A Trace Cache Microarchitecture and Evaluation

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

Steve Bennett
Intel Corporation
sbennett@ichips.intel.com

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

Abstract

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

Trace caches overcome this limitation by caching traces *of the dynamic instruction stream, so instructions that are otherwise noncontiguous appear contiguous. In this paper we present and evaluate a microarchitecture incorporating a trace cache. The microarchitecture provides high instruction fetch bandwidth with low latency by explicitly sequencing through the program at the higher level of traces, both in terms of (1) control flow prediction and (2) instruction supply. For the SPEC95 integer benchmarks, trace-level sequencing improves performance from 15% to 35% over an otherwise equally-sophisticated, but contiguous multiple-block fetch mechanism. Most of this performance improvement is due to the trace cache. However, for one benchmark whose performance is limited by branch mispredictions, the performance gain is due almost entirely to improved prediction accuracy.*

Keywords: instruction cache, instruction fetching, multiple branch prediction, superscalar processors, trace cache

1. Introduction

High performance superscalar processor organizations divide naturally into an instruction fetch mechanism and an instruction execution mechanism. These two mechanisms are separated by instruction issue buffers, for example, issue queues or reservation stations. Conceptually, the instruction fetch mechanism acts as a "producer" which fetches, decodes, and dispatches instructions into the buffer. The instruction execution engine is the "consumer" which issues instructions from the buffer and executes them, subject to data dependence and resource constraints.

The instruction issue buffers are collectively called the instruction *window*. The window is the mechanism for exposing instruction-level parallelism (ILP) in sequential programs: a larger window increases the opportunity for finding data-independent instructions that may issue and execute in parallel. Thus, the trend in superscalar design is to construct larger instruction windows, and provide wider issue/execution paths to exploit the corresponding increase in available ILP.

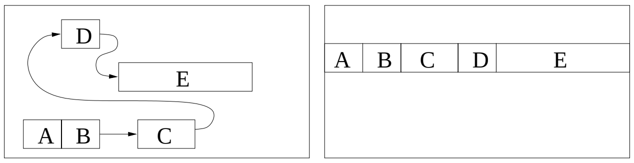
These trends place increased demand on the instruction supply mechanism. In particular, the peak instruction fetch rate should match the peak instruction issue rate, or the benefit of aggressive ILP techniques are diminished.

In this paper, we are concerned with instruction fetch bandwidth becoming a performance bottleneck. Current fetch units are limited to one branch prediction per cycle and can therefore fetch no more than one basic block per cycle. Previous studies have shown, however, that the average size of basic blocks in integer codes is small, around four to six instructions [30, 3]. While fetching a single basic block each cycle is sufficient for implementations that issue at most four instructions per cycle, it is not for processors with higher peak issue rates. If multiple branch prediction [30, 3, 4, 26] is used, then the fetch unit can at least fetch multiple *contiguous* basic blocks in a cycle. As will be shown in this paper, fetching multiple contiguous basic blocks is important, but the upper bound on fetch bandwidth is still limited due to the frequency of taken branches. Therefore, if a taken branch is encountered, it is necessary to fetch instructions down the taken path in the same cycle that the branch is fetched.

1.1. The trace cache

The job of the fetch unit is to feed the *dynamic* instruction stream to the decoder. A problem is that instructions are placed in the cache in their compiled order. Storing programs in this *static* form favors fetching code with infrequent taken branches or with large basic blocks. Neither of these cases is typical of integer programs.

Figure 1(a) shows an example dynamic sequence of basic blocks as they are stored in the instruction cache. The arrows indicate taken branches. Even with multiple branch predictions per cycle, four cycles are required to fetch the instructions in basic blocks ABCDE because the instructions are stored in noncontiguous cache locations.



(a) Instruction cache. (b) Trace cache.

Figure 1. Storing a noncontiguous sequence of instructions.

It is for this reason that several researchers have proposed a special instruction cache for capturing long dynamic instruction sequences [15, 22, 23, 24, 21]. This structure is called a *trace cache* because each line stores a snapshot, or trace, of the dynamic instruction stream. Referring again to Figure 1, the same dynamic sequence of blocks that appear noncontiguous in the instruction cache are contiguous in the trace cache (Figure 1(b)).

The primary constraint on a trace is a maximum length, determined by the trace cache line size. There may be any number of other implementation-dependent constraints, such as the number and type of embedded control transfer instructions, or special terminating conditions for tuning various performance factors [25].

A trace is fully specified by a starting address and a sequence of branch outcomes which describe the path followed. The first time a trace is encountered, it is allocated a line in the trace cache. The line is filled as instructions are fetched from the instruction cache. If the same trace is encountered again in the course of executing the program, i.e. the same starting address and predicted branch outcomes, it will be available in the trace cache and is fed directly to the decoder in a single cycle. Otherwise, fetching proceeds normally from the instruction cache.

Other high bandwidth fetch mechanisms have been proposed that are based on the conventional instruction cache [30, 4, 3, 26]. Every cycle, instructions from noncontiguous locations are fetched from the instruction cache and assembled into the predicted dynamic sequence. This typically requires multiple pipeline stages: (1) a level of indirection through special branch target tables to generate pointers to all of the noncontiguous instruction blocks, (2) a moderate to highly interleaved instruction cache to provide simultaneous access to multiple lines, with the possibility for bank conflicts, and (3) a complex alignment network to shift and align blocks into dynamic program order, ready for decoding/renaming.

The trace cache approach avoids this complexity by caching dynamic instruction sequences themselves, rather than information for constructing them. If the predicted dynamic sequence exists in the trace cache, it does not have to be recreated on the fly from the instruction cache's static representation. The cost of this approach is redundant instruction storage: the same instructions may reside in both the primary cache and the trace cache, and there is redundancy among different lines in the trace cache.

