In Proceedings of the International Symposium on Code Generation and Optimization (CGO 2006).

A Self-Repairing Prefetcher in an Event-Driven Dynamic Optimization Framework

Weifeng Zhang Brad Calder Dean M. Tullsen

Department of Computer Science and Engineering
University of California, San Diego

Abstract

Software prefetching has been demonstrated as a powerful technique to tolerate long load latencies. However, to be effective, prefetching must target the most critical (frequently missing) loads, and prefetch them sufficiently far in advance. This is difficult to do correctly with a static optimizer, because locality characteristics and cache latencies vary across data inputs and across different machines.

This paper presents a mechanism that dynamically inserts prefetch instructions into frequently executed hot traces. Hot traces are dynamically analyzed to identify delinquent loads and the appropriate prefetch distance for those loads. Those prefetches are then inserted into the hot trace. The low overhead of the event-driven dynamic optimization system allows the optimizer to continuously monitor the performance of the software prefetches. This is done to find an accurate and stable prefetch distance and to adapt to changes in program behavior using what we call Self-Repairing prefetching. Relative to the baseline hardware stride prefetching, we find a total 23% improvement when we use the self-repairing mechanism to adaptively discover the best prefetch distance for each load, which is 12% better performance than dynamic prefetching techniques without adaptive repairing.

1 Introduction

The performance of modern processors is increasingly dominated by the widening latency gap between processors and memory subsystems. One way to bridge the latency gap is by prefetching load values into the cache, which attempts to overlap the memory stalls with the execution of other useful instructions in the same program. This decreases the observed latency, increases memory level parallelism, and allows cache-hit dominated performance even when the working set is larger than the cache. Software based prefetching [4, 18, 15, 31, 14, 6, 11, 20] has been shown to be a promising technique to address this issue, and all modern high-performance instruction set architectures provide support for software prefetching.

In order for software prefetching to be effective, prefetches should be accurate (i.e., targeting the loads that are actually missing the cache) and timely (far enough ahead of time to fully cover the miss latency). Static compilers do not always do this well, even with offline profiling, because which loads are critical, and the average latencies of those loads, will vary across different data inputs. Additionally, if the code runs on multiple machines, the correct software prefetching distance for one machine will be inappropriate for the other.

In this paper, we extend the event-driven dynamic optimization framework called *Trident* [33] to dynamically permute the object code by inserting prefetch instructions. Trident exploits a hardware multithreading processor, using an otherwise idle hardware thread to concurrently optimize a thread that is running. Trident also assumes conservative hardware support to identify performance-critical events to trigger this dynamic optimization. The concurrent optimization and hardware event-driven monitoring provides an extremely low-overhead dynamic optimization system that can support very aggressive optimizations.

In the prior work, Trident [33] focused on hot trace generation and dynamic value specialization. In this paper we extend our approach to support adaptive software prefetching. We modified both the hardware support and the dynamic optimizer used by Trident to create hot traces with software prefetching. In this approach, basic instruction blocks which are frequently executed together are collected to form *hot traces*, and they are monitored by hardware to detect *delinquent loads*, which frequently miss in the data cache. Upon detection of such loads, hardware-generated *hot events* trigger the execution of a software thread to perform optimization. The thread inserts prefetch instructions into the original hot trace. This approach may cover multiple delinquent loads per hot trace.

Prefetching a delinquent load too late will prevent the prefetch from hiding the entire latency. However, prefetching too far in advance may unnecessarily displace useful data, increases the likelihood that prefetched data will be replaced, and also increases the likelihood of prefetching unneeded data due to unexpected intervening control flow. Therefore, we want to prefetch a load just in time, so its value appears in the cache right before it is needed. This is the goal behind the adaptive (self-repairing) part of our prefetcher. Our adaptive prefetcher re-evaluates the effectiveness of the inserted prefetches through hardware monitoring, so that its prefetch distance may be adjusted, or it may be removed altogether, according to the program's runtime behavior. Like hardware prefetching, this technique operates on dynamic information rather than static information to initiate prefetching, and it works on legacy code without sacrificing software compatibility with future processors.

The remainder of this paper is organized as follows. Section 2 discusses the prior research on various software and hardware prefetching techniques. Section 3 describes our dynamic software prefetching architecture. Section 4 shows our simulation methodology. Section 5 presents performance results using our prefetching techniques, and we conclude the paper in section 6.

2 Related Work

This paper is based on a large body of prior research in memory prefetching. In this section we focus on summarizing previous software and hardware prefetching techniques.

