Journal of Instruction-Level Parallelism 1 (1999) Submitted 9/98; published 10/99

Performance Limits of Trace Caches

Matt Postiff postiffm@eecs.umich.edu
Gary Tyson tyson@eecs.umich.edu
Trevor Mudge tnm@eecs.umich.edu
Advanced Computer Architecture Laboratory, EECS Department
University of Michigan
Ann Arbor, MI 48109-2122

Abstract

A growing number of studies have explored the use of trace caches as a mechanism to increase instruction fetch bandwidth. The trace cache is a memory structure that stores statically non-contiguous but dynamically adjacent instructions in contiguous memory locations. When coupled with an aggressive trace or multiple branch predictor, it can fetch multiple basic blocks per cycle using a single-ported cache structure. This paper compares trace cache performance to the theoretical limit of a three-block fetch mechanism. The three-block fetch mechanism is modeled by an idealized 3-ported instruction cache with a zero-latency alignment network. Several new metrics are defined to formalize analysis of the trace cache. These include fragmentation, duplication, indexability, and efficiency metrics. We show that performance is more limited by branch mispredictions than ability to fetch multiple blocks per cycle. As branch prediction improves, high duplication and the resulting low efficiency are shown to be among the reasons that the trace cache does not reach its upper bound. Based on the shortcomings of the trace cache shown in this paper, we identify some potential future research areas.

1. Introduction

Instruction supply is a key element in the performance of current superscalar processors. Because of the large number of branch instructions in the typical instruction stream and the small size of basic blocks, fetching through multiple branches per cycle is critical to high performance processors. Traditional instruction cache designs cannot fetch past multiple branches per cycle, and in particular through multiple taken branches per cycle.

The trace cache fetch mechanism is a solution to the problem of fetching past multiple branches in a single cycle. It stores dynamically adjacent instructions in a contiguous memory block and can do so with intervening branch instructions. When it is coupled with a multiple-branch predictor, it can provide a high-bandwidth mechanism to fetch multiple basic blocks per cycle.

This paper presents a study of the limits of trace cache performance and their causes. The goal is not to compare the trace cache against other competing mechanisms or to introduce any new features, but to study where current trace cache configurations can improve.

The contributions of this study are:

- an examination of the limit of trace cache performance based on an idealized 3-block fetch mechanism that is modeled by a 3-ported instruction cache with a perfect instruction alignment network;
- a definition of several metrics to aid in analysis of trace cache performance;

- a study of the sources and extent of trace cache inefficiency;
- identification of branch mispredictions as being a major cause of low trace cache performance; and
- identification of new research opportunities.

The rest of this paper is organized as follows. Section 2 describes previous work and the basic trace cache fetch mechanism. Section 3 introduces several metrics that we use to evaluate the trace cache. Section 4 provides information on our simulation environment, and Section 5 evaluates the limits of trace cache performance. Section 6 extends the results of Section 5 by evaluating trace cache performance in terms of the metrics introduced in Section 3. Section 7 concludes and presents some future directions for trace cache research.

2. Related Work and the Trace Cache Fetch Mechanism

Many caching techniques have been proposed to enhance instruction fetch in superscalar processors. The fill unit assembles multiple instructions from a single basic block for single-cycle issue to a wide-issue processor [2, 4, 5]. The fill unit in [2] is a post-decode cache for CISC instructions which contains partially renamed groups of micro-operations. It was primarily intended as a mechanism to allow a large number of micro-operations to be executed concurrently on an out-of-order processor. In conjunction with the decoded instruction cache, this model reduces both the decoding and dependency checking necessary in the critical execution path. The fill unit of [4] is designed to eliminate complex dependency checking logic in the processor's critical path by assembling instructions into VLIW format and caching the result in a separate shadow cache. The work in [5] is an extension for superscalar processors with complex decoding requirements.

More recently, several fetch mechanisms have been proposed to reduce the impact of branches in the instruction stream. The collapsing buffer [7] relies on multiple accesses to a branch target buffer to produce the addresses needed for fetching multiple basic blocks in a single cycle. The branch address cache [3] requires a highly interleaved instruction cache to support multiple accesses per cycle. The trace cache is an extension of the fill unit and loop/trace buffer [1] that attempts to collect noncontiguous basic blocks from the dynamic instruction stream into a single contiguous cache memory location [6, 8, 9, 10, 11, 12, 20]. The trace cache is compared to several of the previous proposals in [8].

A diagram of the trace cache fetch mechanism is shown in Figure 1. The branch predictor is either a multiple branch predictor [10] or a trace predictor [12]. The fill unit collects basic blocks and builds traces for storage in the trace cache. It merges several basic blocks into a single trace whereas earlier fill units stopped at the first branch instruction. The trace cache is backed up by a conventional instruction cache in the case of a trace miss.

The fetch engine simultaneously presents an address to the trace cache, the conventional instruction cache, and the branch prediction unit. If the trace cache contains a trace starting at the address that also agrees with the branch prediction information, the trace cache signals a *hit* and returns the trace. If the trace cache contains a trace at the address, but the branch prediction information does not completely agree, a *partial hit* is indicated. Instruction cache accesses occur in parallel with the trace cache; this of course, need not be the case if a power-savings is required.

If the trace cache does not contain a trace beginning at the specified address, it signals a *miss*. The instruction cache then supplies the line containing the requested address to the execution engine and the fill unit. The fill unit begins building a new trace.

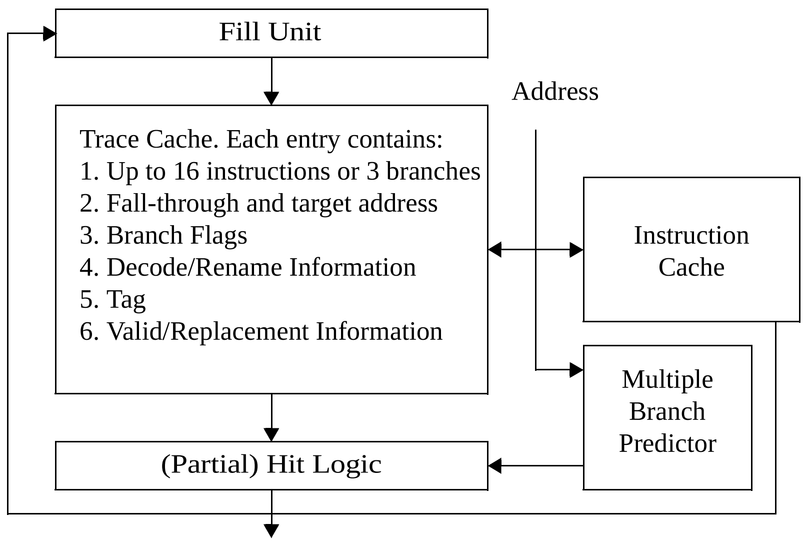


Figure 1: A fetch engine with a trace cache. The fill unit can be filled speculatively, as shown in the diagram, or with traces formed from retired instructions. Lines that cross are connected.