1.2. Related prior work

Alternative High Bandwidth Fetch Mechanisms

Four previous studies have focused on mechanisms to fetch multiple, possibly noncontiguous basic blocks each cycle from the instruction cache. These are the *branch address cache* [30], the *subgraph predictor* [4], the *collapsing buffer* [3], and the *multiple-block ahead predictor* [26].

Trace Cache Development

Melvin, Shebanow, and Patt proposed the *fill unit* and *multinodeword cache* [18, 16]. The first work qualitatively describes the performance implications of smaller or larger *atomic* units of work at the instruction-set architecture (ISA), compiler, and hardware levels. The authors argue for small compiler atomic units and large execution atomic units to achieve highest performance. The fill unit is proposed as the hardware mechanism for compacting the smaller compiler units into the large execution units, which are then stored for reuse in a decoded instruction cache. The follow-on work [16] evaluates the performance potential of large execution atomic units. Although this work only evaluates sizes up to that of a single VAX instruction and a basic block, it also suggests joining two consecutive basic blocks if the intervening branch is “highly predictable”.

In [17], software basic block enlargement is discussed. In the spirit of trace scheduling [5] and trace selection [11], the compiler uses profiling to identify candidate basic blocks for merging into a single execution atomic unit. The hardware sequences at the level of execution atomic units as created by the compiler. The advantage of this approach is the compiler can optimize and schedule across basic block boundaries.

Franklin and Smotherman [6] extended the fill unit's role to dynamically assemble VLIW-like instruction words from a RISC instruction stream, which are then stored in a *shadow cache*. This structure eases the issue complexity of a wide issue processor. They further applied the fill unit and a decoded instruction cache to improve the decoding performance of a complex instruction-set computer (CISC) [27]. In both cases the cache lines are augmented to store *trees* to improve the utilization of each line.

Four works have independently proposed the trace cache as a complexity-effective approach to high bandwidth instruction fetching. Johnson [15] proposed the *expansion cache*, which addresses cache alignment, branch prediction throughput, and instruction run merging. The expansion process also predetermines the execution schedule of instructions in a line. Unlike a pure VLIW cache, the schedule may consist of multiple cycles via *cycle tagging*. Peleg and Weiser [22] describe the design of a *dynamic flow instruction cache* which stores instructions independent of their virtual addresses, the defining characteristic of trace caches. Rotenberg, Bennett, and Smith [23, 24] motivate the concept with comparisons to other high bandwidth fetch mechanisms proposed in the literature, and defines some of the trace cache design space. Patel, Friendly, and Patt [21] expand upon and present detailed evaluations of this design space, arguing for a more prominent role of the trace cache.

The *mispredict recovery cache* proposed by Bondi, Nanda, and Dutta [1] caches instruction threads from alternate paths of mispredicted branches. The goal of this work is to quickly bypass the multiple fetch and decode stages of a long CISC pipeline following a branch mispredict. Nair and Hopkins [19] employ *dynamic instruction formatting* to cache large scheduled groups, similar in spirit to the cycle tagging approach of the expansion cache.

There has also been recent work incorporating trace caches into new processing models. Vajapeyam and Mitra [29], Sundararaman and Franklin [28], and Rotenberg, Jacobson, Sazeides, and Smith [25] exploit the data and control hierarchy implied by traces to overcome complexity and architectural hurdles of superscalar processors. Jacobson, Rotenberg, and Smith [14] propose a control prediction model well suited to the trace cache called *next trace prediction*, discussed in later sections. Friendly, Patel, and Patt propose a new processing model called *inactive issue* for reducing the effects of branch mispredictions [7], and dynamically optimizing traces before storing them in the trace cache, reducing their execution time significantly [8].

Microcode, VLIW, and Block-Structured ISAs

Clearly the concept of traces exists in the software realm of instruction-level parallelism. Early work by Fisher [5], Hwu and Chang [11], and others on trace scheduling and trace selection for microcode recognized the problem imposed by branches on code optimization. Subsequent VLIW architectures and novel ISA techniques, for example [12, 10], further promote the ability to schedule long sequences of instructions containing multiple branches.

2. Trace cache microarchitecture

In Section 1.1 we introduced the concept of the trace cache – an instruction cache which captures dynamic instruction sequences, or traces. We now present a microarchitecture organized around traces.

2.1. Trace-level sequencing

The premise of the proposed microarchitecture, shown in Figure 2, is to provide high instruction fetch bandwidth with low latency. This is achieved by explicitly sequencing through the program at the higher level of traces, both for (1) control flow prediction and (2) supplying instructions.

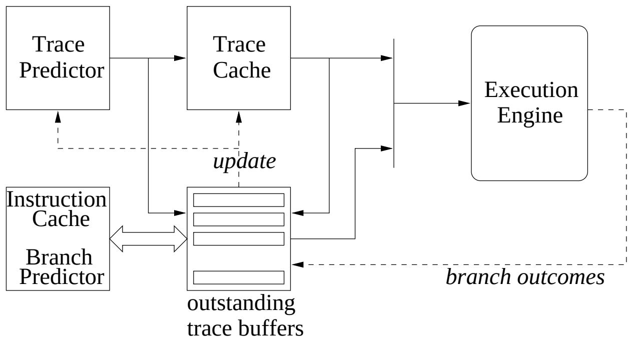


Figure 2. Microarchitecture.

A *next trace predictor* [14] treats traces as basic units and explicitly predicts sequences of traces. Because traces are the unit of prediction, rather than individual branches, high branch prediction throughput is implicitly achieved with only a single trace prediction per cycle. Jacobson et al [14] demonstrated that explicit trace prediction not only removes fundamental constraints on the number of branches in a trace (usually a consequence of adapting single branch predictors to multiple branch predictor counterparts [23]), but it also holds the potential for achieving higher overall branch prediction accuracy than single branch predictors. Details of next trace prediction are presented in Section 2.3.