2.1 Software-based Prefetching

A large amount of research has been done on compiler-enabled software prefetching [4, 18, 16]. Luk and Mowry [16] examine pointer chain prefetching for recursive data structures (RDS). They also add *jump pointers* to prefetch heap objects farther in advance than one pointer traversal. Roth and Sohi [22] extend the jump pointer prefetching technique via a software/hardware scheme to provide various trade-offs between accuracy and prefetching overhead. Cahoon and McKinley [3] propose an effective data-flow analysis to identify RDS traversals in Java to guide greedy prefetching. Zhang and Torrellas [34] employ user-added grouping instructions that are used to group together fields/objects that should be prefetched together. These groups are stored in a hardware buffer, and any miss in the group triggers prefetches for all cache blocks in the group. Yamada, et al. [32] propose a compiler-assisted hardware technique to combine data relocation and block prefetching to improve memory performance. Saavedra and Park [24] propose an adaptive execution scheme in which the compiler inserts software prefetches and generates a *software agent* to control these prefetches at runtime. This scheme uses a single prefetching distance to control all prefetches within the whole loop body. In contrast, our technique targets true cache misses by dynamically generating software prefetches which are tuned to each individual load.

Static compiler-generated software prefetching cannot optimize for all data inputs, or for different machines. Our self-adapting software prefetching extends this prior research by applying some static prefetching techniques dynamically. Our technique focuses on enabling effective prefetching by automatically adapting to the program's runtime behavior. In addition, our technique works transparently on existing binary code.

2.2 Hardware-based Prefetching

Smith and Hsu [28] introduce tagged next-line prefetching. Jouppi [12] introduces stream buffers as a more efficient next-line prefetching architecture. The stream buffers allow multiple prefetching streams to run in parallel, and prefetch distances greater than one iteration. Palacharla and Kesseler [19] propose a non-unit stride detection scheme to enhance the effectiveness of stream buffers. This model was further extended by Farkas, et al. [9] to use a PC-based stride predictor to predict strides on a per load basis, instead of using the global miss addresses. Sherwood, et al. [27] extend the above architecture to use a stride-filtered Markov predictor to guide the prediction stream. The predictor-directed stream buffer (*PSB*) can generate the next prefetch address without a fixed stride if a Markov transition is found. Timely prefetches may be achieved by allowing the stream buffers to run independently ahead of the execution stream.

While our approach maintains much of the runtime adaptability of these hardware schemes, it can target more complex memory access behaviors, because it is based on analysis of the actual code.

2.3 Prefetching via Dynamic Optimization Systems

Inagaki, et al. [11] extend an efficient software profiling algorithm [31] to target both intra- and inter- loop stride loads in a dynamic optimization system. Their technique mainly focuses on Java compilers. They proposed a lightweight profiling technique by interpreting the Java object a few times to identify load access patterns. Compared with our approach, profiling via interpretation still imposes a high overhead. In addition, our prefetching works for general purpose and even legacy programs.

Lu, et al. [14, 5] developed a dynamic optimization system, called *ADORE*, to perform software prefetching on delinquent loads. ADORE analyzes the code in the hot trace to identify load access patterns. Loads with more complicated patterns are predicted using Wu's algorithm [31]. The prefetching distance is calculated according to the load's average miss latency. Our technique builds on that research, but it has clear distinctions from their work. First, our prefetching technique adapts the prefetch distance and repairs the hot trace instead of having to re-generate the entire hot traces. This gives us the ability, for example, to do efficient, adaptive search for the optimal prefetch distance. Second, we perform jump-pointer type prefetching and same-object prefetching.

Chilimbi and Hirzel [6] propose an automated approach to inject prefetching code into hot data streams based on the correlation of hot data reference sequences. This scheme gathers a temporal data reference profile via bursty sampling, and extracts data reference patterns frequently occurring in the same order. Prefetching is inserted dynamically at proper program points to prefetch these references. Compared with our approach, this scheme has higher runtime overhead due to software profiling, and requires static binary instrumentation.

2.4 Prefetching via Pre-execution Threads

Various methods have been proposed to pre-compute load addresses and issue prefetches early in a separate thread. Those systems use the extra thread contexts to run prefetch code without modifying the original code. In comparison, we only use the extra thread contexts to compile the hot traces and to insert prefetch instruction into these hot traces.

Pre-execution code can be constructed statically [8, 15, 13] or dynamically [7, 23]. Pre-computation code typically runs in a spare hardware context [7, 23] or a dedicated hardware engine [17, 1, 21], in parallel with the main thread.