The trace cache fill unit continues to receive instructions until one of the trace termination conditions is met. The trace *termination policy*[1] determines when trace construction is completed. A trace is terminated under any of the following conditions: 16 instructions, 3 basic blocks, or any trap, return, indirect jump or other serializing instruction such as a cache flush.

3. Trace Cache Metrics

In addition to the common metrics of *hit rate* and *IPC*, we use other metrics to help us analyze the trace cache. These are *fragmentation*, instruction *duplication*, *efficiency*, *indexability*, and *retirement rate*. Since performance is the reason for having a trace cache in the first place, IPC must be the metric of choice in determining the best configuration, assuming no degradation in cycle time. Fragmentation, duplication, efficiency and indexability are used to analyze why various configurations perform as they do. The remainder of this section defines and explains these metrics.

3.1. Hit Rate

The hit rate metric measures the effectiveness of the trace cache in providing instructions to the front end of the processor. It is important to note whether the trace cache hit rate is computed using accesses and hits only on the correct execution path or if it is computed without regard to the right or wrong execution path. That is:

$$\text{Correct path hit rate} = \frac{\text{\# hits or partial hits on correct execution path}}{\text{\# accesses on correct execution path}} \qquad \text{(EQ. 1)}$$

$$\text{All path hit rate} = \frac{\text{\# hits or partial hits on right or wrong path}}{\text{\# accesses on right or wrong path}} \qquad \text{(EQ. 2)}$$

1. Also called *trace selection* or *trace finalization policy.*

The all-path hit rate makes no distinction between correct and incorrect execution path. There can be a significant difference between these two metrics because of branch prediction accuracy and processor pipeline width and depth. While we recognize this discrepancy in the two ways of measuring hit rate, we will always show the correct-path hit rate. It is generally higher than the all-path hit rate.

3.2. Fragmentation

Like hit rate, fragmentation indicates how efficiently the trace cache stores instructions. Fragmentation is a measure of storage utilization which describes the portion of the trace cache that is unused because of traces shorter than 16 instructions. It is essentially wasted storage. Fragmentation is related directly to the trace selection policy. More conservative trace selection results in shorter traces, and thus higher fragmentation. Rotenberg [12] showed that average trace length was reduced by about 20 percent when backward branch and call instructions were added to the trace termination conditions.

During a particular clock cycle, fragmentation is the ratio of empty instruction slots to total instruction slots, counting empty trace lines. Average fragmentation is computed by summing the fragmentation values for each cycle and dividing by the number of cycles. A higher value for fragmentation indicates a less efficient trace cache; a conventional cache has no fragmentation.

We defined fragmentation to include empty lines because sometimes trace cache lines cannot be used; that is, there are no basic blocks or fragments that *start at* address X in the benchmark, so location X in the trace cache is forced to be empty. Also, this definition of fragmentation allows a more intuitive definition of the efficiency metric (defined below).

3.3. Duplication

Another measure of instruction fetch capability is duplication. Duplication is a measure of how efficiently the “un-fragmented” storage in the trace cache is used. Duplication is a consequence of the method of indexing the trace cache and is really an intended side effect. In a conventional instruction cache, a particular instruction can only appear once because only the instruction address is used to index the cache. In a trace cache, the instruction address along with branch prediction information is used to identify a trace, so a given block may begin a trace and also appear as an interior member of many traces in the trace cache.

Code duplication in the trace cache occurs because a program revisits a section of code. It may be that conditional branch instructions in the code take different directions each time they are executed, as can be the case with if-then-else constructs. In such cases, duplication is due to the multiple inclusion of fork and join points in the control flow graph. This is illustrated in Figure 2(A).

Duplication may also occur because of a loop whose length is not an integer multiple of the maximum trace cache line size. This case is illustrated in Figure 2(B). If N is the number of instructions that can fit in the trace cache line, a loop of L instructions will result in N/GCD(L, N) trace lines being stored. In the case where a loop has one more instruction than the trace cache line can hold (i.e. GCD(L, N) = 1), each instruction will be stored N times, and the trace cache will be swamped with N similar (shifted) trace lines. This pathological case only occurs if the loop is executed N times dynamically and could be avoided entirely if the compiler tailored the loop to the particular trace cache configuration.

We use the formula

$$\text{duplication} = \frac{(\text{total instructions} - \text{unique instructions})}{\text{total instructions}} \quad \text{(EQ. 3)}$$

to capture the duplication in the trace cache. As with fragmentation, the value we report is an average across the benchmark cycle count. This is analogous to the dynamic redundancy factor reported in [13]. A higher duplication indicates lower utilization of the trace cache.

A conventional cache has no duplication, though a cache hierarchy may exhibit duplication due to cache inclusion. Duplication in the trace cache is more serious than inclusion-duplication because the duplicates appear in a single memory structure instead of across several levels of memory structures. Since the memory structure must be larger than if it held no duplicates, the access time of the structure is increased.

