Trace Preconstruction

Quinn Jacobson
Sun Microsystems
901 San Antonio Road
Palo Alto, CA 94303-4900
408 616-5655
quinn.jacobson@sun.com

James E. Smith
University of Wisconsin – Madison
Department of Electrical & Computer Engineering
Madison, WI 53706
608 265-5737
jes@ece.wisc.edu

ABSTRACT

Trace caches enable high bandwidth, low latency instruction supply, but have a high miss penalty and relatively large working sets. Consequently, their performance may suffer due to capacity and compulsory misses. Trace preconstruction augments a trace cache by performing a function analogous to prefetching. The trace preconstruction mechanism observes the processor's instruction dispatch stream to detect opportunities for jumping ahead of the processor. After doing so, the preconstruction mechanism fetches static instructions from the predicted future region of the program, and constructs a set of traces in advance of when they are needed.

Trace preconstruction can significantly increase both the performance of the trace cache and the robustness of the trace cache to varying workloads. All but one of the SPECint95 benchmarks see a notable reduction in trace cache miss rates from preconstruction. The three benchmarks that have the largest working set (gcc, go and vortex) see a 30% to 80% reduction in trace cache misses. We also consider the integration of preconstruction with another trace-specific mechanism (preprocessing) to produce a high performance frontend. When combined, preconstruction and trace preprocessing produce an average speedup of 14% for the SPECint95 benchmarks.

1. Introduction

Trace caches [10][9] have been proposed as a mechanism to enable low latency, high bandwidth instruction fetching. Trace caches store programs in a representation that is a hybrid of the static program representation and the dynamic instruction stream. Traces are snapshots of short segments of the dynamic instruction stream that are cached. When a dynamic path is taken repetitively, instructions are provided from the trace cache, yielding a contiguous block of dynamic instructions that may correspond to noncontiguous blocks of code from the static representation.

Previous work has shown the potential benefit of adding trace caches to traditional processor cores [10][9], and of developing processors specifically around the trace cache [11][4][8]. The latter approach provides reduced complexity and localized communication, as well as the ability to optimize programs dynamically.

The dynamic behavior of traces, which enables the trace cache to provide high instruction fetch bandwidth, also makes trace caches vulnerable to compulsory and capacity misses. A compulsory miss problem occurs because traces are "learned" from observing previous dynamic program behavior. If a given dynamic trace has not been observed before, the trace cache will not be able to provide the trace. The learning time for traces is longer than for conventional instruction caches. Furthermore, there can be a number of unique traces as different paths are followed through a piece of code. Each static instruction may occur in several different dynamic sequences. Consequently the working set size of traces is larger than the comparable static representation. This can cause capacity misses and exacerbate the compulsory miss problem. It also reduces the robustness of trace caches to varying workloads and environments.

Instruction prefetching is a common remedy for capacity and compulsory misses in conventional instruction caches [12][14][15]. When applying the concept of prefetching to trace caches, the dynamic aspect of traces presents a number of obstacles. First, trace caches are not part of a true memory hierarchy, as there is no base level that contains all possible traces. Therefore the term "prefetching" is not entirely accurate, as there is nowhere from which to fetch complete traces. We use the term trace *preconstruction* because traces need to be constructed from static instructions, in advance of when they are needed.

Second, predicting the composition of future traces is a difficult problem. Traces are defined by their starting instruction and the outcomes of branches within the trace. To be effective, the preconstruction mechanism must identify a future point in the program that the processor will reach, and then identify the most likely dynamic paths that will pass through that point. A critical sub-problem is that the preconstruction mechanism must identify the trace alignment along each future path. Two traces are *aligned* if one terminates exactly where the next begins. For a single path through a region of code there are many possible sequences of traces that can be identified, depending on where the first trace starts. If the trace starting points identified by the preconstruction mechanism do not match the starting points needed by the processor, the preconstruction effort will have been wasted.

Third, is the issue of timeliness. The preconstruction mechanism must stay sufficiently ahead of the processor to accommodate the high latency of constructing traces. The preconstruction mechanism must be responsive to the processor "catching up" to it; i.e., knowing when to give up on a region of the program and move farther ahead of the processor. The preconstruction mechanism must also avoid getting too far ahead of the processor;

the preconstruction mechanism should not tie up resources with traces that will not be needed in the near future.

The primary objective of this paper is to propose and evaluate a trace preconstruction method. The trace preconstruction mechanism observes the processor's instruction dispatch stream to detect opportunities for jumping ahead of the processor. After doing so, the preconstruction mechanism fetches static instructions from the predicted future region of the program, and constructs a set of traces in advance of when they are needed. In this paper we propose the concept of trace preconstruction in the context of a trace processor. Trace preconstruction is equally applicable to more conventional superscalar processors that use a trace cache. In order to evaluate the potential of preconstruction a specific microarchitecture is modeled.

A secondary objective of this paper is to place the trace preconstruction in the context of an extended pipeline model for high performance processing. Trace preconstruction is a good complement to trace preprocessing [4][8]. We propose an integrated, high performance front-end that combines trace preconstruction and preprocessing. We evaluate this extended pipeline model and show that the overall performance improvement is greater than the sum of the parts.

In the next section, we describe the general trace preconstruction method to be used. Then in section 3, we discuss an implementation that we propose. This implementation includes the basics and some performance optimizations. In Section 4 we explain our simulation and methodology. In Section 5 we quantify the benefits of incorporating preconstruction, based on the performance of the SPECint95 benchmarks. Finally, in Section 6, we show how prepreprocessing fits into an extended pipeline model that is enabled by the trace cache. A model that is also used for another trace specific optimization, preprocessing. We show that preconstruction and preprocessing are complementary, and together produce a speedup greater than the sum of the individual contributions.

2. Trace Preconstruction

As stated above, we perform preconstruction in the context of a trace processor model [11]. The main components of the trace processor frontend are the next-trace predictor [7] and the trace cache [10][9] (see Figure 1).