Collins, et al. [7] present a dynamic speculative pre-computation (*DSP*) scheme to dynamically identify a small number of delinquent loads and backward-slice dependent instructions to construct (in hardware) pre-computation slices. These slices run in a spare hardware thread to prefetch these delinquent loads. A chaining prefetcher allows a single thread to loop through multiple prefetches of the same load, allowing the prefetching thread to get sufficiently ahead of the main thread. While this is a hardware technique, this work did employ some trial-and-error adaptation to create helper thread prefetchers with different assumed control flow, induction unrolling, and run-ahead count until the helper thread appears to be ”working”. Our scheme has similar spirit but dynamically patches embedded/inlined prefetching instruction bits in the memory to specify a prefetch distance. Ganusov and Burtscher [10] propose a prefetching technique, called *Future Execution*, which is similar to DSP except it uses predicted values as initial live-in values to start prefetching threads a few iterations ahead. This technique reserves an entire hardware context to run prefetching code, whereas our technique only uses another hardware context occasionally for concurrent dynamic optimization.

3 Dynamic Prefetching Architecture

In this section we describe our self-repairing adaptive software prefetching approach implemented within Trident [33].

3.1 Overview of Trident Architecture

Trident is an event-driven, multithreaded dynamic optimization framework [33]. Trident exploits modern processors' abundant on-chip parallelism and hardware performance monitoring mechanisms, to enable low overhead dynamic optimization. It provides low overhead performance monitoring via hardware event monitoring. It is low because optimization is done in a separate thread that runs concurrently with the main thread. This eliminates the need to interrupt the main thread, and the interference between the two threads is minimal. In this work, we find that this low overhead makes it possible to pursue more aggressive optimizations, and to apply a given optimization much more frequently. This allows continuous incremental improvement or even allows the system to use trial and error to apply an optimization most effectively.

An overview of the Trident architecture is shown in Figure 1. Trident includes a few generic hardware structures and corresponding software infrastructure for runtime optimization. The generic hardware structures monitor the program's behavior and generate optimization events. These structures

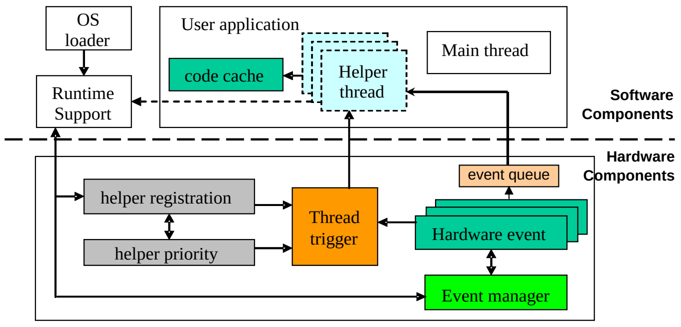

Figure 1: Trident dynamic optimization architecture

resemble the performance monitor hardware in modern processors. We introduce some minor changes to make them functionally more suitable for our technique. Runtime optimization is invoked when an interesting monitored hardware event occurs. Optimization events that were supported in the prior Trident work [33] focused on efficient generation and performance of hot traces. A hardware branch history profiler is used to identify hot traces for optimization. A hardware structure called the *watch table* is also used to monitor the performance of hot traces that are executing, in order to identify and back out of hot traces that are under-performing.

When a program is loaded for execution, the runtime support is invoked to create a generic helper thread environment to process hardware events. The helper thread is only invoked to process optimization events that occur, otherwise it does not run. A *registration structure* is also created in the program's address space. This software structure contains a pointer to the starting code of the helper thread, as well as the stack pointer, global data pointer, pointer to the code cache structure, and thread priority. This structure is used to keep the data needed to start the execution of the helper thread for a optimization event. Because we only need a pointer to the registration structure in order to initialize and start up the helper thread, it provides a fast mechanism for spawning an optimization thread, and an efficient mechanism to keep track of state across context switches and helper thread invocations [33].

3.2 The Prefetching Architecture Overview

The goal of this research is to use dynamic trace optimization to improve the performance of the memory subsystem for a thread. We use the Trident framework to generate an optimized hot trace. When the hardware monitoring structure detects delinquent load events (loads that frequently miss and have high average miss latency), optimized hot traces are re-optimized to insert software prefetching instructions. The following provides a high level overview of how Trident works and how our prefetching approach works inside of Trident:

- **Trace Formation.** Trident uses a generic branch profiler to detect hot traces. A hot trace is represented as a starting PC followed by a branch direction bitmap. When a hot trace event is triggered, the helper thread is invoked to streamline typically non-contiguous instruction blocks as indicated by the bitmap to form a trace, and perform on it several classical compiler optimizations such as redundant branch/load removal, constant propagation, instruction re-association, and strength reduction. Trident also performs a couple of specific optimizations to improve the performance of legacy code. For example, conversion between a long integer and a floating point number might be done by a pair of store/load instructions in legacy code. Trident converts this pair to a simple *MOVE* instruction (assumed added to the ISA).