3.4. Efficiency

Fragmentation and duplication are important metrics because they indicate how efficiently a trace cache configuration can provide instruction bandwidth to the processor. We define

$$\text{efficiency} = (1 - \text{fragmentation}) \times (1 - \text{duplication}) \quad \text{(EQ. 4)}$$

to be the single number that wraps this information together. Efficiency represents the fraction of the whole trace cache that is actually storing unique instructions as opposed to simply (1 - duplication), which measures the fraction of the utilized trace cache that stores duplicate instructions. Combining equation (4) with (3) and the definition of fragmentation, we arrive at a somewhat more intuitive definition of efficiency:

$$\text{efficiency} = \frac{\text{unique instructions}}{\text{total instruction slots}} \quad \text{(EQ. 5)}$$

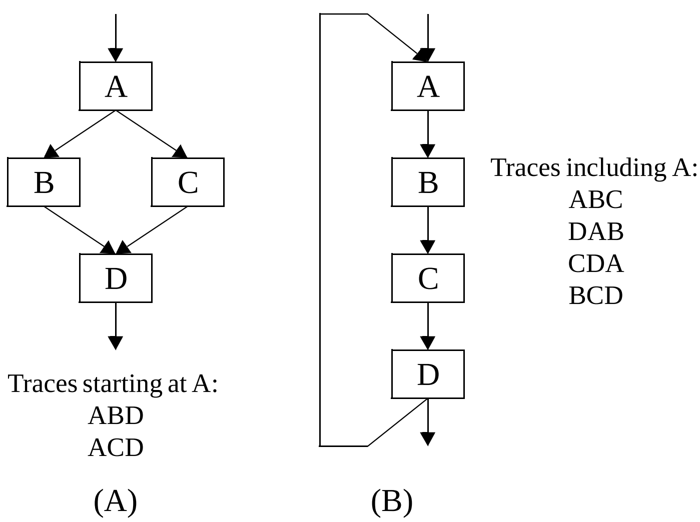


Figure 2: Causes of duplication in the trace cache. (A) illustrates duplication due to conditional branches, while (B) shows a pathological case of shift redundancy (duplication) due to a backward loop branch.

A1	A2	A3	A4	A5	B1	B2	B3	B4	B5						
E1	E2	F1	F2	F3	F4	G1	G2	G3	G4	G5	G6				
C1	C2	C3	B1	B2	B3	B4	B5								
D1	D2	D3	D4	D5	E1	E2	G1	G2	G3	G4	G5	G6			

Figure 3: A 4-entry trace cache with fragmentation = (6+4+8+3) / (4*16) = 33% The duplication in this example is (43 - 30) / (10+12+8+13) = 30% since there are 43 total instructions and 30 unique instructions in the trace cache. The efficiency is (1 - 0.33)*(1-0.30) = 30/(4*16) = 47%.

We include empty trace lines in the fragmentation metric so that storage efficiency is measured across the entire trace cache structure. Figure 3 shows an example.

For a conventional instruction cache, the duplication is zero and the fragmentation in the steady state is zero. Therefore, the efficiency of the instruction cache is 1. There is no internal fragmentation of conventional cache lines, but portions of some cache lines could be dynamically unused. We did not track the usage frequency of instructions within the cache and so will not consider this special case any further.

3.5. Indexability

Indexability provides information about the presence of traces even if they do not start a trace line. Since trace lookup is anchored at the address that starts the trace line, a miss may occur because it is not possible to directly access interior blocks. In this case, the trace cache performs worse than an idealized three-ported instruction cache with perfect alignment mechanism.

Specifically, we define indexability to be a miss rate that indicates how often a trace starting address is simply not in the trace cache at all, even at an interior block. When an address is requested from the trace cache, we not only use the traditional indexing scheme (chop the offset and tag bits) but we also examine every set in the trace cache to determine if some portion of a trace contains that address. If no such partial trace can be found, the indexability miss count is incremented. The indexability value is lower than the correct-path miss rate since it examines all the traces in the cache. A more sophisticated indexing mechanism that can access some internal blocks of traces could improve correct-path hit rates.

For a conventional instruction cache, the miss rate is equal to the indexability because a given instruction can only reside at one directly-accessible location in the instruction cache.

We present indexability as a limit. It is not practically implementable since it requires looking at all trace cache lines simultaneously and finding the longest match. It will show how important proper trace-cache indexing is to trace cache performance.

3.6. Trading off Fragmentation, Duplication, and IPC

There is a fundamental trade-off to be made between the performance metrics introduced above. The constrained trace selection policy mentioned in Section 3.2 will serve as a good example. It was noted in [12] that the average trace length is reduced by conservative trace selection, that is, adding trace termination conditions. While shorter traces mean that fragmentation will increase, our simulation results show that duplication decreases correspondingly. This is to be expected because the termination of traces on backward branches eliminates duplication due to loops. Thus fragmentation increases but duplication decreases, resulting in little change in overall efficiency. Furthermore, the trace cache hit rate increased under constrained trace selection but we observed in our simulations that in some cases overall performance actually decreased because of the decrease in fetch bandwidth due to the shorter traces. These metrics will be discussed in more detail in Section 6.

3.7. Retirement Rate

The previous metrics evaluate how effectively the trace cache structure stores traces and how it provides instructions to the front end of a processor. The goal of the retirement rate metric is to evaluate the effects of employing a trace cache on other processor resources. The increased fetch bandwidth made possible by incorporating a trace cache will require additional resources at later stages of the pipeline. IPC measures do not show the pipeline resource requirements of those instructions, which may be squashed prior to retirement (i.e. the wrong path instructions). Retirement rate[2] is the ratio of the number of instruction fetched into the pipeline to the number retired:

$$\text{retirement rate} = \frac{\text{total instructions retired}}{\text{total instructions fetched}} \times 100 \qquad \text{(EQ. 6)}$$

Retirement rate is one measure of the amount of pipeline resources wasted due to wrong-path instructions. Retirement rate is a function of branch prediction accuracy, pipeline depth (or branch resolution time) and issue width. Retirement rate will be considered in Section 5.2.

4. Simulation Environment

Simulation results were obtained with a modified version of the sim-outorder simulator from the SimpleScalar tools [14]. For all experiments, the SPEC95 integer benchmarks were run on the input sets listed in Table 1. The benchmark binaries provided in the SimpleScalar distribution are used in these experiments. The programs were compiled with GNU GCC 2.6.2, GNU GAS 2.5, and GNU GLD 2.5 with maximum optimization (-O3). Loop unrolling was enabled (-funroll-loops). The simulator parameters common across all configurations simulated are shown in Table 2.

Benchmark	Input Set	Insts (M)	Instructions per Branch
compress	30000 q 2131	121	5.0
gcc	regclass.i	124	5.5
go	9 9 null.in	133	6.6
ijpeg	specmun.ppm	124	11.1
li	boyer.lsp	174	4.4
m88ksim	dcrand.train.lit	48	4.4
perl	jumble.pl < jumble.in	74	5.1
vortex	vortex.in	154	6.3

Table 1: Benchmarks and data sets used. All benchmarks were simulated to completion (some were scaled down from training input).