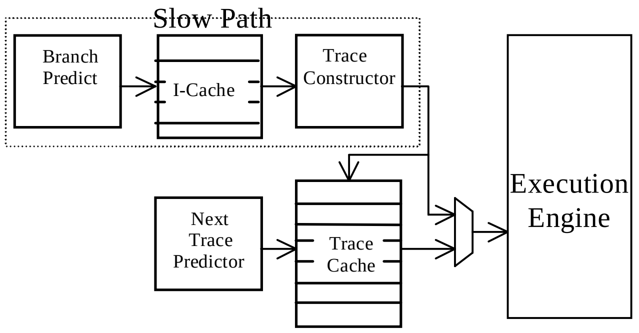

Figure 1 Trace processor frontend.

Next-trace prediction implicitly performs branch prediction and branch target prediction with sufficient bandwidth to permit the high fetch rate of the trace cache. During normal operation, the next-trace predictor and the trace cache provide a stream of instructions to the processor's execution engine. When the execution engine detects a branch misprediction, the next-trace predictor backs up and makes a new prediction. If the next-trace predictor can not generate a prediction to match the needed instructions, or if the trace cache does not have the needed trace, the *slow path* is used. The slow path uses a conventional branch predictor and instruction cache to provide instructions to the execution engine.

During periods of time that the trace cache is able to provide the correct instruction sequence to the processor, the slow path hardware is idle (including the instruction cache). This provides an opportunity to fetch instructions from the instruction cache and preconstruct valid traces that may be useful in the future.

2.1 Preconstruction Method

The overall preconstruction method scans the dynamic instruction stream and identifies *region start points*. For preconstruction to be successful, the region start points must identify instructions that the actual execution path will reach in the future. This requires start points that are many instructions ahead of the currently executed instructions. To "leap ahead" in the instruction stream, our preconstruction method uses a heuristic based on two common program constructs: loops and procedures. When a loop back edge or a procedure call is observed, the preconstruction mechanism assumes that the code after the loop exit or procedure return point will be reached in the near future.

Given a region start point, the preconstruction mechanism begins traversing a "dynamic execution tree" -- essentially a series of dynamic paths beginning at the region start point. Traces are preconstructed and placed in a buffer during the traversal of the dynamic execution tree. This process is best described via an example.

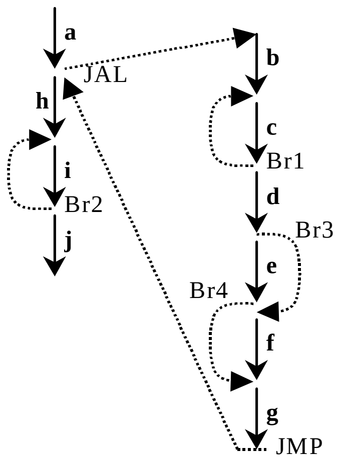

Figure 2 Static representation of example code.

Figure 2 illustrates a static piece of code as a directed graph. Arcs are basic blocks (or transfers of control) and lower case letters label the basic blocks. The static program segment begins with block *a*; then there is a procedure call via a Jump and Link (JAL) instruction. The called procedure executes block *b*, then loops through block *c* a number of times and finishes with an if-then-else construct which contains blocks *d, e, f* and *g*. Then there is a jump (JMP) back to the calling routine. Subsequently, block h is executed, there is a loop of *i* blocks, and, finally, block *j*.

The operation of the trace preconstruction method is shown in Figure 3. The bold line from left to right illustrates the actual dynamic flow of instructions. The bold line is divided into traces, labeled with the basic blocks they contain. The JAL procedure call points to a start point for preconstruction. The start point is the instruction immediately following the JAL; eventually, dynamic execution will reach this point.

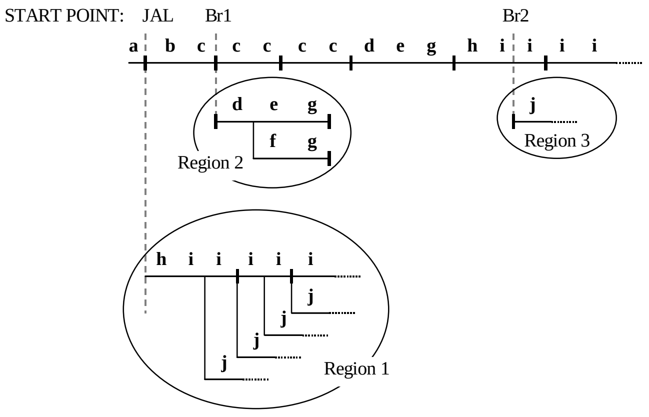

Figure 3 Dynamic representation of example code.

This region start point is pushed onto a "start point stack." As the dynamic execution proceeds, other region start points may be pushed onto the start point stack. This stack is basically a priority device -- details are given in Section 3. When the preconstruction process is ready to begin a new region, it takes the start point at the top of the region start point stack. In our example this will be the return point following the JAL, and the region to be explored is labeled "Region 1".

The preconstruction process follows a breadth first approach to constructing traces within a region. The basic algorithm we implement for traversing paths is based on identifying where traces may potentially start, *trace start points*. Note that trace start points may be different from region start points. When preconstruction for a region begins, the region start point is the first trace start point. While traversing the region, additional trace start points are generated. A small worklist of trace start points is maintained and acts as the primary director of preconstruction. When a trace start point is identified the preconstruction process generates a number of valid traces that originate from that one point. When a valid trace is completed, the instruction following the trace is identified as a new potential trace start point and is placed in the worklist. In Region 1 of our example, the preconstruction process will first identify the first instruction of the region as a trace start point and construct traces <h,i,i> and <h,i,j>. This will produce two new trace start points, one that begins with block i and one that begins after block j. The preconstruction process will then attempt to construct traces beginning from each of these points.

The preconstruction effort for a region will terminate if the processor reaches the region of code (catches up). The preconstruction effort for a region may also terminate when it reaches a resource bound. These bounds are a feature of the implementation, and are described in Section 3. Briefly, the resource limitations are a fixed number of trace preconstruction buffers (Section 3.1) and a fixed number of static instructions that may be fetched from a given region (Section 3.4.1). The preconstruction effort may also be bounded by reaching jump instructions for which the target cannot be resolved.