- **Linking Trace.** Trident inserts the trace into a memory buffer, called the Code Cache, and patches the original binary to redirect execution to use the hot trace.
- **Monitor Trace Loads.** We add a hardware structure called the Delinquent Load Table (DLT), as shown in the Table below, to Trident to monitor the performance of loads that are executed on these hot traces. In Trident, when an instruction is committed, the hardware knows if it resides within a hot trace formed by Trident based upon Trident's hardware *watch table* [33]. We therefore update the DLT with only loads that are in hot traces. Note that the watch table also monitors a trace's minimal execution time, and we will describe its use in Section 3.5.2. The Table below shows all of the fields in the watch table and the DLT. The fields in the DLT will be described in more detail in the rest of this section.

Watch table	Trace starting PC
	Trace length
	Trace minimal execution time
	Trace optimization flag
Delinquent load table (DLT)	Load tag
	Access counter
	L1 miss counter
	Total miss latency
	Stride
	Stride confidence bits
	Last effective address
	Mature flag

- **Delinquent Load Event.** When a hot trace load misses in the memory hierarchy, we then look up the DLT and determine if it meets the criteria to be classified as a delinquent load. The criteria are that the load's miss rate is above a threshold, and the load's average memory latency for the last M misses is larger than the half of the L2 miss latency. If both of these conditions are true, then the DLT will trigger a Delinquent Load event. When a delinquent load event is triggered for a hot trace, we set a bit in the Trident watch table for that hot trace to indicate that the hot trace is currently being re-optimized. This is to prevent other re-optimization events being triggered for that hot trace while we are doing our optimization.
- **Insertion of Prefetches into Hot Traces.** A Delinquent Load event will start the execution of the helper thread described earlier (if a context is available) in Trident to run our software optimizer to perform the prefetch insertion algorithms described below.
- **Linking in the Re-Optimized Hot Trace.** Once the trace is re-optimized Trident links it into the execution by re-patching the original binary to jump to the newly formed trace, and Trident removes the old hot trace from the hardware watch table. A thread's execution will then automatically start using the new hot trace.

3.3 Delinquent Load Table

In this research, we add to Trident's hardware event monitoring a *Delinquent Load Table* (DLT), which is used to monitor loads within a hot trace. When a load commits and it is part of a hot trace (determined by the Trident watch table as described earlier), we perform a lookup and update the DLT.

In order to determine if a load is delinquent, we need to find its miss rate and average miss latency. If the load is considered delinquent, it triggers a delinquent load event. When Trident sees this event, it will start a helper thread to process the event and perform dynamic optimization to insert software prefetching. The DLT is used to identify the delinquent loads and is treated as an associative cache, tagged by the load PC, and uses the least recently used replacement policy.

To determine if a load is delinquent or not, we examine the miss rate and average miss latency after a load has been executed N times. This is called the *load monitoring window*. After N accesses, these statistics are calculated to determine if the load is delinquent, and then the counters are cleared, and the load is re-examined at the end of the next load monitoring window (after the next N accesses). The DLT keeps track of the following information for the load:

- **Access counter.** This counter keeps track of how many times this load has been accessed during a given monitoring window. At the end of the window (i.e. N accesses), it resets itself (along with other counters as described below) to start a new round of counting.
- **Miss counter and miss latency.** The miss counter keeps track of how many cache misses this load encounters during the current window. Together with the access counter, it provides an approximate miss rate within a monitoring window.

  When a load misses in the cache, the miss counter increments and its miss latency is added to the sum. At the end of the window, a load is claimed as delinquent if (1) its miss counter reaches a threshold (i.e. miss rate is above the threshold), and (2) its average miss latency is higher than half of the L2 miss latency. Here, the average miss latency is calculated as the total miss latency divided by the total miss count. The delinquent load then triggers a delinquent load event, which in turn invokes the helper thread to run. These counters and total miss latency stay unchanged and will be cleared later by the helper thread during optimization.

  If the access counter reaches its threshold before the miss counter, the load is not delinquent. At the end of the window, the access and miss counters are reset, and the monitoring of the load continues.