The output of the trace predictor is a *trace identifier*: a given trace is uniquely identified by its starting PC and the outcomes of all conditional branches embedded in the trace. The trace identifier is used to lookup the trace in the trace cache. The index into the trace cache can be derived from just the starting PC, or a combination of PC and branch outcomes. Using branch outcomes in the index has the advantage of providing *path associativity* – multiple traces eminating from the same start PC can reside simultaneously in the trace cache even if it is direct mapped [24].

The output of the trace cache is one or more traces, depending on the cache associativity. A trace identifier is stored with each trace in order to determine a trace cache hit, analogous to the tag of conventional caches. The desired trace is present in the cache if one of the cached trace identifiers matches the predicted trace identifier.

The trace predictor and trace cache together provide fast trace-level sequencing. Unfortunately, trace-level sequencing does not always provide the required trace. This is particularly true at the start of the program or when a new region of code is reached – neither the trace predictor nor the trace cache has "learned" any traces yet. Instruction-level sequencing, discussed in the next section, is required to construct non-existent traces or repair trace mispredictions.

2.2. Instruction-level sequencing

The *outstanding trace buffers* in Figure 2 are used to (1) construct new traces that are not in the trace cache and (2) track branch outcomes as they become available from the execution engine, allowing detection of mispredictions and repair of the traces containing them.

Each fetched trace is dispatched to both the execution engine and an outstanding trace buffer. In the case of a trace cache miss, only the trace prediction is received by the allocated buffer. The trace prediction itself provides enough information to construct the trace from the instruction cache, although this typically requires multiple cycles due to predicted-taken branches.

In the case of a trace cache hit, the trace is dispatched to the buffer. This allows repair of a partially mispredicted trace, i.e. when a branch outcome returned from execution does not match the path indicated within the trace. In the event of a branch misprediction, the trace buffer begins reconstructing the tail of the trace (or all of the trace if the start PC is incorrect) using the corrected branch target and the instruction cache. For subsequent branches in the trace, a second-level branch predictor is used to make predictions.

We advocate an aggressive instruction cache design for providing robust performance over a broad range of trace cache miss rates. The instruction cache is 2-way interleaved so that up to a full cache line can be fetched each cycle, independent of PC alignment [9]. The second-level branch prediction mechanism is simple – a 2-bit counter and branch target stored with each branch. Logically, the instructions, counters, and targets are all stored in the instruction cache (as opposed to a separate cache and branch target buffer) to allow fast, parallel prediction of any number of not-taken branches. We call this instruction fetch mechanism SEQ.n in keeping with the terminology of [24] – any number (denoted *n*) of *sequential* basic blocks, up to the line size, can be fetched in a single cycle.

When a trace buffer is through constructing its trace, it is written into the trace cache and dispatched to the execution engine. If the newly constructed trace is a result of misprediction recovery, the trace identifier is also sent to the trace predictor for repairing its path history.

2.3. Next trace prediction

The next trace predictor, shown in Figure 3, is based on Jacobson's work on path-based, high-level control flow prediction [13, 14].

An index into a correlated prediction table is formed from the sequence of past trace identifiers. The hash function used to generate the index is called a **DOLC** function: **'D'**epth specifies the path history depth in terms of traces; **'O'**ldest indicates the number of bits selected from each trace identifier except the two most recent ones; **'L'**ast and **'C'**urrent indicate the number of bits selected from the second-most recent and most recent trace identifiers, respectively.

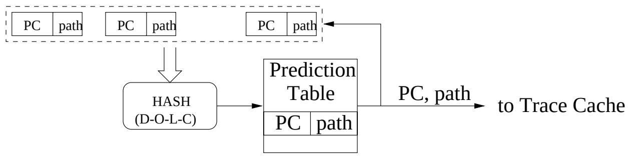


Figure 3. Jacobson's next trace predictor.

Each entry in the correlated prediction table contains a trace identifier and a 2-bit counter for replacement. The predictor is augmented with several other mechanisms [14].

- *Hybrid prediction*. In addition to the correlated table, a second, smaller table is indexed with only the most recent trace identifier. This second table requires a shorter learning time and suffers less aliasing pressure.

- *Return history stack*. At call instructions, the path history is pushed onto a special stack. When the corresponding return point is reached, path history before the call is restored. This improves accuracy because control flow following a subroutine is highly correlated with control flow before the call.

- *Alternate trace identifier*. An entry in the correlated table may be augmented with an alternate trace prediction, a form of associativity in the predictor. If a trace misprediction is detected, the outstanding trace buffer responsible for repairing the trace can use the alternate prediction *if it is consistent with known branch outcomes in the trace*. If so, the trace buffer does not have to resort to the second-level branch predictor; instruction-level sequencing is avoided altogether if the alternate trace also hits in the trace cache.

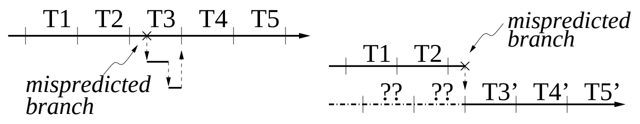
2.4. Trace selection

The performance of the trace cache is strongly dependent on *trace selection*, the algorithm used to divide the dynamic instruction stream into traces. Trace selection primarily affects average trace length and trace cache hit rate, both of which, in turn, affect fetch bandwidth. The interaction between trace length and hit rate, however, is not well understood. Preliminary studies indicate that longer traces result in lower hit rates, but this may be an artifact of naive trace selection policies. Sophisticated selection techniques that are conscious of control flow constructs – loop back-edges, loop fall-through points, call sites, and re-convergent points in general – may lead to different conclusions. The reader is referred to [21, 25, 20] for a few interesting control-flow-conscious selection heuristics.

Trace selection in this paper is constrained only by the maximum trace length of 16 instructions, and indirect branches (returns and jump/call indirects) terminate traces.