Returning to the example, as the dynamic execution proceeds, the loop closing branch Br1 denotes another region start point, and another set of potential traces are preconstructed on the fall-through path (shown in Region 2). When Br1 is again encountered, the algorithm will detect that it is already being processed so preconstruction is not re-initiated for this start point. As the actual execution proceeds further, the trace containing basic blocks <d,e,g> is eventually encountered, and this trace has already been preconstructed in Region 2. Similarly, <h,i,i> and <i, i> will have been preconstructed in Region 1 before they are reached.

There is potentially a very large number of paths in any of the preconstruction regions. To reduce this number, we use a heuristic that follows highly-biased branches only through their dominant direction. This can be done by using state in the slow-path dynamic branch predictor. We assume a bimodal branch predictor (table of 2-bit saturating counters indexed by branch address [13]). During preconstruction, the predictor is referenced for each forward branch. If the branch is strongly taken (or strongly not taken) only the strongly biased path is followed during preconstruction.

There may also be a number of traces that are preconstructed, but not used. For example, trace <d,f,g> from Region 2 is not used (at least in the portion of the example shown). There may also be overlap among regions. That is, the regions may contain identical traces, and this could lead to redundant trace preconstruction effort. Our trace algorithm terminates preconstruction at jump indirect instructions (the target is unknown) Consequently, this often avoids overlap. In the example, overlap between Regions 1 and 2 is avoided in this manner. On the other hand, Regions 1 and 3 do overlap in the example. In this case, redundant work may be performed. An effort could be made to avoid this redundant work, but our studies have shown the penalty to be small.

2.2 Trace Alignment

In order for a preconstructed trace to be useful, it must "align" with the actual execution path. In the example of Figure 3, this is achieved when the preconsructed trace <d,e,g> aligns with the c loop exit. To enhance the probability of correct alignment, we implement some heuristics to guide trace selection.

The trace processor uses a trace selection heuristic that forces traces to end at return instructions [11], so the first trace of a region following a return will start at the first instruction. Consequently alignment will naturally occur for traces that begin at return points. Trace alignment is more complicated for regions starting at a loop exit points. When the loop exits there may be a trace that contains some instructions from the last iteration of the loop and some instructions from beyond the exit of the loop. In this case, the trace ends at an arbitrary point, and the chances of correct alignment with a preconstructed trace are small. One solution is to force the trace to end at the loop exit, but this would lead to shorter traces than necessary. A compromise solution is to force traces to end at some even multiple of instructions beyond a loop exit [6]. For our later simulations we use the heuristic of stopping a multiple of four instructions beyond a backward branch for both the base trace processor and the trace processor with preconstruction. This heuristic also limits the overall number of unique traces, helping the compulsory and capacity miss problems of the trace cache.

3. Implementing Trace Preconstruction

The previous section outlines the overall method to be followed when preconstructing traces. We now turn to an implementation of the trace preconstruction method. This description includes the required hardware structures for implementing the basic algorithm plus some performance optimizations. More implementation detail can be found in the PhD thesis [6].

Figure 4 shows the full processor implementation. The main hardware feature of trace preconstruction is that the trace cache is supplemented with trace preconstruction buffers. Otherwise, trace preconstruction can be implemented by adding additional control and bookkeeping logic to the trace construction unit and making use of the slow-path hardware when it is idle. The additional hardware includes logic to monitor the instruction stream of the processor and a small stack to record region startpoint events.

To further optimize the performance of trace preconstruction, additional hardware can be incorporated into the microarchitecture. This additional hardware consists of extra trace constructor units (shown in Figure 4) that allow multiple traces to be constructed in parallel. Then, to extend the instruction cache's bandwidth, a set of small prefetch caches are added to service the multiple constructor units. The benefit of this extra hardware can be substantial, and our performance results in section 5 use these performance enhancements.

3.1 Preconstruction Buffers

When prefetching into conventional instruction caches, it is common to use prefetch buffers [12][14][15]. The prefetch buffers and cache are accessed in parallel. If the cache misses, but the line is in the prefetch buffer, then it is copied into the cache. Using prefetch buffers in this way avoids polluting the instruction cache whenever prefetched instructions are not actually used. Similarly, our design includes a set of preconstruction buffers to hold preconstructed traces until they are used (or discarded). At the time it is created, a preconstructed trace is allocated a preconstruction buffer. The preconstruction buffers are accessed in parallel with the trace cache. If a trace is found in a preconstruction buffer, then it is copied into the trace cache.

An important optimization is to avoid redundancy between the trace cache and the preconstruction buffers. After a trace is copied from a preconstruction buffer to the trace cache, the buffer is invalidated. Furthermore, before a trace is assigned to a preconstruction buffer, the trace cache is first checked to see if the trace is already present. In our proposed implementation, the preconstruction buffers are arranged as a 2-way set associative structure indexed by hashing the starting address of the trace with the branch outcomes of a trace [6]. This is the same general organization as the primary trace cache. Each trace in a preconstruction buffer corresponds to the preconstruction region (either current or past) from which it was originally formed. The replacement policy for the preconstruction buffers is based on the relative priorities of the corresponding preconstruction regions. Active regions have priority over past regions. The more recent the active region, the higher its relative priority. A trace generated for a region will not displace an existing trace from the same region. Consequently, the availability of preconstruction buffers is the primary implementation feature that bounds the preconstruction process within a region.

3.2 Identifying Start Points: Start Point Stack

As described in Section 2, the preconstruction process relies on two common, easily identifiable constructs for initiating trace preconstruction: procedure calls and loop terminations [1][6]. These constructs delineate region start points, and it is beneficial to prioritize them for trace preconstruction in newest-first order. Because of loop and subroutine nesting, this priority will tend to preconstruct regions more likely to be encountered sooner.

Consequently, potential region start points are maintained in a small hardware stack. We have found a stack of depth 16 works well. In order to stay ahead of the processor, the dispatch instruction stream, including speculative instructions, is observed. Start points are pushed onto the stack when a call or backward branch is observed in the dispatch stream. When the stack fills, the oldest entry on the stack is discarded to make room for newer entries. To avoid redundancy, a new start point is not pushed if it corresponds to the same region as the current top of the stack (as happened in the example of section 2). The retirement stream of the processor is observed to determine when a start point should be removed. Start points are removed from the stack if they correspond to misspeculation or when the processor's execution has reached the region to which they correspond.

