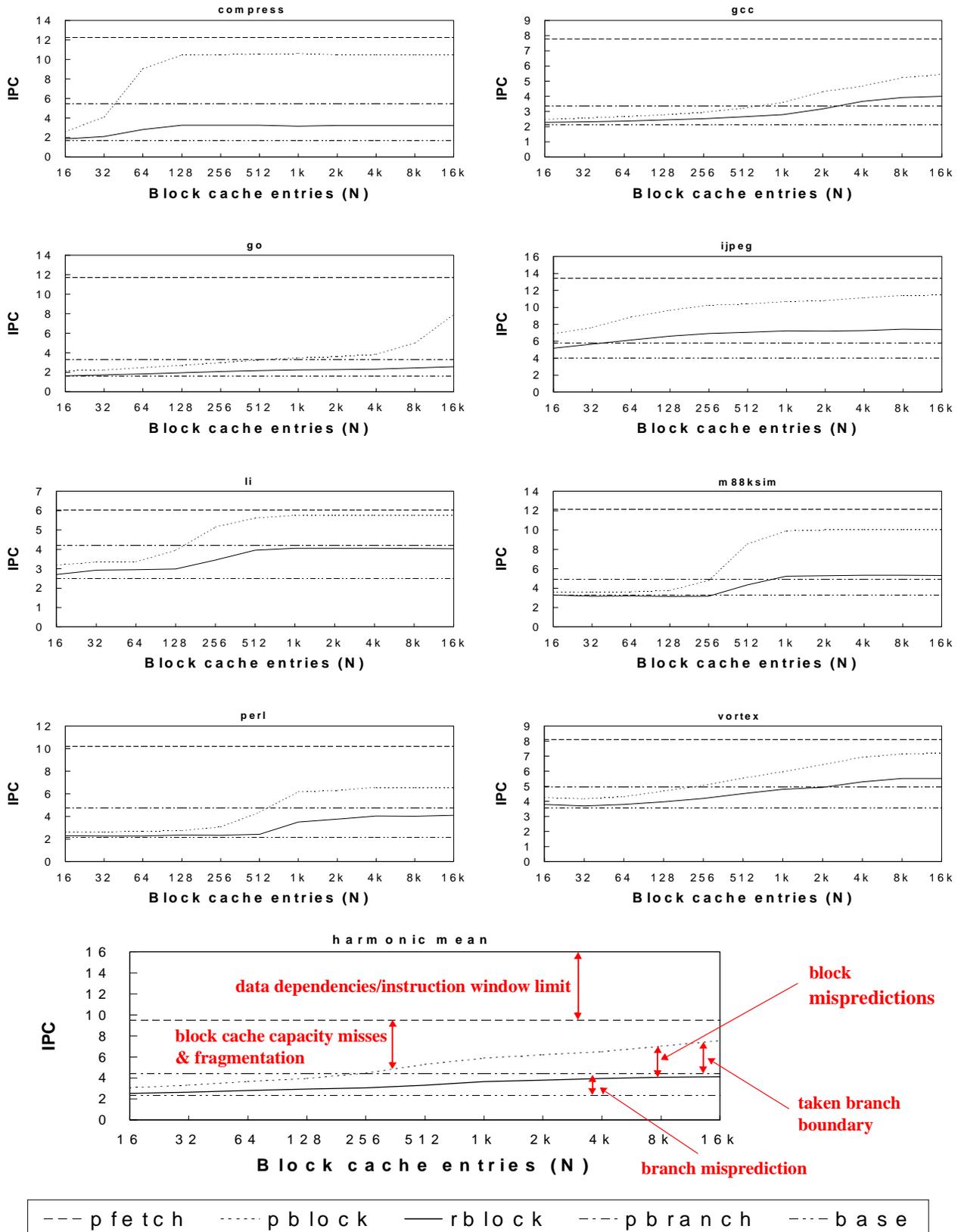
Figure 12 - IPC as a function of block cache size (N=entries), with harmonic mean for all benchmarks.

suite loses 6.4 IPC out of a potential 16 to data dependencies and the limited instruction window. Furthermore, crossing the taken branch boundary demonstrates significant performance improvement over normal branch prediction. At N=16384 perfect next trace prediction (7.6 IPC) approaches the perfect fetch limit for these benchmarks (9.5 IPC). This implies that with better next trace prediction, the block-based trace cache can achieve very high IPC performance.