2.5. Hierarchical sequencing

In Figure 4(a), a portion of the dynamic instruction stream is shown with a solid horizontal arrow from left to right. The stream is divided into traces T1 through T5. This sequence of traces is produced independent of where the instructions come from – trace predictor/trace cache, trace predictor/instruction cache, or branch predictor/instruction cache.



(a) Hierarchical. (b) Non-hierarchical.

Figure 4. Two sequencing models.

For example, if the trace predictor mispredicts T3, the trace buffer assigned to T3 resorts to instruction-level sequencing. This is shown in the diagram as a series of steps, depicting smaller blocks fetched from the instruction cache. The trace buffer strictly adheres to the boundary between T3 and T4, dictated by trace selection, even if the final instruction cache fetch produces a larger block of sequential instructions than is needed by T3 itself.

We call this process *hierarchical sequencing* because there exists a clear distinction between inter-trace control flow and intra-trace control flow. Inter-trace control flow, i.e. trace boundaries, is effectively *pre-determined* by trace selection and is unaffected by dynamic effects such as trace cache misses and mispredictions.

A contrasting sequencing model is shown in Figure 4(b). In this model, trace selection is "reset" at the point of the mispredicted branch, producing the shifted traces T3′, T4′, and T5′. This sequencing model does not work well with path-based next trace prediction. After resolving the branch misprediction, trace T3′ and subsequent traces must somehow be predicted. However, this requires a sequence of traces leading to T3′ and no such sequence is available (indicated with question marks in the diagram).

A potential problem with hierarchical sequencing is misprediction recovery latency. Explicit next trace prediction uses a level of indirection: a trace is first predicted, and then the trace cache is accessed. This implies an extra cycle is added to the latency of misprediction recovery. However, this extra cycle is not exposed. First, consider the case in which the alternate trace prediction is used. The primary and alternate predictions are supplied by the trace predictor at the same time, and stored together in the trace buffer. Therefore, the alternate prediction is immediately available for accessing the trace cache when the misprediction is detected. Second, if the alternate is not used, then the second-level branch predictor and instruction cache are used to fetch instructions from the correct path. In this case, the instruction cache is accessed immediately with the correct branch target PC returned by the execution engine.

In our evaluation, we assume a trace must be fully constructed before any of its instructions are dispatched to the execution engine, because traces are efficiently renamed as a unit [29, 25]. This aggravates both trace misprediction and trace cache miss recovery latency. We want to make it clear, however, that this is not due to any fundamental constraint of the fetch model, only an artifact of our dispatch model.

3. Simulation methodology

3.1. Fetch models

To evaluate the performance of the trace cache microarchitecture, we compare it to several more constrained fetch models. We first determine the performance advantage of fetching multiple *contiguous* basic blocks per cycle over conventional single block fetching. Then, the benefit of fetching multiple *noncontiguous* basic blocks is isolated.

In all models a next trace predictor is used for control prediction, for two reasons. First, next trace prediction is highly accurate, and whether predicting one or many branches at a time, it is comparable to or better than some of the best single branch predictors in the literature. Second, it is desirable to have a common underlying predictor for all fetch models so we can separate performance due to fetch bandwidth from that due to branch prediction (more on this in Section 3.2).

What differentiates the following models is the trace selection algorithm.

- SEQ.1 (*"sequential, 1 block"*): A "trace" is a single basic block up to 16 instructions in length.
- SEQ.n (*"sequential, n blocks"*): A "trace" may contain any number of sequential basic blocks up to the 16 instruction limit.
- TC (*"trace cache"*): A trace may contain any number of conditional branches, both taken and not-taken, up to 16 instructions or the first indirect branch.

The SEQ.1 and SEQ.n models do not use a trace cache because an interleaved instruction cache is capable of supplying a "trace" in a single cycle [9] – a consequence of the sequential selection constraint. Therefore, one may view the SEQ.1/SEQ.n fetch unit as identical to the trace cache microarchitecture in Figure 2, except the trace cache block is replaced with a conventional instruction cache. That is, the next trace predictor drives a conventional instruction cache, and the trace buffers are used to construct "traces" from the L2 cache/main memory if not present in the cache.

Finally, to establish an upper bound on the performance of noncontiguous instruction fetching, we introduce a fourth model, TC-perfect, which is identical to TC but the trace cache always hits.

3.2. Isolating trace predictor/trace cache performance

An interesting side-effect of trace selection is that it significantly affects trace prediction accuracy. In general, smaller traces (resulting from more constrained trace selection) result in lower accuracy. We have determined at least two reasons for this. First, longer traces naturally capture longer path history. This can be compensated for by using more trace identifiers in the path history if the traces are small; that is, a good DOLC function for one trace length is not necessarily good for another. For the TC model, DOLC = {7,3,6,8} (a depth of 7 traces) consistently performs well over all benchmarks [14]. For SEQ.1 and SEQ.n, a brief search of the design space shows DOLC = {17,3,4,12} (a depth of 17 traces) performs well.

We have observed, however, that tuning the DOLC parameters is not enough – trace selection affects accuracy in other ways. The graph in Figure 5 shows trace predictor performance using an unbounded table, i.e. using full, unhashed path history to make predictions. The graph shows trace mispredictions per 1000 instructions for SEQ.1, SEQ.n, and TC trace selection, as the history depth is varied. For the *go* benchmark, trace mispredictions for the SEQ.n model do not dip below 8.8 per 1000 instructions, whereas the TC model reaches as few as 8.0 trace mispredictions per 1000 instructions. Unconstrained trace selection results in the creation of many unique traces. While this trace explosion generally has a negative impact on trace cache performance, we hypothesize it also creates many more unique contexts for making predictions. A large prediction table can exploit this additional context.

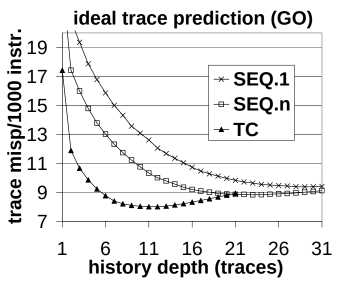


Figure 5. Impact of trace selection on unbounded trace predictor performance.