To avoid redundant work, the preconstruction mechanism remembers the most recent regions for which preconstruction has completed, and preconstruction is not performed for these start points. These regions are held in extra entries in the start point stack. A few entries (four in our implementation) are reserved for this purpose.

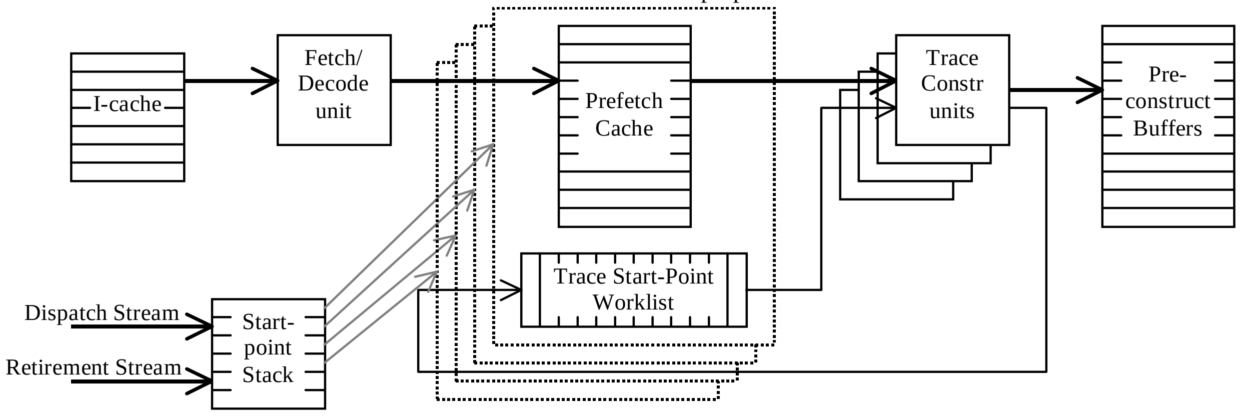

Figure 4 Trace preconstruction hardware.

3.3 Optimizations

We now describe two complementary hardware structures that work together to optimize the preconstruction process. The first optimization decouples the instruction fetch and trace construction operations with a small buffer called a prefetch cache. The second optimization incorporates parallel trace constructors to increase the bandwidth at which traces can be constructed.

3.3.1 Prefetch Caches

The same static instructions are often used in many traces. And, fetching a block of instructions from the instruction cache and decoding them will likely require a number of cycles. Consequently, it is inefficient to always fetch instructions from the instruction cache. In our implementation we incorporate special prefetch caches that can hold 256 instructions. Instructions that are fetched as part of a preconstruction region are placed into one of these prefetch caches. In our implementation we include four prefetch caches that service the parallel constructor units. The caches are assumed to be fully associative in our simulations, and they are allowed to "fill up". That is, we don't replace lines; when the cache is full, preconstruction from its associated region is terminated. In general, a lower associativity cache could likely be used with similar results.

3.3.2 Parallel Trace Construction

By incorporating multiple prefetch caches, the instruction fetch bandwidth from a single instruction cache port is sufficient to support the parallel construction of multiple traces. Our implementation makes use of this by incorporating multiple (four) trace construction units. Each trace constructor follows the algorithm to be discussed in the next section; each working on a different start point (from the same or different region).

3.4 Constructing Traces

In our proposed implementation there are four prefetch caches, each holding one region at a time. Each prefetch cache has a small worklist for maintaining trace start points belonging to its region. The four parallel trace constructors can operate on any of the four regions in a time-multiplexed fashion.

As soon as preconstruction for a region is complete and a prefetch cache/worklist is freed up, the worklist is initialized with the starting address of the highest priority region from the top of the region start point stack. This address is the first trace start point of a new region. Then, each time a trace constructor completes work with a given trace start point, it takes a new trace start point from the highest priority worklist. The trace constructor will then attempt to construct traces beginning at the start point. The needed instructions are fetched, and decoded to identify jump and branch instructions. When a strongly-biased conditional branch instruction is identified, the biased path is followed. If the branch is not strongly-biased, the constructor initially follows the not-taken path and pushes the decision point onto a small internal stack. After generating a trace, the trace constructor will pop the last decision point from its internal hardware stack and backup to start generating the alternative trace. Finally, as each new trace is constructed, a new trace start point is identified and pushed on the worklist.

3.5 Hardware Complexity

Trace preconstruction takes advantage of the slow path hardware, when it would otherwise be idle. The most costly hardware -- the instruction cache and branch predictor -- are completely shared. The trace cache hardware is effectively partitioned into two comparable components, the primary trace cache and preconstruction buffers. In theory a single trace cache could be used by simply reserving some entries for preconstruction. Our simulation results compare the performance of a trace cache and separate prefetch buffers with a larger trace cache containing the same total area. The results show that reducing the trace cache size to support preconstruction buffers is a very attractive tradeoff.

The extra logic and hardware mechanism to support the control logic for preconstruction are relatively minor. Furthermore, the preconstruction hardware is decoupled from the main processor core. Consequently it does not add to critical paths in the processor core, and the logic is also isolated for the purpose of design verification.

To optimize the performance of trace preconstruction we incorporate additional hardware in the form of prefetch buffers and extra trace constructors. Compared to the size of the trace cache, the size of the prefetch caches is small. E.g., the combined size of the prefetch caches is 1/16th the size of the trace cache. The hardware for the trace constructors is relatively simple and requires minimal area.

4. Simulation Methodology

4.1 Simulator

Simulation is performed with a detailed execution-driven simulator that models a trace processor with a distributed backend, based on the design proposed in [11]. It is composed of a number of processing units each with a register file, instruction window and execution units. Synchronization of register communication between traces is implemented through the global renaming of registers. Synchronizing of dependences through memory is enforced by special hardware [3] in the memory subsystem.