7   Comparison to Conventional Trace Cache

This section compares the block-based trace cache to the conventional trace cache. For this comparison study a conventional trace cache is implemented using the same methodology and simulation model discussed in Section 4. We attempt to implement the conventional trace cache as described in [5][14]. Wherever possible we keep the parameters for the two trace caches identical. The conventional design uses a direct-mapped trace cache with a trace size of 16 instructions. The trace fetch address is renamed to a trace_id; the rename table for this conventional trace cache is 4-way set-associative. Here perfect next trace prediction is assumed. Perfect partial matching is also implemented. The fill unit minimizes the replication of instructions in the trace cache. It also terminates traces on branch instructions if the branch is near the end of a trace.

The block-based trace cache used in this comparison study, is the design described in Section 3. The block size b=6, the block cache replication w=4, and a 4-way set associative rename table are used. Perfect trace prediction and partial matching are also assumed. Hence the results in Figure 13 represent a limit study of the performance potential of the two trace cache implementations.

Figure 13 compares the IPC of the block-based trace cache with the conventional trace cache as a function of the total number of bytes available for trace storage. The replication of the block cache is accounted for in the total number of bytes. Each entry of the conventional trace cache contains: 16 instructions, the fetch address of the first instruction (needed due to trace renaming), and 8 bits of branch history, totaling 69 bytes. For the block-based design, each block cache line contains 6 instructions and the fetch address, yielding 28 bytes per line. With four replicated copies, the block cache yields 112 bytes per line. The conventional trace cache is simulated with {256, 512, 1024, 2048, 4096, 8192, 16384, 32768} entries. The block-based design is simulated with four copies of the block cache each with {16, 32, 64, 128, 256, 512, 1024, 2048, 4096, 8192, 16384} entries.

The results show the block-based trace cache is more effective at achieving higher IPC with limited trace storage capacity. As expected, extremely large conventional trace caches can exceed the performance of the block-based trace cache. This is due to the greater internal fragmentation of the blocks in the block-based design. Based on Figure 13, in the storage capacity range of 1KB to 1000KB, the conventional trace cache can require over an order of magnitude more storage capacity in order to achieve the same level of IPC.

As the storage capacity increases, the block-based design gains IPC at a faster rate than the conventional design. This is possibly because the block cache warms up faster than the conventional trace cache. Furthermore, since traces are constructed out of blocks, the block-based trace cache can make more efficient use of limited trace cache storage capacity to achieve higher IPC.

The above comparison does not take into account the next trace predictor size for the two designs. Our view is that the domain of next trace prediction is still a wide-open issue and requires a great deal more research. For this paper we simply assume that the next trace predictor table of the conventional design is comparable to the trace table (its counterpart) in the block-based design.

The issue of cycle time impact should also be raised relative to the two designs. Both designs perform next trace prediction in the cycle prior to the fetch cycle. During the fetch cycle, the conventional design accesses the trace cache while the block-based design accesses the replicated block cache and performs final collapse. Since the block cache is a direct mapped structure of most likely 1K-4K entries and does not require tag match, its latency should not be problematic. Given the availability of the steering bits from the trace table (accessed during the trace predict cycle) the latency of the final collapse is simply the propagation of data through the MUX (or a shallow MUX tree). Looking at the data points for the harmonic mean of all the benchmarks in Figure 13, in order to achieve the same IPC (~6.0) level of a block-based design with a block cache of 1K entries, the conventional design will require a trace cache of 32K entries. Granted, the 1K-entry block cache must be replicated four times; however all four copies are accessed in parallel. The much larger trace cache will incur greater latency than the block cache. This difference is likely more than the latency of the final collapse MUX.