We conclude that it is difficult to separate the performance advantage of the trace cache from that of the trace predictor, because both show positive improvement with longer traces. Nonetheless, when we compare TC to SEQ.n or SEQ.1, we would like to know how much benefit is derived from the trace cache itself.

To this end, we developed a methodology to statistically "adjust" the overall branch prediction accuracy of a given fetch model to match that of another model. The trace predictor itself is not adjusted – it produces predictions in the normal fashion. However, after making a prediction, the predicted trace is compared with the *actual* trace, determined in advance by a functional simulator running in parallel with the timing simulator. If the prediction is incorrect, the actual trace is substituted for the mispredicted trace *with some probability*. In other words, some fraction of mispredicted traces are corrected. The probability for injecting corrections was chosen on a per-benchmark basis to achieve the desired branch misprediction rate.

This methodology introduces two additional fetch models, SEQ.1-adj and SEQ.n-adj, corresponding to the "adjusted" SEQ.1 and SEQ.n models. Clearly these models are unrealizable, but they are useful for performance comparisons because their adjusted branch misprediction rates match that of the TC model.

3.3. Simulator and benchmarks

A detailed, fully-execution driven superscalar processor simulator is used to evaluate the trace cache microarchitecture. The simulator was developed using the *simplescalar* platform [2]. This platform uses a MIPS-like instruction set and a gcc-based compiler to create binaries.

The datapath of the fetch engine as shown in Figure 2 is faithfully modeled. The next trace predictor has 2^{16} entries. The DOLC functions for compressing the path history into a 16-bit index were described earlier in Section 3.2, for both the TC and SEQ models. The trace cache configuration – size, associativity, and indexing – is varied. There are sufficient outstanding trace buffers to keep the instruction window full. The trace buffers share a single port to the combined instruction cache and second-level branch predictor. The instruction cache is 64KB, 4-way set-associative, and 2-way interleaved. The line size is 16 instructions and the cache hit and miss latencies are 1 cycle and 12 cycles respectively. The second-level branch predictor consists of 2-bit counters and branch targets, assumed to be logically stored with each branch in the instruction cache.

An instruction window of 256 instructions is used in all experiments. The processor is 16-way superscalar, i.e. the processor can fetch and issue up to 16 instructions each cycle. Five basic pipeline stages are modeled. Instruction *fetch* and *dispatch* take 1 cycle each. *Issue* takes at least 1 cycle, possibly more if the instruction must stall for operands; any 16 instructions, including loads and stores, may issue each cycle. *Execution* takes a fixed latency based on instruction type, plus any time spent waiting for a result bus. Instructions *retire* in order.

For loads and stores, address generation takes 1 cycle and the cache access is 2 cycles for a hit. The data cache is 64KB, 4-way set-associative with a line size of 64 bytes and a miss penalty of 14 cycles. Realistic but aggressive memory disambiguation is modeled. Loads may proceed ahead of any unresolved stores, and any memory hazards are detected as store addresses become available – recovery is via selective reissuing of misspeculated loads and their dependent instructions [25].

Seven of the SPEC95 integer benchmarks, shown in Table 1, are simulated to completion.

Table 1. Benchmarks.

benchmark	input dataset	dynamic instr. count
gcc	-O3 genrecog.i	117M
go	9 9	133M
jpeg	vigo.ppm	166M
li	queens 7	202M
m88ksim	-c < ctl.in (dcrand.big)	120M
perl	scrabbl.pl < scrabbl.in	108M
vortex	persons.250	101M

4. Results

4.1. Performance of fetch models

Figure 6 shows the performance of the six fetch models in terms of retired instructions per cycle (IPC). The TC model in this section uses a 64KB (instruction storage only), 4-way set-associative trace cache. The trace cache is indexed using only the PC (i.e. no explicit path associativity, except that afforded by the 4 ways).

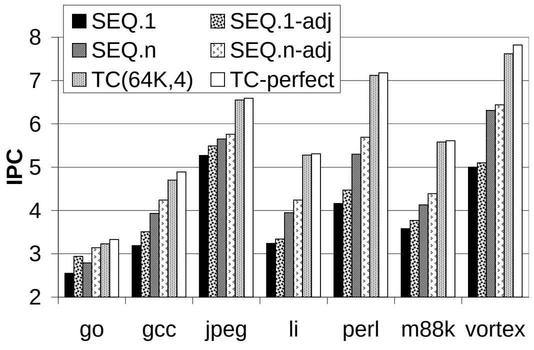


Figure 6. Performance of the fetch models.

We can draw several conclusions from the graph in Figure 6. First, comparing the SEQ.n models to the SEQ.1 models, it is apparent that predicting and fetching multiple *sequential* basic blocks provides a significant performance advantage over conventional single-block fetching. The graph in Figure 7 shows that the performance advantage of the SEQ.n model over the SEQ.1 model ranges from about 5% to 25%, with the majority of benchmarks showing greater than 15% improvement. Similar results hold whether or not branch prediction accuracy is adjusted for the SEQ.n and SEQ.1 models.

This first observation is important because the SEQ.n model only requires a more sophisticated, high-level control flow predictor, and retains a more-or-less conventional instruction cache microarchitecture.

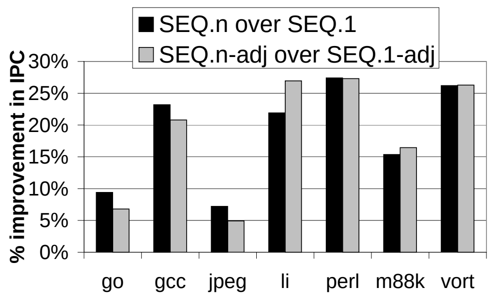


Figure 7. Speedup of SEQ.n over SEQ.1.