- **Stride address prediction.** Our software prefetching optimization focuses on taking advantage of stride predictable loads. Therefore, each DLT entry keeps track of (a) the load's last address, and (b) the load's last address stride, and (c) a 4-bit address stride confidence counter. These values are updated every time the load is committed (not just on misses). The confidence counter starts with the value of 0 and is incremented by 1 if the current stride equals the last stride, and decremented by 7 if they are different. A load is said to be stride predictable if the stride confidence counter is 15. Although in many cases the stride can be identified by analyzing the code in the hot trace, the hardware support allows us to identify a large number of pointer loads that turn out to have stride access patterns, due to the way memory structures are allocated and used. This allows effective prefetching of loads that a static software prefetcher will have great difficulty with.
- **Prefetch mature flag.** This flag is used to indicate if a load has been tuned enough times. We want to avoid generating too many delinquent load events for a load that our prefetch algorithm can not cover or hide all of the latency for. If the mature flag is set, then the load will not generate a delinquent event on a miss.

3.4 Dynamic Prefetch Optimizer

In Trident, the runtime optimizer is triggered to run as a helper thread on an idle hardware context when a delinquent load event is detected. If a prefetch instruction has not been inserted into the hot trace to prefetch this delinquent load, the prefetch optimizer will generate a new trace and insert a prefetch instruction to target this load. Otherwise, the optimizer will try to repair the prefetch instruction as described in Section 3.5. Before the optimizer finishes, it resets the hot trace's optimization flag so that it can be re-optimized in the future.

3.4.1 Delinquent load identification

During prefetch insertion, the dynamic optimizer first identifies all delinquent loads within the trace, and then partitions these loads into different types so that prefetch instructions can be inserted accordingly.

Because there are at least a few thousand cycles between when the delinquent load event is triggered and when the dynamic prefetch optimizer is ready to start its execution (if there was no contention for the spare thread), the optimizer first checks if there are other loads that need to be prefetched in the same hot trace. To identify all delinquent loads in the hot trace, we look up each of the loads in the DLT. If they satisfy the delinquent load classification described in Section 3.3, then they are added to the delinquent load list. Note if a load has not yet completed execution of a full monitoring window, its miss rate and latency are calculated using current counter values in a partial monitoring window.

After all of the delinquent loads are identified in the trace, the optimizer then classifies all of these delinquent loads as Stride, Pointer, or Same Object based upon the following criteria:

- **Stride.** For a load instruction within a loop, if the recurrence between instances of the load is a single simple arithmetic instruction (e.g. LDA, or ADD, SUB) whose arguments are a constant and the base register, then this load is classified as a stride load. This simple definition picks up most strided loads, and is sufficient because we also mark any load the DLT found stride-predictable (which picks up more complex recurrences).
- **Pointer.** If the load is not classified as Stride, then we check to see if it is a pointer load. If this load's destination register is used (before any modification) as the base register of any other load instruction, then the destination register contains a pointer value. So this load is classified as a Pointer load.
- **Same Object.** For each delinquent load classified as Stride, the optimizer examines both forward and backward on the hot trace the other loads with the same live base register. If these loads do exist, then the optimizer puts them into a group, called *Same Object* group. The end result is a set of same object groups, where each set contains at least one delinquent load that is stride predictable. Note, a delinquent load can only belong to at most one group, and a group can contain more than one delinquent load. The degenerate case is that a group can consist of only one single load, which is the stride address delinquent load.

  If we have multiple loads using the same base register which has been identified as a pointer, we also classify those loads as same object and prefetch them together. The same object classification also allows us to prefetch multiple loads with a single prefetch instruction, and eliminate redundant prefetches to the same cache line. When applying our self-repairing optimization, it allows us to also repair all the object prefetch distances as a group, rather than one at a time with separate optimization events.

Any load instruction that is not classified as one of the above types will not be prefetched in our current framework.

3.4.2 Stride-Based Prefetching of Same-Object Loads

We first focus on stride address predictable groups, because these are the loads for which we can perform timely prefetching. As long as a same object group has at least one delinquent load that is Stride predictable, then the whole group is classified as stride address predictable.

Each stride address predictable same object group is processed using the following algorithm:

- Find the minimum load offset from the base register in the group.

- Insert a stride prefetch instruction using the group's base register and the minimal offset as this format:

$$prefetch(offset + stride)(base) \quad (1)$$

- Find the delinquent load with the next smallest offset. If its offset from the prior prefetch is less than the cache line block size, simply mark this load as prefetched and skip it; otherwise, insert another stride-based prefetch as above with the non-covered delinquent load's offset. When a load is skipped, the offset plus the base register actually may put that load into the next cache block, which should have been prefetched. To address this, we prefetch one additional cache block after a skipped load. This still allows us to skip several loads, and only prefetch each block once.

This process repeats until all delinquent loads in the group are processed.