The trace processor has 4 processing elements each with a window of 16 instructions (one trace length) for a total window size of 64 instructions. The processor has 2-way issue per processing element, for a total issue width of 8. For the data memory subsystem, we model realistic level-one caches and a perfect level-two cache. We model a four ported level-one data cache of which any single processing element can only access two ports per cycle. The data cache is non-blocking and is write-back. Both the data cache and instruction cache have 64 byte lines, are 4-way set associative and have a total size of 64Kbytes. The data cache has two cycle hit latency, the instruction cache has a one cycle hit latency and the level-two cache has ten cycle hit latency.

The simulator executes the Simplescalar instruction set [2]. The latency of each operation is equivalent to the latency of the corresponding operation in the MIPS R10000 processor. Each processing element has full bypasses internally and can support back-to-back dependent operations. For communicating register values between processing elements there are global result busses. There are 8 total global result busses. It takes a full cycle for global results to be broadcast on the result bus. If an instruction is executed in one processing element in cycle *N* the result can be broadcast in cycle *N+1* and dependent operations can be executed in other processing elements in cycle *N+2*.

Traces have a maximum length of 16 instructions. We vary the size of the trace cache from 64 entries up to 1024 entries (4 Kbytes to 64 Kbytes). The trace cache is 2-way set associative and uses LRU replacement. We vary the size of the preconstruction buffer from 32 entries up to 256 entries (2 Kbytes to 16 Kbytes). The buffer is also 2-way set associative. The preconstruction hardware corresponds to the description in Section 3. There are four prefetch caches (each holding 256 instructions) and four trace constructors available for preconstruction. A region start-point stack of depth 16 is used to keep track of potential preconstruction opportunities.

4.2 Benchmarks

We use all SPECint95 benchmarks for our studies. The training input sets are used for all the benchmarks and each benchmark is run for the first 200 million instructions. The benchmarks are compiled with the Simplescalar compiler, which is a derivative of gcc-2.6.3. The benchmarks *gcc* and *go* have the largest instruction working sets of the SPEC95 benchmarks and therefore stress the trace cache the most. Many of the benchmarks have such small working sets that even very small trace caches perform well, and there is little room for improvement.

5. Performance

5.1 Impact on Trace Cache Performance

The reduction in trace cache misses is a good first-cut metric of preconstruction performance. Figure 5 gives the trace cache miss rates, in the units of misses per 1000 instructions for a variety of trace cache and preconstruction configurations for the SPECint95 benchmarks. The graphs present the miss rate as a function of the combined size of the trace cache and the preconstruction buffer. The trace cache size varies over a range of 64 to 1K entries for the larger benchmarks and 64 to 256 entries for the smaller benchmarks. In section 5.3 the performance implications of reducing the trace miss rate are discussed.

The largest benchmarks, *gcc* and *go*, both see significant benefit from trace preconstruction. For a given trace cache size, there is a 30% to 40% decrease in miss rate for the smallest preconstruction configuration and a 45% to 50% decrease in miss rate for the largest preconstruction configuration. The benefit from preconstruction is noticeably more significant than allocating comparable area to the trace cache. This is most pronounced for *go*, where the benefit from increasing the trace cache size rapidly diminishes. For comparable area, the best preconstruction configurations offer approximately 30% to 40% lower miss rates for both benchmarks.

The benchmark *gcc* sees the most benefit from incorporating a small preconstruction buffer and allotting most of the area to the trace cache. On the other hand, the benchmark *go* sees the most benefit from a relatively large preconstruction buffer. Because of this behavior either a compromise has to be made, or a design that dynamically allocates space for the preconstruction buffer may need to be used. We do not investigate dynamically partitioning space between the trace cache and preconstruction buffer, but this could likely be done.

Two of the benchmarks, *compress* and *ijpeg* have such small working sets that the even a very small trace cache performs very well and there is little opportunity to improve. The other benchmarks, *lisp*, *m88ksim*, *perl* and *vortex*, have larger working sets that limit the performance of a trace cache. The benchmarks *lisp*, *m88ksim* and *perl* show notable benefits with preconstruction. The benchmark *vortex* strains the trace cache almost as much as *gcc* or *go*. Preconstruction works extremely well for *vortex*, reducing the miss rate by 80%.

5.2 Impact on instruction cache performance

By increasing the number of trace cache hits, the number of instructions that need to be supplied from the instruction cache is reduced. Table 1 shows the number of instructions that are fetched from the instruction cache for the two benchmarks *gcc* and *go* with and without preconstruction. For both benchmarks the number of instructions supplied from the instruction cache is reduced by over 20%.

Table 1 Instructions supplied by the I-cache (per 1000 instr).

Bench -mark	512 entry trace cache	256 entry trace cache & 256 entry pre-construct buffer
gcc	233	181
go	326	213

Table 2 I-cache misses (per 1000 instructions).

Bench -mark	512 entry trace cache	256 entry trace cache & 256 entry preconstruct buffer
gcc	3.0	6.2
go	7.8	11

Table 3 Instructions supplied by I-cache misses (per 1000 instructions).

Bench -mark	512 entry trace cache	256 entry trace cache & 256 entry preconstruct buffer
gcc	10	7.1
go	35	14

The potential drawback of any prefetching scheme is an increase in memory traffic. Preconstruction requires large bandwidth from the instruction cache, but this does not interfere with other memory requests to lower levels of the memory hierarchy. Preconstruction may also increase the number of instruction cache misses that are issued to lower-levels of memory. These instruction cache misses will compete with other memory requests, so quantifying the increase is important. Table 2 gives the instruction cache miss rates with and without preconstruction. For the benchmarks *gcc* and *go*, preconstruction approximately doubles the number of instruction cache misses. But, the absolute number of misses is small, so the overall effect is not significant.

Preconstruction increases the total number of instruction cache misses, but it reduces the number of instruction cache misses observed by the slow-path. Table 3 shows the number of instructions supplied from instruction cache misses with and without preconstruction. Part of the reduction is due to fewer instructions being supplied by the instruction cache. But, the reduction in instructions supplied from instruction cache misses is greater than the reduction in total instructions supplied from the instruction cache. This suggests that the preconstruction engine is prefetching instruction cache lines that are used by the slow-path fetch mechanism.