Second, the ability to fetch multiple, possibly *noncontiguous* basic blocks improves performance significantly over sequential-only fetching. The graph in Figure 8 shows that the performance advantage of the TC model over the SEQ.n model ranges from 15% to 35%.

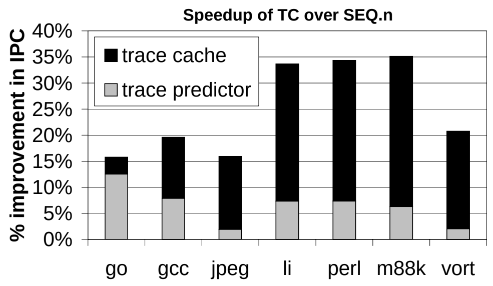


Figure 8. Speedup of TC over SEQ.n.

Figure 8 also isolates the contributions of next trace prediction and the trace cache to performance. The lower part of each bar is the speedup of model SEQ.n-adj over SEQ.n. And since the overall branch misprediction rate of SEQ.n-adj is adjusted to match that of the TC model, this part of the bar approximately isolates the impact of next trace prediction on performance. The top part of the bar therefore isolates the impact of the trace cache on performance.

For *go*, which suffers noticeably more branch mispredictions than other benchmarks, most of the benefit of the TC model comes from next trace prediction. In this case, the longer traces of the TC model are clearly more valuable for improving the context used by the next trace predictor than for providing raw instruction bandwidth. For *gcc*, however, both next trace prediction and the trace cache contribute equally to performance. The other five benchmarks benefit mostly from higher fetch bandwidth.

Finally, Figure 6 shows the moderately large trace cache of the TC model very nearly reaches the performance upper bound established by TC-perfect (within 4%).

Table 2 shows trace- and branch-related measures. Average trace lengths for TC range from 12.4 (*li*) to 15.8 (*jpeg*) instructions (1.6 to over 2 times longer than SEQ.n traces).

The table also shows predictor performance: primary and alternate trace mispredictions per 1000 instructions, and overall branch misprediction rates (the latter is computed by checking each branch at retirement to see if it caused a misprediction, whether originating from the trace predictor or second-level branch predictor). In all cases prediction improves with longer traces. TC has from 20% to 45% fewer trace mispredictions than SEQ.1, resulting in 15% (*jpeg*) to 41% (*m88ksim*) fewer total branch mispredictions. Note that the *adjusted* branch misprediction rates for the SEQ models are nearly equal to those of TC.

Shorter traces, however, generally result in better alternate trace prediction accuracy. Shorter traces result in (1) fewer total traces and thus less aliasing, and (2) fewer possible alternative traces from a given starting PC. For all benchmarks except *gcc* and *go*, the alternate trace prediction is almost always correct given the primary trace prediction is incorrect – both predictions taken together result in fewer than 1 trace misprediction per 1000 instructions.

Trace caches introduce redundancy – the same instruction can appear multiple times in one or more traces. Table 2 shows two redundancy measures. The *overall redundancy factor*, $RF_{overall}$, is computed by maintaining a table of all unique traces ever retired. Redundancy is the ratio of total number of instructions to total number of *unique* instructions for traces collected in the table. $RF_{overall}$ is independent of trace cache configuration and does not capture dynamic behavior. The *dynamic redundancy factor*, RF_{dyn}, is computed similarly, but using only traces *in the trace cache in a given cycle*; the final value is an average over all cycles. RF_{dyn} was measured using a 64KB, 4-way trace cache.

$RF_{overall}$ varies from 2.9 (*vortex*) to 14 (*go*). RF_{dyn} is less than $RF_{overall}$ and only ranges between 2 and 4, because the fixed size trace cache limits redundancy, and perhaps temporally there is less redundancy.

4.2. Trace cache size and associativity

In this section we measure performance of the TC model as a function of trace cache size and associativity. Figure 9 shows overall performance (IPC) for 12 trace cache configurations: direct mapped, 2-way, and 4-way associativity for each of four sizes, 16KB, 32KB, 64KB, and 128KB.

Associativity has a noticeable impact on performance for all of the benchmarks except *go*. *Go* has a particularly large working set of unique traces [25], and total capacity is more important than individual trace conflicts. The curves of *jpeg* and *li* are fairly flat – size is of little importance, yet increasing associativity improves performance. These two benchmarks suffer few general conflict misses (otherwise size should improve performance), yet conflicts among traces with the same start PC are significant. Associativity allows simultaneously caching these path-associative traces.

The performance improvement of the largest configuration (128KB, 4-way) with respect to the smallest one (16KB, direct mapped) ranges from 4% (*go*) to 10% (*gcc*).

Figure 10 shows trace cache performance in *misses per 1000 instructions*. Trace cache size is varied along the x-axis, and there are six curves: direct mapped (DM), 2-way (2W), and 4-way (4W) associative caches, both with and without indexing for path associativity (PA). We chose (somewhat arbitrarily) the following index function for achieving path associativity: the low-order bits of the PC form the set index, and then the high-order bits of this index

Table 2. Trace statistics.