2. We will call it a “rate” even though it is a ratio, in keeping with common usage.

Parameter	Value	Budget
L1 instruction cache	256 sets, 64-byte line, 4-way associative, 1-cycle access/ throughput/blocking (actually 128-byte line of SS insts)	64 KB
L1 data cache	512 sets, 32-byte line, 4-way associative, 1-cycle access/ throughput/blocking	64 KB
L2 unified cache	2048 sets, 128-byte line, 4-way associative, 6-cycle access	1 MB
Memory Latency	50 cycles for the first 8 bytes, 1 cycle each 8 bytes thereafter	
Branch Predictor	16-bit gshare accessed three times per cycle; perfect RAS	
Trace Cache	2-way associative, 1-cycle hit latency, line size fixed at a maximum of 16 instructions, partial hits and path associativity; 1-cycle fill unit delay	
Fetch queue	128 entries	
Width	16 instructions per cycle	
Function Units	16 symmetric (each can do all instructions)	
RUU/LSQ sizes	512/256	

Table 2: Configuration parameters common across all simulations, unless otherwise noted.

To stress the fetch engine, the processor’s execution engine is very aggressive. There are 16 of each of the five types of function units (integer ALU, integer multiplier, memory port, floating point ALU and floating point multiplier). The instruction cache simulated was 128KB, but SimpleScalar instructions are 64 bits long, so this is effectively a 64KB cache of conventional 32-bit instructions. We will quote all L1 instruction cache and trace cache sizes as if SimpleScalar instructions were 32-bits.

The trace cache is simulated with 64 and 1024 sets and is 2-way associative in both cases. With 64 sets and 16 instructions, there are 64*16*4*2 = 8KB of instruction storage. For the 1024-set trace cache, there is 1024*16*4*2 = 128KB of storage. Traces are finalized on instruction boundaries when any of the following conditions are met: 1) 16 instructions; 2) three branches; or 3) trap or indirect jump instruction. Branch prediction information is used as part of the tag match instead of as part of the index into the trace cache to determine the longest matching trace for path associativity and partial hits. This assumes that branch prediction lookup and trace cache lookup cannot happen in series in a single cycle, which would be necessary if the branch predictions were used as part of the trace cache index and the fetch mechanism were not pipelined.

4.1. The Branch Predictor

The branch predictor used for all non-perfect simulations is a 16-bit gshare predictor where the shift register value is XORed with the lower PC bits and indexes a 2^{16}-entry table of two-bit counters. When multiple branches are being predicted per cycle, it is accessed the required number of times in series, as if the hardware could be accessed that many times in one cycle.

The branch predictor uses speculative history information. All wrong-path history bits are squashed once a mis-prediction has been identified. This is done because neither speculative update nor non-speculative update alone provide the best performance [16]. The reason for this is that trace cache processing enables considerable speculation, resulting in non-speculative history that is too old, or speculative history that contains too many history bits from the wrong path. The solution is to maintain speculative history which is squashed when a mis-prediction occurs. This method provides the same prediction accuracy regardless of the amount of speculation performed by the processor, and is at least as accurate as speculative or non-speculative update alone. Of course, this is not practical because it requires the saturating counters to be re-adjusted to their state prior to the misprediction. It is a simulation ideal.

5. Limits of Trace Cache Performance

The trace cache strives for the performance of a fetch mechanism that can fetch three basic blocks per cycle without a multi-ported instruction cache. Most previous studies have compared trace cache performance to the performance of sequential fetching mechanisms, i.e. a fetch engine that can fetch up to one branch (SEQ.1) or up to 3 branches where the first two are predicted not taken (SEQ.3) [8]. While this highlights the performance improvement of fetching non-contiguous blocks over fetching only sequential blocks, it is not a true upper bound to performance. This study compares trace cache performance to the theoretical limit of a three-block fetch mechanism equivalent to an idealized three-ported instruction cache with a perfect alignment network. This cache can provide three non-contiguous blocks each cycle and merge them for placement into a fetch buffer (a block is defined in the same way as for the trace cache). The latency of the merge operation is not counted in our simulations. We call it NONSEQ.3 to conform to previous terminology. A similar NONSEQ.3 baseline is used in [9].

A lower bound on trace cache performance is a single-block fetch mechanism equivalent to a conventional instruction cache. It can fetch up to the first branch or up to some maximum number of instructions (16 in these simulations). The next two subsections show the results of the limit simulations for gshare and perfect branch prediction.

5.1. Gshare vs. Perfect Branch Prediction - 1 and 3 block fetch

Table 3 shows the performance of 1- and 3-block fetch engines. The left half of the table presents data for configurations with a gshare branch predictor as described in Section 4.1. The right half shows speedup when using perfect branch prediction. The first two columns in each portion of the table show the IPC for the conventional instruction cache and the 3-ported instruction cache, respectively. The third columns show the potential speedup of employing a trace cache. We also include the branch prediction accuracy for the gshare configurations.

Because the branch predictor uses speculative history which is corrected after a wrong path is encountered, the 3-block fetch engine will always perform better than the 1-block mechanism. The benchmarks with very good branch prediction (li, m88ksim, and vortex) achieve significant performance improvements in the 3-block case—50% or more—indicating that the trace cache can provide significant performance benefit for these programs. The other configurations (cc1, compress95, go, and ijpeg) suffer from lower prediction accuracy and cannot take advantage of the two extra blocks per cycle because there are many wrong-path instructions that must be squashed. As the 3-block case is the limit of performance for the trace cache modeled, the trace cache can only provide a performance benefit of around 20% for these programs. These results are more optimistic than previously-published data would suggest [8], though no previous study has shown the true 3-block fetch limit.

As can be seen from the data on the right side of Table 3, there is potential for significant improvement in trace cache performance when branch prediction is perfect—often 60% or more improvement in IPC.

SPEC95 Program	Fetch 1 Block IPC gshare	Fetch 3 Block IPC gshare	Percent Increase	Branch Predict %	Fetch 1 Block IPC perfect BP	Fetch 3 Block IPC perfect BP	Percent Increase
cc1	2.64	3.35	27%	93.27%	4.80	9.59	100%
compress	3.22	3.74	16%	94.75%	5.31	8.80	66%
go	2.50	2.81	13%	86.12%	5.75	7.94	38%
ijpeg	5.59	7.01	25%	93.93%	7.59	10.74	42%
li	3.47	5.48	58%	97.01%	4.40	10.15	131%
m88ksim	4.17	10.41	150%	99.82%	4.25	11.03	159%
perl	3.69	4.63	26%	98.55%	5.08	8.64	70%
vortex	4.56	6.75	48%	98.89%	5.80	9.31	61%

Table 3: The performance of fetching 1 block or 3 blocks under gshare and perfect branch prediction.