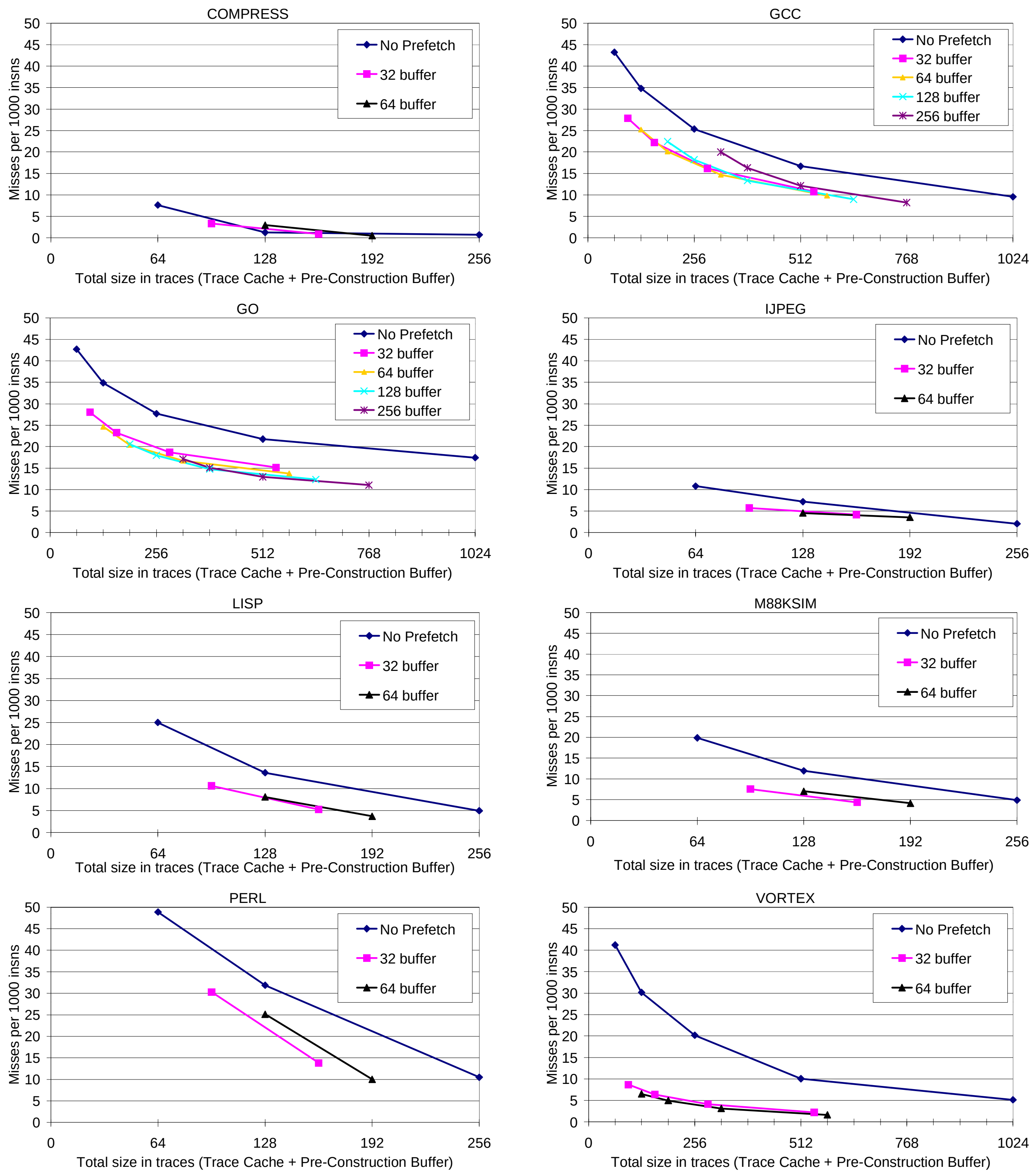

Figure 5 Trace cache performance for the SPECint95 benchmarks.