3.4.3 Prefetching for Pointer Loads

After the stride-based same object prefetching is done, the only delinquent loads we target for additional prefetching are pointer loads if they have not yet been processed in the algorithm above. For example, a pointer chasing load in a loop looks something like:

ld r1, offset(r1)

and we dereference this pointer twice by inserting the following instructions after the above instruction in the hot trace:

ld scratch, offset(r1)
prefetch offset(scratch)

These two instructions potentially prefetch the object in the next two iterations of the loop. Notice that the first instruction should be a non-faulting load.

Note that if a pointer load belongs to a Same Object group, the pointer is also dereferenced right after its stride-based prefetch instruction. We found a significant portion of pointer loads prefetchable using the stride-based same-object algorithm described above.

3.5 Prefetch Distance for Stride Address Predictable Loads

The stride prefetching insertion algorithm described above only prefetches one iteration ahead in a loop for the object. What we really want to do is determine how far ahead to prefetch an object, which is called the prefetch distance.

Prefetching can target loads which miss at different memory levels. Existing dynamic prefetching systems such as [14, 5] estimate the prefetch distance (in number of iterations) as:

$$distance = \frac{average\ load\ miss\ latency}{average\ cycles\ per\ iteration} \quad (2)$$

where, in our case, the average load miss latency for a particular load and the average number of cycles spent in the trace are calculated by sampling hardware counters. With this the stride based prefetch instruction described in statement (1) becomes:

$$prefetch(offset + (stride * distance))(base) \quad (3)$$

In this paper, we provide results for this approach, where we calculate a fixed prefetch distance for a load by using average load miss latency and the average cycles per loop iteration for a trace. Most prior prefetching systems keep the prefetch distance fixed like this after it is determined either statically or dynamically, and do not provide a mechanism to later tune this distance. A primary contribution of our paper is the ability to adapt this distance (as well as the stride) – not only allowing us to get it right more often, but also allowing us to further adapt if the nature of the load changes, which we describe next.

3.5.1 Adaptive, Self-Repairing Prefetching

The above prefetching distance estimation gives us a good starting point to initiate prefetching. The problem is that as you insert prefetches, even for the prefetched load, the recurrence time between instances of that load will change; that is, the iteration time used to calculate the prefetch distance is no longer correct. This problem is exacerbated by neighboring loads that are subsequently prefetched. Each successful optimization may expose other loads that were previously being prefetched on time.

Due to the heavy interaction between neighboring loads and the correct prefetch distance for each, we found that careful estimation of the correct distance was of little use, and a much simpler scheme provided equivalent performance.

Our Adaptive prefetching algorithm works as follows.

- All stride based prefetch instructions for delinquent loads are inserted in the hot trace as in statement (3) with the initial distance of 1.
- We continue monitoring the behavior of these loads in the DLT. If the prefetch is not hiding enough latency, the load will eventually be marked again as a delinquent load and cause another delinquent load event.
- If the delinquent load is stride predictable and there exists a prefetch instruction for it, the optimizer increases or decrements the distance stored in the instruction as outlined in the next section, and we patch the prefetch instruction in the trace. The prior distance can be back calculated by using the predicted stride and the known offset, or using book-keeping information stored along with the trace.

The above optimization is done by the helper thread. Note that the repairing is easy and fast, since we do not generate a new trace or change the layout of an existing trace. We just update the prefetch instruction bits with the new distance. This process is repeated until the prefetch distance causes the load to stop triggering delinquent load events, or the load becomes *mature*, which we describe later.

This approach works very well, especially when there are potentially multiple delinquent loads in a hot trace. Each load will have its prefetch distance adjusted until the loads in the trace are no longer delinquent. As each load is prefetched more effectively, neighboring loads that then become exposed because the code runs faster will generate another delinquent load event, and be repaired. Stabilization is achieved quickly because the repair operation is much quicker than generating a new prefetch-optimized hot trace.

We also modeled a scheme where the initial distance is set to the estimated distance from the previous section, but saw no gain because the low overhead of the optimization system allows it to converge quickly.

3.5.2 Prefetch Maturing

When a load triggers the delinquent load event for the first time, the optimizer inserts a prefetch instruction to target this delinquent load. Any subsequent delinquent load events from this load will cause its prefetch instruction to be repaired by the optimizer. Note if a delinquent load cannot be prefetched, as described in Section 3.4.1, or it cannot be repaired due to lack of stride patterns, the optimizer sets its mature flag in the DLT. So it will never cause a delinquent event, until the mature flag is cleared. For our experiments, the only way the mature flag was cleared is when a load is replaced due to capacity constraints in the table. Future work may want to examine clearing the mature flag when there is a working set or phase change in the program's execution to capture potentially new behavior [26].