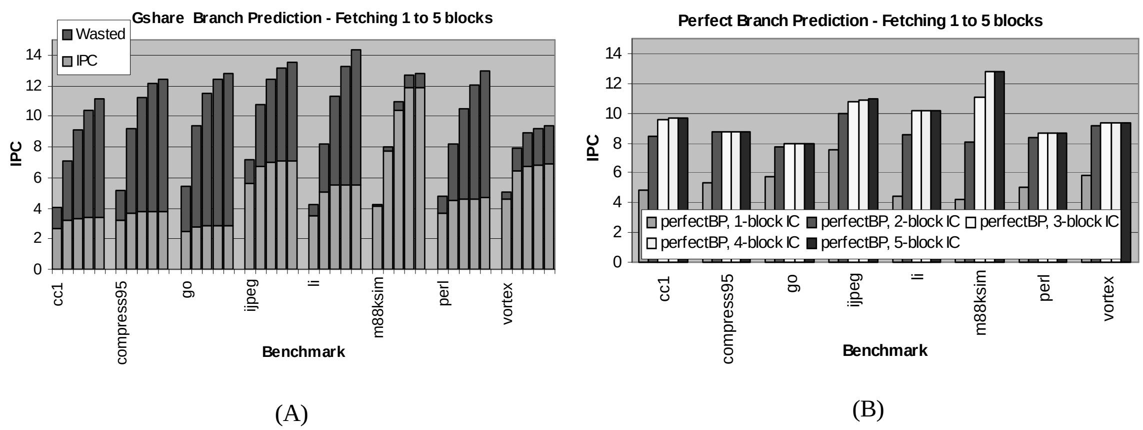


Figure 4: Performance of n-block fetch mechanisms under (A) gshare and (B) perfect branch prediction. The perfect predictor shows the performance potential of a multi-block fetch mechanism. The portion of the bar labeled 'Wasted' indicates instructions that were fetched but never retired.

In summary, Table 3 shows that when branch prediction accuracy is high, the performance potential of fetching multiple blocks per cycle is significant, i.e. 50% or more. When branch prediction accuracy is low, fetching multiple blocks only helps performance up to about 20%. Performance is limited more by branch prediction than the inability to fetch multiple blocks per cycle. Nevertheless, as branch prediction improves, a mechanism like the trace cache that can fetch multiple blocks per cycle becomes more beneficial.

5.2. Gshare vs. Perfect Branch Prediction - 1 to 5 block fetch

The graphs in Figure 4 show a superset of the data in Table 3. To highlight the resource allocation required to support the greater number of instructions fetched by more aggressive configurations, Figure 4(A) shows the performance of configurations which fetch one, two, three, four, and five blocks per cycle. The machine is otherwise configured as shown in Table 2. The dark upper portion of the bars indicate instructions that are fetched but later squashed because of branch mis-predictions. Figure 4(B) is similar but uses perfect branch prediction, so no instructions are squashed.

SPEC95 Program	Fetch 1 Block gshare Retirement Rate	Fetch 2 Block gshare Retirement Rate		Fetch 3 Block gshare Retirement Rate		Fetch 4 Block gshare Retirement Rate		Fetch 5 Block gshare Retirement Rate	
	Overall	Overall,	Extra	Overall,	Extra	Overall,	Extra	Overall,	Extra
cc1	65%	45%	19%	37%	7%	33%	4%	31%	3%
compress	62%	40%	13%	33%	1%	31%	1%	30%	0%
go	46%	29%	6%	24%	3%	23%	2%	22%	2%
ijpeg	78%	62%	32%	57%	17%	54%	5%	52%	2%
li	82%	62%	41%	48%	12%	42%	3%	39%	1%
m88ksim	98%	97%	95%	95%	91%	94%	85%	93%	16%
perl	78%	56%	25%	43%	4%	38%	0%	36%	4%
vortex	90%	82%	66%	76%	31%	74%	22%	73%	20%

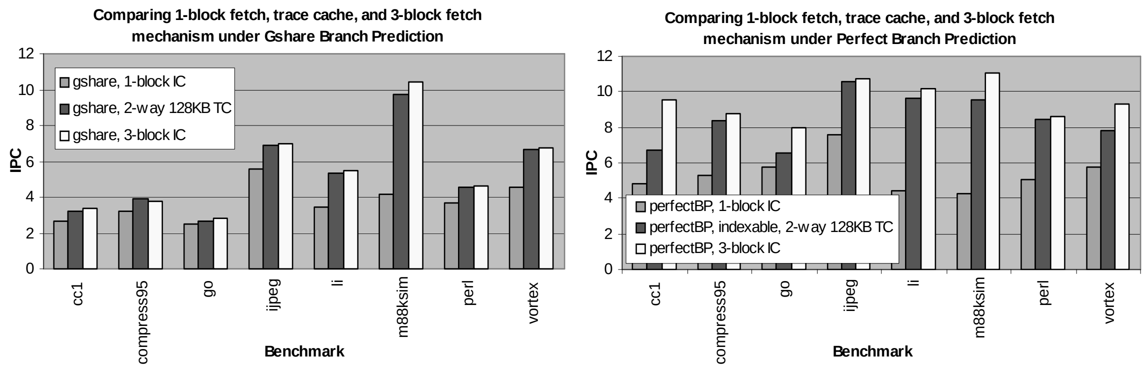
Table 4: Retirement rates for 1- to 5-block fetch configurations in Figure 4(A). The overall retirement rate is computed as defined in Section 3. The extra retirement rate shows the retirement rate of the extra instructions fetched by that configuration compared to the previous column.

Table 4 examines the data in Figure 4(A) by showing the retirement rates for each of the configurations. As the machine fetches past more branches, the retirement rate decreases monotonically. The retirement rate decreases rapidly after the first and second branches, then less so after the third and fourth branches; the retirement rate of cc1, for example, falls from 65% to 31%. These results suggest that while the capability of the front end of the pipeline has dramatically increased with the additional blocks fetched, the resource utilization at the backend of the machine is very low because of the low prediction accuracy.