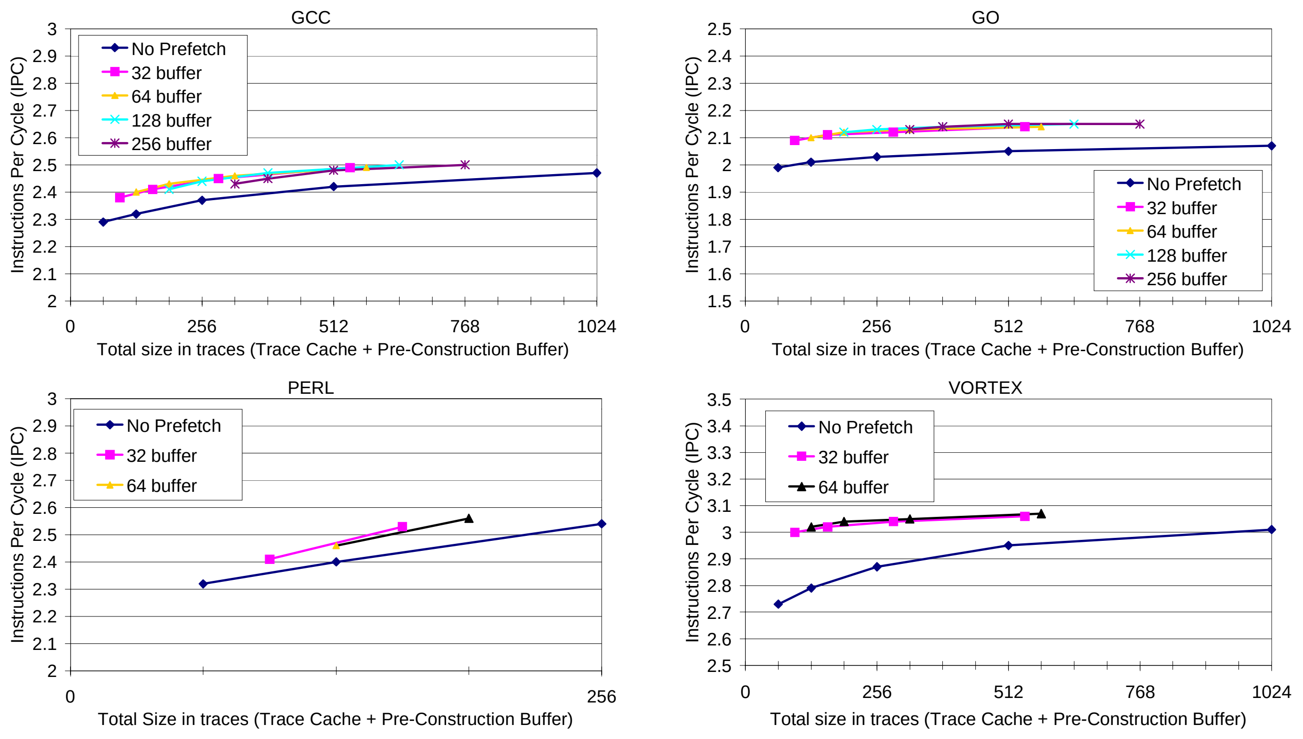

Figure 6 Performance improvements from preconstruction.

5.3 Impact on overall performance

The real metric of any optimization is how much it reduces the execution time. Figure 6 shows the performance improvements for four of the benchmarks, *gcc*, *go*, *perl* and *vortex*. The benchmarks *lisp* and *m88ksim* has similar performance benefits as *perl* and the remainder of the benchmarks see little impact from incorporating preconstruction. For the benchmarks *gcc*, *go*, *perl* and *vortex,* the performance benefit of adding preconstruction is between 3% and 10%. The benefit of preconstruction is more pronounced when combined with other optimizations that increase the execution engines throughput, as is seen in section 6.

6. Extended Pipeline Model

A traditional processor microarchitecture consists of the frontend (instruction fetch pipeline) and backend (execution pipeline). The instruction window in an out-of-order superscalar processor decouples these pipelines. Using a trace cache enables an extended pipeline organization (see Figure 7). The extended pipeline model contains a new *preprocessing pipeline*, distinct from the fetch and execute pipelines. The preprocessing pipeline works on instructions before they are fed into the normal processing phases. The trace cache decouples this preprocessing engine from the traditional processor core.

The primary source of the performance improvement is the reduction in trace cache misses. By reducing the trace cache miss rate, preconstruction increases the peak rate at which instructions can be fetched into the instruction window. The focus is the peak fetch bandwidth, not the average fetch bandwidth. The average instruction fetch rate can not be higher than the number of instructions retired per cycle, which is less than a basic block per cycle. Trace caches (and preconstruction) helps performance by filling the instruction window quickly, to expose potentially independent instructions, when the window is nearly empty. A nearly empty instruction window is most commonly caused by control mispredictions, which force a significant part of the window to be flushed.

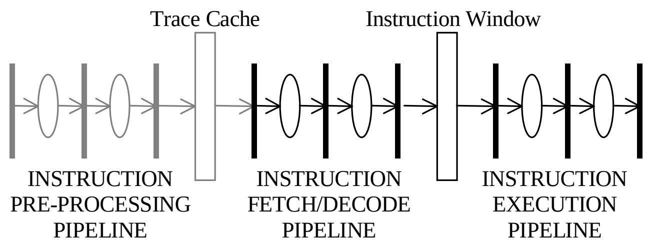
Trace Cache
Instruction Window
INSTRUCTION PRE-PROCESSING PIPELINE
INSTRUCTION FETCH/DECODE PIPELINE
INSTRUCTION EXECUTION PIPELINE

Figure 7 Extended pipeline model.

We now place trace preconstruction hardware in the larger context of the extended pipeline model. Only when integrated into an overall microarchitecture is it possible to realize the full performance potential of a number of trace cache optimizations. In particular, we will integrate trace preconstruction with two other trace-oriented optimizations.

1. An accurate control flow predictor capable of predicting multiple branch instructions per cycle; We use a path-based next trace predictor [7] that treats traces as basic units of prediction and explicitly predicts sequences of traces. The predictor collects histories of trace sequences (paths) and makes predictions based on these histories. The basic predictor is enhanced to a hybrid configuration that reduces performance losses due to cold starts and aliasing in the prediction table. The Return History Stack was introduced in [7] to increase predictor performance by saving path history information across procedure call/returns.

2. A trace preprocessing mechanism [4][8]; The trace cache enables a new class of hardware optimizations that transform the instructions within traces to increase the performance of the processor's execution engine. Traces are preprocessed for both optimizing common dynamic instruction sequences and to utilize implementation-specific execution resources. Three specific optimizations are implemented: instruction scheduling, constant propagation and targeting a new ALU. The new ALU adds two register operands, each of which can be shifted left by a small immediate amount, and a third immediate operand. Refer to [4] or [8] for more details.

Of particular interest is the combination of trace-based mechanisms. Trace prediction and preconstruction attempt to increase the instruction supply (frontend) bandwidth while preprocessing attempts to increase the instruction execution (backend) bandwidth. The frontend and backend mechanisms can be incorporated independently and will not interfere with each other. However, if only the backend is improved, the frontend may be a bottleneck, and vice versa. Only if both are simultaneously improved can their full potential be realized. In other words, the performance improvement of the combination may be greater than the sum of the parts; our results to follow show that this is indeed the case.

The extended pipeline organization takes advantage of two characteristics of traces. First, a valid trace can be placed into the trace cache at any time, independently of what the rest of the processor is doing. Second, the instructions within a trace need not be identical to the instructions specified in the static program representation, just functionally equivalent. The first characteristic is important for implementing preconstruction, while the second is exploited for implementing instruction preprocessing.

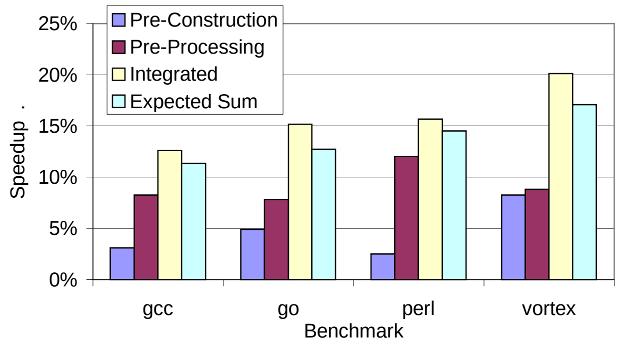

Figure 8 Speedup from extended pipeline model.