For each of the repairable delinquent loads in the trace, the optimizer re-calculates its maximal prefetch distance. The maximum is the memory access latency divided by the trace's minimal execution time from the watch table. Here, the minimal execution time of the trace should represent the best possible scenario where all loads in the trace potentially hit in the cache. When executing an optimized trace, the number of cycles from when the trace is first fetched until it finishes execution represents the time to execute the trace, and the watch table keeps the minimum number of cycles seen for each trace currently being used.

When a prefetch instruction is repaired, the optimizer increases the load's prefetch distance by 1 up to its maximal distance. Increasing the prefetch distance allows the prefetch to happen further ahead of the potential use of the data, which will hopefully reduce the load miss latency. Thus we expect the average access latency for the load to decrease when we increase the prefetch distance.

However, as the prefetch distance increases, the possibility of prefetched data being replaced by data from other loads/prefetches also increases. If this occurs, the load's average access latency may instead increase. We therefore calculate the average access latency when repairing a prefetch, and when it is observed to start to increase the optimizer decrements the distance by one. To do this calculation, the average access latency is computed using the load's access counter, miss counter, and total miss latency from the DLT table. In addition, we store in an optimization buffer in program's memory the load's previous average access latency seen.

Therefore, our repairing mechanism varies a load's prefetch distance from one to its maximal distance, trying to find an optimal distance. To avoid a load being repaired too many times, the optimizer sets the load's mature flag in the DLT when the number of repairs attempted is twice as many as its maximal (distance) value. When a load is first optimized, we set a repair counter for the load to this number. Each time a load is repaired the counter is decremented. When the counter is zero, then we no longer try to repair the load, and the mature flag is set in the DLT.

Note, in order to make the above decisions, the optimizer always maintains relevant information from all delinquent loads, such as the number of repairs left, the maximal distance, and the average access latency history. This is stored in a memory buffer used by the optimizer. This information could alternatively be stored in the DLT.

4 Methodology

To evaluate the performance of our self-repairing software prefetching technique, we wrote a dynamic compiler, which runs concurrently with the applications to be optimized. The evaluation is done on a simulated SMT processor with hardware stride prefetching.

4.1 Baseline Processor Architecture

Our baseline architecture is simulated as a 20-staged simultaneous multithreading (SMT) processor [30, 29] with 2 hardware contexts. The baseline configuration is shown in Table 1.

Performance is evaluated using the SMTSIM processor simulator [30], modified to model the Trident hardware and runtime infrastructure. The simulator also models memory timing and bus occupancy among different memory hierarchies. It simulates the actual execution of the main thread, running concurrently with the optimizing helper threads as they modify the executable, place traces in the code cache, and patch the main thread to begin using the new traces. Significant care is taken to insure that instruction throughput (IPC) results correspond to only the number of instructions the *original* code would have executed.

Since modern processors often include a hardware prefetching mechanism, we implement a reasonably aggressive hardware stream buffer prefetching [27] in our baseline architecture. The stream buffers are guided by a stride predictor, and buffers are allocated using a confidence scheme. We simulate two stream buffer configurations: (1) 4 stream buffers and each buffer has 4 entries, (2) 8 stream buffers and each has 8 entries. As shown in Figure 2, the 4X4 configuration achieves an average 35% speedup relative to no prefetching, and the 8X8 configuration, 40%. We therefore choose the hardware stream buffers of the 8X8 configuration

Pipeline	20-stage, 256-entry ROB, 224 registers Two hardware contexts
Queue Sizes	64 entries each IQ, FQ, and MQ
Fetch Bandwidth	4 total instructions
Issue Bandwidth	4 instructions per cycle up to 4 Integer, 2 FP, 2 loads/stores
Branch Predictor	2bcgskew, 64K entry Meta and gshare 16K entry bimodal table
ICache size & latency	64 KB 2-way associative, 3 cycles
L1 size & latency	64 KB 2-way associative, 3 cycles
L2 size & latency	512 KB 8-way associative, 11 cycles
L3 size & latency	4 MB 16-way associative, 35 cycles
Memory Latency	350 cycles
Hardware stream buffers	8 stream buffers; each buffer 8 entries. History table 1024 entries. Prefetching is guided by a stride predictor.

Table 1: The baseline SMT processor configuration.

as our baseline, which is used to evaluate the relative performance of software prefetching in the next section.

Note that in our current study, software prefetching works independently of the underlying hardware prefetching mechanism. Because it focuses on loads that actually miss, it will naturally adapt to the loads that the hardware prefetcher cannot handle. With this system, the compiler need not know what, if any, hardware prefetcher is active.