Another interesting metric is the retirement rate of the extra instructions brought in by the second, third, fourth, and fifth blocks. The second number in the columns of Table 4 shows the value of this special retirement rate. For example, the overall retirement rate of cc1 is 45% for the 2-block fetch configuration. Only 19% of the additional instructions brought in by the second block are actually retired. Only vortex and m88ksim exhibit extra-instruction-retirement-rates of more than 60% from the first to the second block because their branch prediction accuracy is so high. The other benchmarks generally exhibit rates of 25% or less. As we proceed to three blocks we see the retirement rate of the extra instructions fall below 1 in 10 for most programs. This means that only 1/10th of the pipeline resources are being constructively utilized for extra instructions. Other work has taken advantage of this to reduce power consumption by not fetching these instructions [19]. Additional instructions that could be brought in by a trace cache are simply not useful.

The performance of perfect branch prediction in Figure 4(B) saturates after 3 branches per cycle primarily because of data-dependence limitations in the backend. Function unit contention is not a significant cause of this leveling off of performance because the average IPC never goes above 11. As expected, misprediction recovery time dominates delays due to data dependencies in the gshare configurations.

The trace cache, which is limited above by the 3 block fetch case, suffers from the same branch prediction limitation. It can provide high peak bandwidth, but the overall processor performance is most limited by the branch prediction.



(A) (B)

Figure 5: Trace cache performance with perfect branch prediction and perfect indexability.

5.3. Trace Cache vs. Limit Cases

We have already noted that trace cache performance must fall between the 1- and 3-block fetch cases. Figure 5 demonstrates this for both gshare and perfect branch prediction, with the exception of the compress. Compress exhibits slightly pathological behavior and performs better than the upper bound. This is because the total instruction capacity of the trace cache plus instruction cache is larger than the instruction capacity in the 3-block configuration, which just has the instruction cache. The result is that the trace cache configuration exhibits fewer capacity misses to the second level cache and thus suffers less from memory latency. We also found that go would perform pathologically if the branch prediction was worse (82.70% instead of the present 86.12%). This is because go has a large working set of instruction paths that exceed the capacity of even a large (128KB) trace cache. A 2MB trace cache was simulated and found to eliminate this problem. Still, the performance of go is limited more by branch prediction than anything else, as the difference between 1- and 3-block fetch is only 13%.

Figure 5(A) further shows that the trace cache can come close to the upper bound when branch prediction is not perfect, demonstrating that the trace cache is, for the most part, achieving its goal of 3-block performance with a single-ported memory.

In the case of perfect prediction, Figure 5(B), we see that the 128KB trace cache generally falls short of ideal by 20% or more. This is significant because as branch prediction improves, it appears that the trace cache is falling farther below its upper bound. Thus the trace cache cannot take full advantage of future improvements in branch prediction. We also simulated a 2MB trace cache (not shown) and found that for the large benchmarks like gcc, go, and vortex, the trace cache still fell short of the limit case by more than 10%. Apparently imperfect branch prediction hides some other deficiencies in the trace cache. Reasons for this are examined in the next section.

6. Efficiency and Indexability Results

The results presented in Section 5 show that trace cache performance does not achieve the theoretical limit of 3 block/cycle fetch with perfect branch prediction. This section uses the previously defined metrics to analyze why this might be the case.

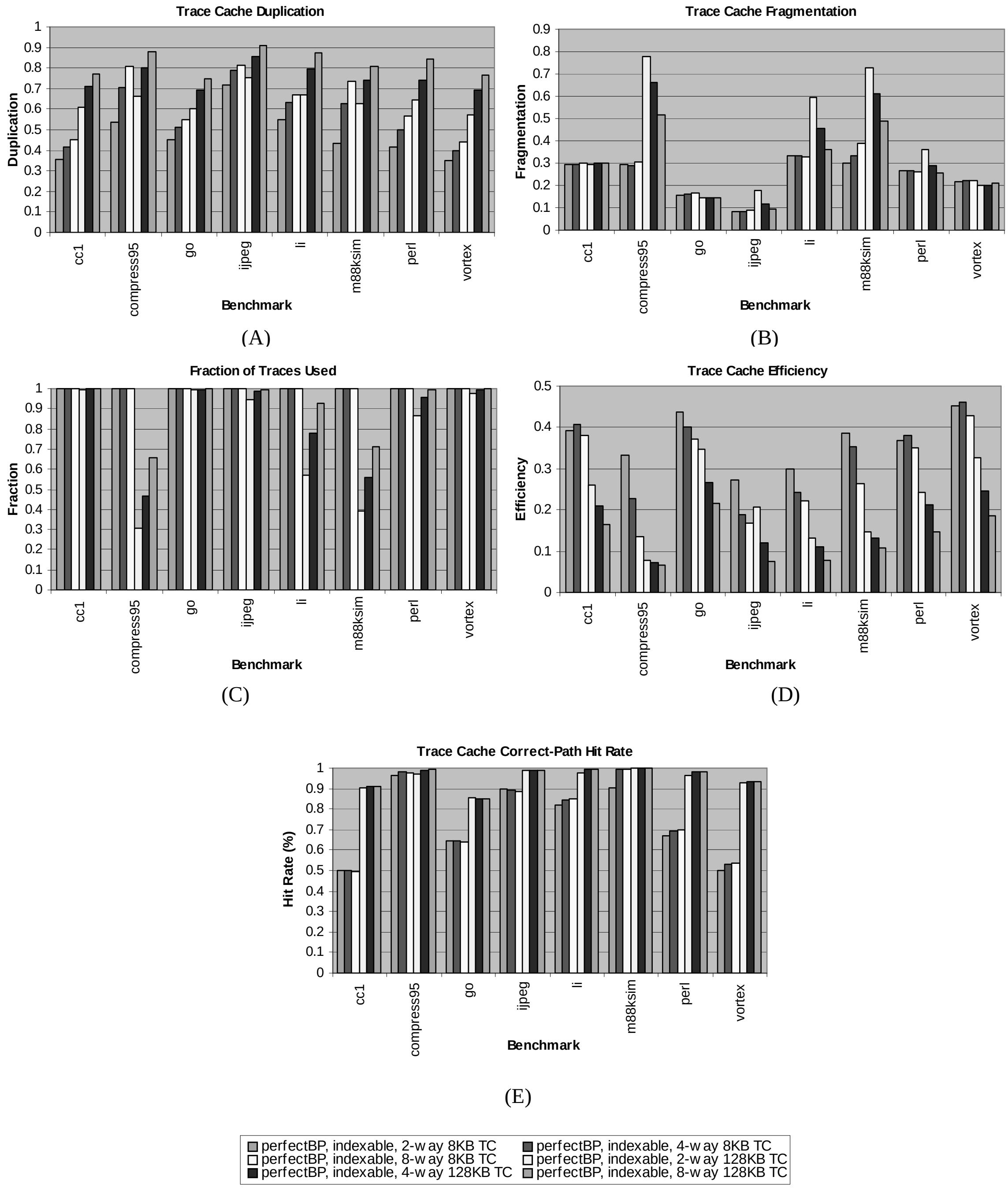


Figure 6: Trace cache duplication, fragmentation, full traces, efficiency, and hit rate for several 8KB and 128KB configurations with perfect indexability.