Figure 8 shows the speedup from preconstruction and preprocessing independently and together for the four benchmarks *gcc, go, perl* and *vortex*. Four bars are shown for each benchmark: 1) the speedup from preconstruction, 2) the speedup from preprocessing, 3) the speedup from combining the mechanisms and 4) the sum of the individual speedups for reference. The preconstruction results compare a processor with a 256 entry trace cache to a processor with a 128 entry trace cache and a 128 entry preconstruction buffer. The speedup from preconstruction is in the range of 2% to 8%. Preprocessing leads to a more substantial speedup, in the range of 8% to 12%. The speedup from combining the two mechanisms is greater than the sum of the two optimizations alone, in the range of 12% to 20%. This demonstrates the complementary nature of these optimizations. This also demonstrates that the potential benefit from preconstruction can be larger than the results seen in the last section if the processor execution engine has sufficient throughput to utilize the extra fetch bandwidth.

7. Conclusions

We have proposed the general concept of trace preconstruction, as well as a specific implementation. Trace preconstruction augments the trace cache by performing a function analogous to prefetching. The preconstruction mechanism sequences ahead of the processor and constructs potentially useful traces from the static program representation. Preconstruction addresses the weakness of the trace cache to compulsory and capacity misses caused by the dynamic nature of traces. Trace preconstruction takes advantage an extended pipeline organization that is enabled by the trace cache, and which decouples the preconstruction mechanism from the main processor core.

Our implementation of trace preconstruction can reduce the trace cache miss rates from 30% to 80% for the SPECint95 benchmarks with large working set sizes (*gcc, go* and *vortex*). By reducing trace cache misses, preconstruction produces a 3% to 10% overall performance improvement for these benchmarks. We believe that preconstruction is necessary to enable the trace cache to scale to large real world applications that are often much larger than the SPEC benchmarks and stress the instruction fetch mechanism much more.

When preconstruction is combined in conjunction with preprocessing, another trace specific optimization, an overall speedup of 14% is seen for the SPECint95 benchmarks. The speedup is greater than the sum of the individual speedups of preconstruction and preprocessing. With the introduction of preconstruction, preprocessing and other optimizations that take advantage of the trace cache, the trace cache becomes a more compelling microarchitectural feature.

8. ACKNOWLEDGMENTS

This work was supported in part by NSF Grant MIP-9505853 and the U.S. Army Intelligence Center and Fort Huachuca under Contract DABT63-95-C-0127 and ARPA order no. D346. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsement, either expressed or implied, of the U.S. Army Intelligence Center and For Huachuca, or the U.S. Government.

Industrial support was provided by an IBM Partnership Award, Sun Microsystems, and Intel Corporation.

The authors would like to acknowledge Eric Rotenberg for his valuable input with respect to this work. Eric Rotenberg also helped in developing the simulation infrastructure used for this research.

9. REFERENCES

[1] H. Akkary, M.Driscoll, "A Dynamic Multithreading Processor," in *Proceedings of the 31st International Symposium on Microarchitecture*, Nov. 1998.

[2] D. Burger, T. Austin and S. Bennett, "Evaluating Future Microprocessors: The SimpleScalar Tool Set," University of Wisconsin - Madison Technical Report #1308, July 1996.

[3] M. Franklin, G. S. Sohi, "ARB: A Hardware Mechanism for Dynamic Memory Disambiguation," *IEEE. Transactions on Computing*, pp. 552-571, Feb. 1996.

[4] D. Friendly, S. Patel, Y. Patt, "Putting the Fill Unit to Work: Dynamic Optimizations for Trace Cache Microprocessors," in *Proceedings of the 31st International Symposium on Microarchitecture*, Nov. 1998.

[5] Anoop Gupta, John Hennessy, Kourosh Gharachorloo, Todd Mowry, and Wolf-Dietrich Weber, "Comparative Evaluation of Latency Reducing and Tolerating Techniques," in *Proceedings of the 18th International Symposium on Computer Architecture*, pp. 254-263, May 1991.

[6] Q. Jacobson, "High-Performance Frontends for Trace Processors," Ph.D. Thesis, Department of Electrical & Computer Engineering, University of Wisconsin – Madison, Aug. 1999.

[7] Q. Jacobson, E. Rotenberg, J. E. Smith, "Path-Based Next Trace Prediction," *Proceedings of the 30th International Symposium on Microarchitecture*, pp. 14-23, Dec. 1997.

[8] Q. Jacobson, J. E. Smith, "Instruction Pre-Processing in Trace Processors," in *Proceedings. Of the 5th International Symposium on High Performance Computer Architecture*, Jan 1999.

[9] S. Patel, D. Friendly and Y. Patt, "Critical Issues Regarding the Trace Cache Fetch Mechanism." University of Michigan Technical Report CSE-TR-335-97, 1997.

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

[11] E. Rotenberg, Q. Jacobson, Y. Sazeides and J. E. Smith, "Trace Processors," *Proceedings. of the 30th International Symposium on Microarchitecture*, pp. 138-148, Dec. 1997.

[12] A. J. Smith, "Sequential Program Prefetching in Memory Hierarchies," *IEEE Computer* 11 (12), pp. 7-21, Dec 1978.

[13] J. E. Smith, "A Study of Branch Prediction Strategies," in *Proceedings of the 8th International Symposium on Computer Architecture*, pp. 135-148, May 1981.

[14] J. E. Smith, W.-C. Hsu, "Prefetching in Supercomputer Instruction Caches," In *proceedings of Supercomputing92*, pp. 588-597, 1992.

[15] C. Young, E. Shekita, "An Intelligent I-Cache Prefetch Mechanism," in *Proceedings of the International Conference on Computer Design*, pp. 44-49, Oct 1993.