model	measure	gcc	go	jpeg	li	m88k	perl	vort
SEQ.1	trace length	4.9	6.2	8.3	4.2	4.8	5.1	5.8
	trace misp./1000	8.8	14.5	5.2	6.9	3.5	3.4	1.5
	alt. trace misp./1000	2.1	4.5	0.1	0.6	0.4	0.1	0.2
	branch misp. rate	5.0%	11.0%	7.7%	3.7%	2.2%	2.2%	1.1%
	adjusted misp. rate	3.6%	8.2%	6.6%	3.2%	1.3%	1.4%	0.8%
SEQ.n	trace length	7.2	8.0	9.6	6.3	6.0	7.1	8.2
	trace misp./1000	7.3	12.7	4.6	6.9	3.3	3.1	1.2
	alt. trace misp./1000	2.7	5.4	0.5	0.9	0.6	0.3	0.3
	branch misp. rate	4.4%	10.1%	7.0%	3.7%	2.1%	2.0%	0.9%
	adjusted misp. rate	3.6%	8.1%	6.7%	3.1%	1.3%	1.4%	0.8%
TC	trace length	13.9	14.8	15.8	12.4	13.1	13.0	14.4
	trace misp./1000	5.4	9.6	4.2	5.5	2.0	2.1	1.0
	alt. trace misp./1000	2.7	5.3	0.9	1.3	0.5	0.3	0.3
	branch misp. rate	3.6%	8.2%	6.7%	3.1%	1.3%	1.5%	0.8%
	control instr. per trace	2.8	2.3	1.3	2.9	2.5	2.5	2.3
	$RF_{overall}$	7.1	14.4	5.3	3.1	3.7	4.1	2.9
	RF_{dyn}	3.0	3.3	3.7	3.2	3.1	2.9	2.1

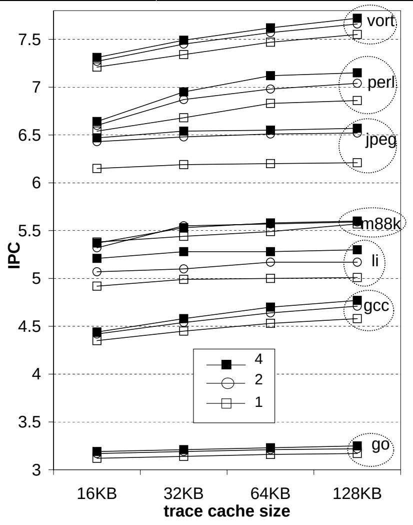


Figure 9. Performance vs. size/associativity.

are XORed with the first two branch outcomes of the trace identifier.

Gcc and *go* are the only benchmarks that do not fit entirely within the largest trace cache. As we observed earlier, *go* has many heavily-referenced traces, resulting in no fewer than 20 misses/1000 instructions.

Path associativity reduces misses substantially, particularly for direct mapped caches. Except for *vortex*, path associativity closes the gap between direct mapped and 2-way associative caches by more than half, and often entirely.

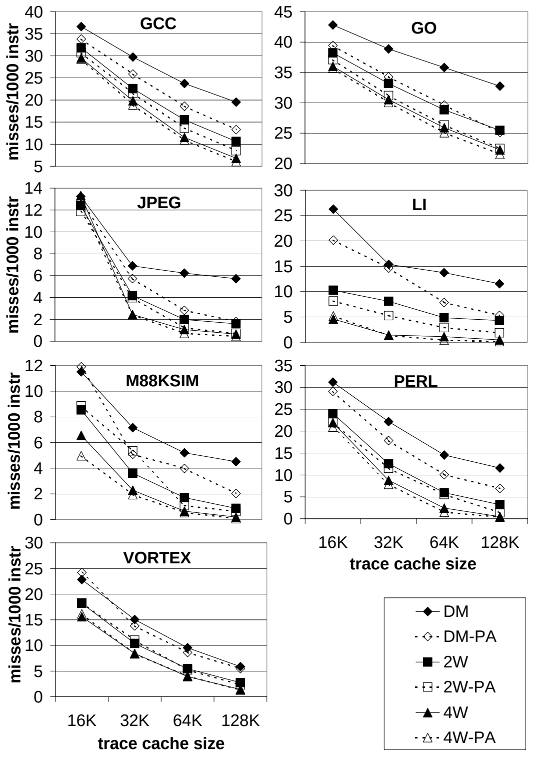


Figure 10. Trace cache misses.

5. Summary

It is important to design instruction fetch units capable of fetching past multiple, possibly taken branches each cycle. Trace caches provide this capability without the complexity and latency of equivalent-bandwidth instruction cache designs. We evaluated a microarchitecture incorporating a trace cache, with the following major results.

• The trace cache improves performance from 15% to 35% over an otherwise equally-sophisticated, but contiguous multiple-block fetch mechanism.

• Longer traces improve trace prediction accuracy. For the misprediction-bound benchmark *go*, this factor contributes almost entirely to the observed performance gain.

• A moderately large and associative trace cache performs as well as a perfect trace cache. For *go*, however, trace mispredictions mask poor trace cache performance.

• Overall performance is not as sensitive to trace cache size and associativity as one might expect, due in part to robust instruction-level sequencing. IPC varies no more than 10% over a wide range of configurations.

• The complexity advantage of the trace cache comes at the price of redundant instruction storage: for *gcc*, a factor of 7 redundancy among all traces created, corresponding to a factor of 3 redundancy in the trace cache.

• An instruction cache combined with an aggressive trace predictor can fetch any number of contiguous basic blocks per cycle, yielding from 5% to 25% improvement over single-block fetching.

Acknowledgments

Our research on trace caches had its genesis in stimulating group discussions with Guri Sohi and his students Todd Austin, Scott Breach, Andreas Moshovos, Dionisios Pnevmatikatos, and T. N. Vijaykumar; their contribution is gratefully acknowledged.

We would also like to give special thanks to Quinn Jacobson for his valuable input and for providing access to next trace prediction simulators.

This work was supported in part by NSF Grants MIP-9505853 and MIP-9307830 and by the U.S. Army Intelligence Center and Fort Huachuca under Contract DABT63-95-C-0127 and ARPA order no. D346.

Eric Rotenberg is supported by an IBM Fellowship.

References

[1] J. Bondi, A. Nanda, and S. Dutta. Integrating a misprediction recovery cache (mrc) into a superscalar pipeline. *29th Intl. Symp. on Microarchitecture*, pp. 14–23, Dec 1996.

[2] D. Burger, T. Austin, and S. Bennett. Evaluating future microprocessors: The simplescalar toolset. Tech Report CS-TR-96-1308, Univ. of Wisconsin, CS Dept., July 1996.