The graphs in Figure 6 show the duplication, fragmentation and efficiency of a trace cache as associativity and size are varied. Figure 6(A) shows that duplication of instructions in the trace cache grows from 30%-50% up to 75%-90% as associativity and size are increased. Increasing the size of the trace cache dramatically increases the duplication. For very large trace caches, an instruction may reside in 10 or more locations.

Similarly, Figure 6(B) shows that fragmentation generally increases for the larger trace caches. For the smaller benchmarks such as compress, li, and m88ksim, this is primarily due to many trace cache slots which are unused throughout the benchmark run—there are simply not enough unique trace starting addresses to utilize the trace cache. This is particularly evident for compress, which uses only a couple of trace slots very heavily throughout the benchmark run and leaves many trace slots unused. For go and the other larger programs, which have a large number of paths, we see that the fragmentation does not generally increase as the trace cache increases in size from 8 to 128KB.

For compress, li, and m88ksim, fragmentation is generally improved as associativity increases from 2 to 4. This is to be expected because any unused traces in the 2-way associative cache can be utilized in the 4-way cache for traces which are competing for the more heavily used trace starting addresses. In other words, the additional flexibility afforded by extra associativity allows the trace cache to use some locations that would otherwise be blank. This is shown in Figure 6(C). The larger benchmarks generally use up all the traces regardless of the associativity and size. In summary, the trace cache loses 20% to 30% of its capacity due to empty and short traces.

The overall efficiency is rarely above 40% and for the 128KB configuration is generally between 20% and 35% as shown in Figure 6(D). Certainly the trace cache is designed to trade-off space efficiency for increased fetch bandwidth, but such low storage efficiency is remarkable. The low efficiency is primarily caused by code duplication. When associativity is increased, the efficiency gain possible because of decreased fragmentation is outweighed by increased duplication.

The overall performance of the trace cache is determined by the hit rate and the length of the trace lines referenced. Experiments in [12] indicate that trace cache hit rates range from 60%-90%, and our experiments confirm this trend. However, previous experiments required that the address in the fetch request must be located in the first entry in some trace line. In this study we also examine a trace structure in which this restriction is removed — the perfectly indexable trace cache. Figure 6(E) shows the hit rate when the complete trace cache is searched for the fetch PC (traces can start anywhere instead of being anchored to the start of the trace storage). When indexing the trace cache is expanded to any instruction in a trace line, the hit rate increases to 90%-99% for most applications. Unfortunately there is a reduction in the average length of trace fetched from the trace line because many paths start from some point in the middle of the trace line. However, this increase in hit rate demonstrates that improved indexing methods can significantly increase the trace cache hit rate. This suggests that current trace cache implementations do not miss because new paths are identified. Instead they miss because cache line allocation policies are naive.

7. Conclusion

In this paper, we have introduced several new metrics for evaluating the performance and efficiency of trace cache implementations. By utilizing these metrics, the functioning of a trace cache can be better understood, enabling us to identify strengths and weaknesses of this approach to increasing parallelism.

The performance of the trace cache configurations studied in this paper suffer primarily from low branch prediction accuracy. Less than 1 in 10 of the additional instructions fetched from a trace cache are retired. The trace cache can provide high bandwidth instruction fetch but because of this a large number of branches are in flight, reducing the efficiency of fetching useful instructions. It is not surprising that a high bandwidth fetch mechanism would stress the branch predictor, but a retirement ratio of 1 in 10 for those extra instructions was lower than we expected.

When perfect branch prediction is simulated, the trace cache is still not able to fetch instructions at the rate of the 3-block limit case. This drop in issue rate is caused by missing in the trace cache and by hitting trace lines that contain only part of the 3-block path. Further study using the metrics defined herein reveals deficiencies in the way the trace cache stores traces. Low trace cache efficiency and poor indexability are the primary reasons for this shortcoming.

This study has identified several potential areas where trace cache performance can be improved. These are:

- Branch prediction. Improvements in branch prediction accuracy will impact overall performance of a trace cache the most. Trace based predictors specifically tuned for trace cache design may be able to identify new correlations currently unexploited by conventional predictors [18]. However, prediction accuracy will still likely be the most limiting factor in overall performance of a trace cache.
- Duplication. Since duplication in current configurations is 50% or more, conservative trace termination policies could be used to reduce duplication due to loops and fork-join points. Duplication can also be reduced by adopting a more restrictive placement algorithm in allocating trace lines. In our studies, all trace paths are placed into the cache each time they are executed. By using a selection criteria to allocate trace cache entries to the most useful paths (e.g. those that have high ILP possibility due to few data dependencies and good branch prediction accuracy) duplication can be reduced without negatively effecting performance.
- Fragmentation. This is caused mostly by a large number of short and empty trace lines. A certain subset of these lines can be left in the instruction cache without any performance penalty, and can thus increase the efficiency of the trace cache. Never-used trace slots are also a problem that should be addressed. Fragmentation can also be effectively reduced by many compiler transformations that increase average basic block size [17]. In this study we did not study the effects of optimizations such as superblock [22] or trace scheduling [23] and predication [24]. These optimizations should reduce fragmentation in the trace cache. It would be interesting to see if trace scheduling, predication, and a single-ported instruction cache with a long line size is able to provide the same performance benefit as a trace cache since branches are eliminated and common paths can be scheduled contiguously. The software trace cache is the only work we are aware of which addresses some of these issues [21].
- Indexability. Measurements showed that often the requested block is in the trace cache but it cannot be reached since it is at an interior block.

The gshare branch predictor used in this study, while overly optimistic, is still better than the branch prediction that will be available in commercial products on real programs. This and the issues defined above lead us to question the utility of the trace cache as a mechanism to fetch multiple blocks per cycle, at least in the next couple of generations of microprocessors. A trace-cache-like structure whose "traces" are a single block, however, would be useful as a post-decode cache to save time and power in instruction decoding and renaming.