It appears that the block-based trace cache is much more efficient than the conventional design in terms of trace storage capacity and hence is able to achieve higher IPC performance for the same storage capacity. Furthermore, to achieve the same IPC level, the number of entries required in the block cache is significantly fewer than that of the conventional design. This will allow the block-based design to scale much better in terms of increasing IPC while minimizing impact on machine cycle time.
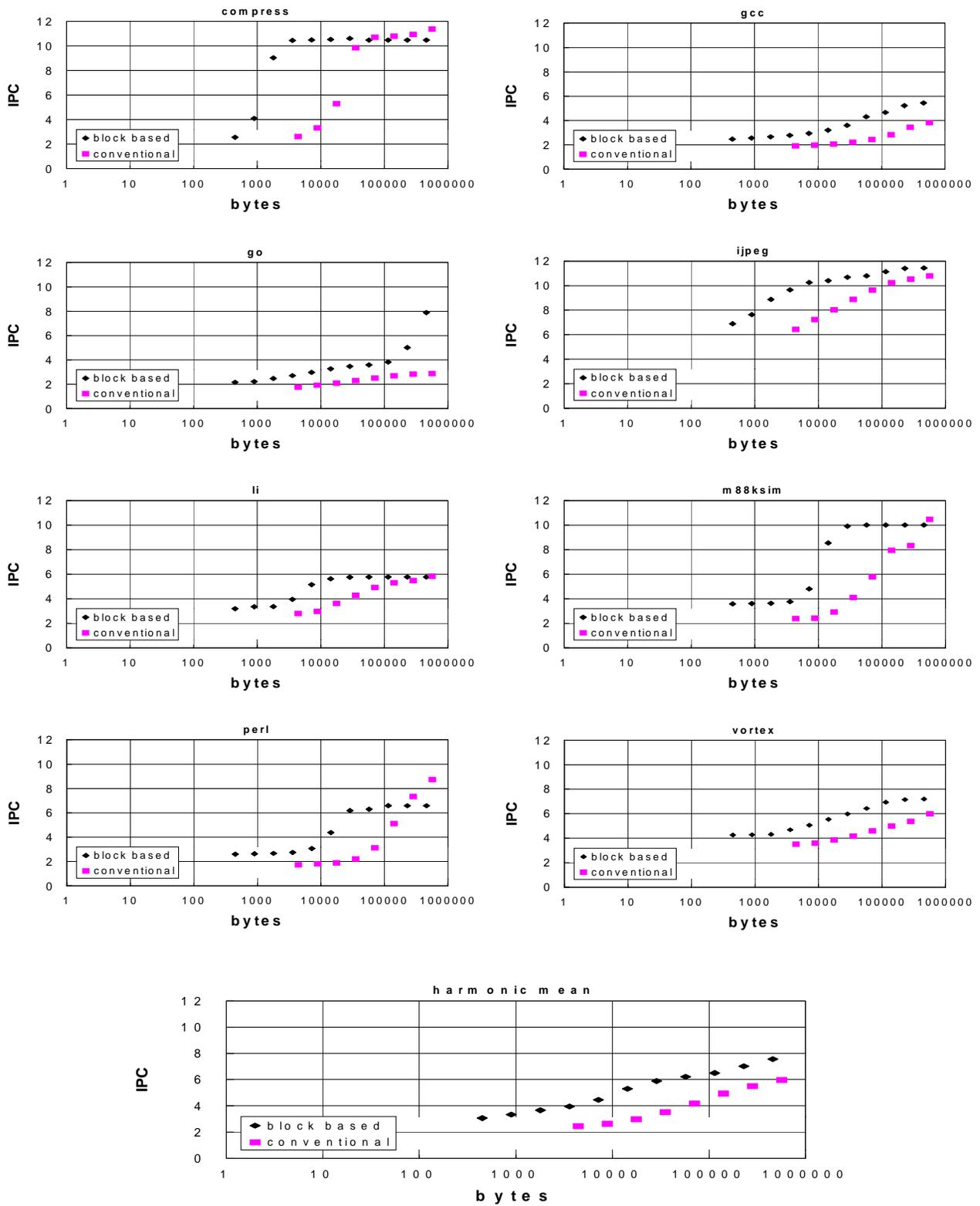
Figure 13 - Achievable IPC per trace storage capacity, with harmonic mean for all benchmarks.

8   Conclusion

This paper presents a new block-based trace cache that can achieve very high instruction fetch bandwidth while yielding an efficient implementation. Comparing to the conventional trace cache, the block-based trace cache is able to achieve much higher IPC when the trace storage capacity is limited. It can also be an order of magnitude more efficient than the conventional design in terms of trace storage while achieving the same IPC. Admittedly, with clever trace selection heuristics the efficiency of the conventional design can be significantly improved. However, the block-based trace cache may be better at increasing IPC while minimizing impact on cycle time.