[3] 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.

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

[5] J. Fisher. Trace scheduling: A technique for global microcode compaction. *IEEE Transactions on Computers*, C-30(7):478–490, July 1981.

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

[7] D. Friendly, S. Patel, and Y. Patt. Alternative fetch and issue policies for the trace cache fetch mechanism. *30th Intl. Symp. on Microarchitecture*, pp. 24–33, Dec 1997.

[8] D. Friendly, S. Patel, and Y. Patt. Putting the fill unit to work: Dynamic optimizations for trace cache microprocessors. *31st Intl. Symp. on Microarchitecture*, Dec 1998.

[9] G. F. Grohoski, J. A. Kahle, L. E. Thatcher, and C. R. Moore. Branch and fixed-point instruction execution units. *IBM RISC System/6000 Technology*, Publication number SA23-2619, 1990.

[10] E. Hao, P.-Y. Chang, M. Evers, and Y. Patt. Increasing the instruction fetch rate via block-structured instruction set architectures. *29th Intl. Symp. on Microarchitecture*, pp. 191–200, Dec 1996.

[11] W. Hwu and P. Chang. Trace selection for compiling large c application programs to microcode. *21st Intl. Symp. on Microarchitecture*, Dec 1988.

[12] W. Hwu and et. al. The superblock: An effective technique for vliw and superscalar compilation. *Journal of Supercomputing*, 7(1):229–248, Jan 1993.

[13] Q. Jacobson, S. Bennett, N. Sharma, and J. E. Smith. Control flow speculation in multiscalar processors. *3rd Intl. Symp. on High Perf. Comp. Arch.*, pp. 218–229, Feb 1997.

[14] Q. Jacobson, E. Rotenberg, and J. Smith. Path-based next trace prediction. *30th Intl. Symp. on Microarchitecture*, pp. 14–23, Dec 1997.

[15] J. Johnson. Expansion caches for superscalar processors. Tech Report CSL-TR-94-630, Computer Science Laboratory, Stanford, CA, June 1994.

[16] S. Melvin and Y. Patt. Performance benefits of large execution atomic units in dynamically scheduled machines. *3rd Intl. Conf. on Supercomputing*, pp. 427–432, June 1989.

[17] S. Melvin and Y. Patt. Exploiting fine-grained parallelism through a combination of hardware and software techniques. *18th Intl. Symp. on Computer Architecture*, pp. 287–296, May 1991.

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

[19] R. Nair and M. Hopkins. Exploiting instruction level parallelism in processors by caching scheduled groups. *24th Intl. Symp. on Computer Architecture*, pp. 13–25, June 1997.

[20] S. Patel, M. Evers, and Y. Patt. Improving trace cache effectiveness with branch promotion and trace packing. *25th Intl. Symp. on Computer Architecture*, pp. 262–271, June 1998.

[21] S. Patel, D. Friendly, and Y. Patt. Critical issues regarding the trace cache fetch mechanism. Tech Report CSE-TR-335-97, University of Michigan, EECS Department, 1997.

[22] A. Peleg and U. Weiser. Dynamic flow instruction cache memory organized around trace segments independent of virtual address line. U.S. Patent Number 5,381,533, Jan 1995.

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

[24] E. Rotenberg, S. Bennett, and J. Smith. Trace cache: a low latency approach to high bandwidth instruction fetching. *29th Intl. Symp. on Microarchitecture*, pp. 24–34, Dec 1996.

[25] E. Rotenberg, Q. Jacobson, Y. Sazeides, and J. Smith. Trace processors. *30th Intl. Symp. on Microarchitecture*, pp. 138–148, Dec 1997.

[26] A. Seznec, S. Jourdan, P. Sainrat, and P. Michaud. Multiple-block ahead branch predictors. *7th Intl. Conf. on Arch. Support for Prog. Lang. and Operating Systems*, Oct 1996.

[27] M. Smotherman and M. Franklin. Improving cisc instruction decoding performance using a fill unit. *28th Intl. Symp. on Microarchitecture*, pp. 219–229, Nov 1995.

[28] K. Sundararaman and M. Franklin. Multiscalar execution along a single flow of control. *ICPP'97*, Aug 1997.

[29] S. Vajapeyam and T. Mitra. Improving superscalar instruction dispatch and issue by exploiting dynamic code sequences. *24th Intl. Symp. on Computer Architecture*, pp. 1–12, June 1997.

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

Eric Rotenberg is a graduate student in the Computer Sciences Department at the University of Wisconsin-Madison. He received a BS in electrical engineering (1991) and an MS in computer sciences (1996) from the same. From 1992 to 1994, he participated in the design of IBM's AS/400 computer in Rochester, MN. He is currently researching instruction-level parallelism, with emphasis on the problems imposed by control flow.

Steve Bennett is a research scientist in the Microprocessor Research Lab at Intel in Hillsboro, OR. His research interests include high performance computer architecture and simulation methodologies. Bennett received a BSE from the University of Michigan, Ann Arbor and an MS in computer sciences from the University of Wisconsin - Madison.

James E. Smith (Member) is with the Department of Electrical and Computer Engineering at the University of Wisconsin-Madison. He first joined the University of Wisconsin in 1976, performing research in fault-tolerant computing. From 1979 to 1981, he took a leave of absence to work for the Control Data Corporation in Arden Hills, MN, participating in the design of the CYBER 180/990. From 1984 to 1989, he participated in the development of the ACA ZS-1, a large-scale scientific computer employing a dynamically scheduled, superscalar processor architecture. In 1989, he joined Cray Research, Inc. in Chippewa Falls, WI. While at Cray Research, he headed a small research team that participated in the development and analysis of future supercomputer architectures. In 1994, he re-joined the Department of ECE at the University of Wisconsin where he holds the position of Professor. His current research interests focus on new paradigms for exploiting instruction level parallelism.