In this paper we have quantified the performance gains possible due to wider instruction issue. We see from the limit study utilizing perfect branch prediction and a perfect 3-block issue mechanism that there is still more parallelism available; however, current trace cache designs suffer from wasted resource utilization and poor branch prediction (relative to the requirements of the wider issue). The fundamental problem is that the trace cache heavily emphasizes the already important requirement for good branch prediction because it requires multiple predictions per cycle. The trace cache does eliminate the need for a multi-ported cache structure but may instead require a multi-ported branch prediction structure or a single-ported structure with a complex selection mechanism (see [15], for example). Instead of trying to fetch past multiple branches, we think an interesting avenue of related research would be to de-emphasize branch prediction and find other means to increase performance.

8. Acknowledgments

This work was supported by DARPA contracts DAA H04-94-G-0327 and DABT63-97-C-0047. The simulation facility was provided through an Intel Technology for Education 2000 grant. We also acknowledge the helpful comments of the JILP reviewers.

References

[1] Chih-Po Wen. Improving Instruction Supply Efficiency in Superscalar Architectures Using Instruction Trace Buffers. *Proc. Symp. Applied Computing*, Volume 1, pp. 28-36, March 1992.

[2] S. Melvin, M. Shebanow, and Y. Patt. Hardware support for large atomic units in dynamically scheduled machines. *21st Intl. Symp. on Microarchitecture*, pp. 60-63, Dec. 1988.

[3] Tse-Yu Yeh, Deborah T. Marr and Yale N. Patt. Increasing the Instruction Fetch Rate via Multiple Branch Prediction and a Branch Address Cache. *1993 Intl. Conf. Supercomputing*, pp. 67-76, July 1993.

[4] Manoj Franklin and Mark Smotherman. A Fill-Unit Approach to Multiple Instruction Issue. *Proc. 27th Intl. Symp. on Microarchitecture*, pp. 162-171, Nov. 1994.

[5] Mark Smotherman and Manoj Franklin. Improving CISC Instruction Decoding Performance Using a Fill Unit. *Proc. 28th Intl. Symp. Microarchitecture*, pp. 219-229, Nov. 1995.

[6] Alexander Peleg and Uri Weiser. *Dynamic Flow Instruction Cache Memory Organized Around Trace Segments Independent of Virtual Address Line*. United States Patent 5,381,533, Jan. 10, 1995. http://www.patents.ibm.com/.

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

[8] Eric Rotenberg, Steve Bennett and Jim Smith. Trace Cache: *A Low Latency Approach to High Bandwidth Instruction Fetching*. University of Wisconsin, Madison Tech. Report. April, 1996.

[9] Eric Rotenberg, Steve Bennett and James E. Smith. Trace Cache: A Low Latency Approach to High Bandwidth Instruction Fetching. *Proc. of the 29th Intl. Symp. on Microarchitecture*, pp. 24-34, Dec. 1996.

[10] Sanjay Jeram Patel, Daniel Holmes Friendly and Yale N. Patt. *Critical Issues Regarding the Trace Cache Fetch Mechanism*. Computer Science and Engineering, University of Michigan Tech. Report. May 1997.

[11] Daniel Holmes Friendly, Sanjay Jeram Patel and Yale N. Patt. Alternative Fetch and Issue Policies for the Trace Cache Fetch Mechanism. *Proc. 30th Intl. Symp. Microarchitecture*, pp. 24-33, Dec. 1997.

[12] Eric Rotenberg, Quinn Jacobson, Yiannakis Sazeides and Jim Smith. Trace Processors. *Proc. 30th Intl. Symp. Microarchitecture*, pp. 138-148, Dec. 1997.

[13] Eric Rotenberg, Steve Bennett and James E. Smith. A Trace Cache Microarchitecture and Evaluation. *IEEE Transactions on Computers*, Vol. 48, No. 2, pp. 111-120. February 1999.

[14] Douglas C. Burger and Todd M. Austin. *The SimpleScalar Tool Set, Version 2.0*. University of Wisconsin, Madison Tech. Report. June 1997.

[15] Sanjay J. Patel, Marius Evers and Yale N. Patt. Improving Trace Cache Effectiveness with Branch Promotion and Trace Packing. *Proc. 25th Intl. Symp. Computer Architecture*, pp. 262-271, June 1998.

[16] Eric Hao, Po-Yung Chang and Yale N. Patt. The Effect of Speculatively Updating Branch History on Branch Prediction Accuracy, Revisited. *Proc. 27th Intl. Symp. Microarchitecture*, pp. 228-232, Nov, 1994.

[17] Sanjay J. Patel. *Delivering Instruction Bandwidth using a Trace Cache*. Ph.D. Dissertation. University of Michigan, 1999.

[18] Q. Jacobson, E. Rotenberg, and J. Smith. Path-based Next Trace Prediction. *30th Intl. Symp. Microarchitecture*, Dec. 1997.

[19] Srilatha Manne, Artur Klauser and Dirk Grunwald. Pipeline Gating: Speculation Control For Energy Reduction. *Proc. 25th Intl. Symp. Computer Architecture*, pp. 132-141, June 1998.

[20] T. Sato. NCB: A mechanism for improving instruction fetching efficiency. *Proc. 9th Joint Symp. on Parallel Processing*, pp. 221-228, May 1997.

[21] Alex Ramirez, Josep Ll. Larriba-Pey, Carlos Navarro, Josep Torrellas, and Mateo Valero. Software trace cache. *Proc. 13th Intl. Conf. on Supercomputing*, pp. 119-126, June 1999.

[22] Wen-mei W. Hwu, Scott A. Mahlke, William Y. Chen, Pohua P. Chang, Nancy J. Warter, Roger A. Bringmann, Roland G. Ouellette, Richard E. Hank, Tokuzo Kiyohara, Grant E. Haab, J. G. Holm and D. M. Lavery. The Superblock: An Effective Technique for VLIW and Superscalar Compilation. *The Journal Supercomputing*, Vol. 7 No. 1-2, pp. 229-248. May 1993.

[23] 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.

[24] P. Y. Hsu and E. S. Davidson. Highly concurrent scalar processing. *Proc. 18th Intl. Symp. Computer Architecture*, pp. 386-395, June 1986.