In this paper, the design space of the block-based trace cache is explored. The achievable performance of the block-based trace cache is compared to that of perfect branch prediction and perfect instruction fetch. For the SPECint95 benchmarks a 16-wide machine with a realistic block-based trace cache design (i.e. a realistic next-trace predictor and a block cache of only 4096 entries) can achieve an average IPC of 3.95. This represents a 68% improvement over a 16-wide baseline design, that is within 12% of perfect branch prediction. With perfect trace prediction, the block-based trace cache can reach an IPC of 7.6, approaching the perfect fetch IPC limit of 9.5.

The results in Figure 12 indicate the need and motivation for developing better trace predictor, and block renaming schemes. Better trace predictors will further improve performance, while better renaming will reduce the block cache storage requirement for instruction blocks.

9   References

[1] B. Black and J. Shen, "Calibration of Microprocessor Performance Models." In COMPUTER, pp. 59-65, May 1998

[2] B. Calder and D. Grunwald, "Next Cache Line and Set Prediction." In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pp. 287-296, June 1995

[3] T. Conte, K. Menezes, P. Mills and B. Patel, "Optimization of Instruction Fetch Mechanisms for High Issue Rates." In *Proceedings of the 22nd International Symposium on Computer Architecture*, pp. 333-343, June 1995

[4] K. Diefendorf, and E. Silha, "The PowerPC User Instruction Set Architecture." In *IEEE Micro*, pp. 30-41, 1994

[5] D. Friendly, S. Patel and Y. Patt, "Alternative Fetch and Issue Policies for the Trace Cache Fetch Mechanism." In *Proceedings of the 30th International Symposium on Microarchitecture*, December 1997

[6] E. Hao, P-Y. Chang, M. Evers and Y. Patt, "Increasing the Instruction Fetch Rate via Block-structured Instruction Set Architectures." In *Proceedings of the 30th International Symposium on Microarchitecture*, December 1997

[7] IBM Microelectronics Division, PowerPC 604 RISC Microprocessor User's Manual, 1994

[8] Q. Jacobson, E. Rotenberg, and J. E. Smith, "Path-based Next Trace Prediction." In *Proceedings of the 30th International Symposium on Microarchitecture*, December 1997

[9] S. Jourdan, T. Hsing, J. Stark, and Y. Patt, "The Effects of Mispredicted-Path Execution on Branch Prediction Structures." In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, October 1996

[10] S. McFarling, "Combining Branch Predictors." Technical Report TN-36, Digital Equipment Corp., June 1993

[11] S. Melvin, and Y. Patt, "Enhancing Instruction Scheduling with a Block-Structured ISA." *International Journal on Parallel Processing*, 23(3):221-243, 1995

[12] R. Nair, "Dynamic Path-based Branch Correlation." In *Proceedings of the 28th International Symposium on Microarchitecture*, December 1995

[13] S-T. Pan, K. So, and J. Rahmeh, "Improving the Accuracy of Dynamic Branch Prediction Using Branch Correlation." In *Proceedings of the 5th International Conference on Architecture Support for Programming Languages and Operating Systems*, pp. 76-84, October 1992

[14] E. Rotenberg, S. Bennett and J. E. Smith, "Trace Cache: A Low Latency Approach to High Bandwidth Instruction Fetching," In *Proceedings of the 29th International Symposium on Microarchitecture*, pp. 24-34, December 1996

[15] E. Rotenberg, Q. Jacobson, Y. Sazeides, and J. Smith, "Trace Processors." In *Proceedings of the 30th International Symposium on Microarchitecture*, December 1997

[16] A. Seznec, S. Jourdan, P. Sainrat, and P. Michaud, "Multiple-Block Ahead Branch Predictors." In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 116-127, October 1996

[17] S. Song, M. Denman, and J. Chang, "The PowerPC 604 RISC Microprocessor." In *IEEE Micro*, pp. 8-17, 1994

[18] S. Wallace and N. Bagherzadeh, "Multiple Branch and Block Prediction." In *Proceedings of the Third International Symposium on High Performance Computer Architecture*, February 1997

[19] T-Y. Yeh, D. Marr, and Y. Patt, "Increasing the Instruction Fetch Rate via Multiple Branch Prediction and a Branch Address Cache." In *Proceedings of the 7th ACM International Conference on Supercomputing*, pp. 67-76, July 1993