4.2 Benchmarks

Performance is evaluated using SPEC 2000 (integer and FP) benchmarks and a few pointer intensive applications from prior research. We selected the top 14 benchmarks with the longest average miss latencies for our study. These include *applu, art, dot, equake, facerec, fma3d, galgel, gap, mcf, mgrid, parser, swim, vis,* and *wupwise*. All benchmarks are compiled on the Alpha platform (Digital Unix V4.0F) with the highest optimization options. Each benchmark is simulated for 100 million instructions beyond the single simulation points from SimPoint [25] except *dot* and *vis*, which both are fast forwarded 5 billion instructions.

The simulator is warmed up with 5 million instructions before the true simulation starts. Dynamic optimization and related structures are not enabled until after warmup is finished. 100 million instructions are simulated to demonstrate Trident's ability to quickly capture and then benefit from concurrent optimization. We expect even better performance improvement when simulating more instructions because the dynamic optimization cost and ramp-up time will be amortized, since we start simulation with no hot traces generated. However, even within this small optimization window, we found dynamically inserted prefetches still work well.

Figure 2 shows the base performance of each benchmark when executed alone on the baseline architecture. The performance from our baseline (the 8X8 hardware prefetching) is used for future performance comparison.

4.3 Prefetching via Trident Architecture

The goal of this research is to use dynamic code optimization to improve the performance of the memory subsystem. We use the event-driven multithreaded dynamic optimization

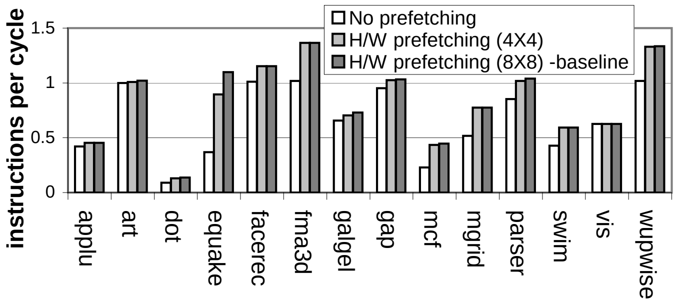

Figure 2: Performance on the baseline SMT processor

Branch profiler	256-entry, 4-way associative. Each entry has a 4-bit counter. Three standalone 16-bit bitmaps
Watch table	256-entry. Each entry monitors current trace's minimal execution time.
Delinquent Load Table	2-way associative; total 1024 entries. Access counter: 256 Miss counter threshold: 8 Each entry keeps track of the load's accesses, misses, miss latency, last address, and its stride.

Table 2: Trident hardware monitoring structures

framework (Trident) to generate base optimized hot traces. Upon delinquent load events, base-optimized hot traces are re-optimized to insert software prefetching instructions to target frequent cache-missing loads.

The runtime optimization code performs optimizations on the streamlined instruction traces. Optimizations include forming hot traces with base optimizations as outlined in Section 3.2, inserting software prefetching instructions into hot traces, and repairing prefetching as needed. The runtime code executed in Trident is written in *C* and compiled with *gcc -O5* on the Alpha platform. Special care is taken to make the runtime code thread safe. When the runtime optimizer, which runs as a helper thread in a separate processor hardware context, is triggered, we simulate the startup of the thread, with a 2000 cycle latency, and we simulate the optimizations in detail on our SMT simulator. The startup consists of executing our run-time system to initialize the helper thread's registration structure for the optimization to be performed, which sets the helper thread's starting PC, stack pointer, global data pointer, and the priority.

Monitoring Hardware Trident uses a few small hardware structures to monitor the program's execution. These hardware structures can generate hot events upon detection of certain program behaviors, and trigger Trident to perform dynamic optimization. The configurations of the major hardware structures – the branch profiler, the hot trace watch table, and the delinquent load table – are shown in Table 2. The default DLT table has the access counter threshold of 256 and the miss counter threshold of 8, which approximates a cache miss threshold of 3% for delinquent loads.

5 Results

In this section, we evaluate the costs, effectiveness, and performance of our dynamic prefetching technique. Performance improvement is relative to the baseline architecture, whose performance is shown in Figure 2.

5.1 Overhead of the Dynamic Prefetch Optimizer

Our adaptive, event-driven prefetching approach has costs that neither traditional hardware nor software techniques incur, which is the cost of generating the prefetch code at runtime. If this cost is high, it can negate the performance gains of prefetching. However, our approach keeps this cost low, because we never interrupt the main thread to run the optimizer, we run the optimizer at a lower execution priority on a spare hardware context, and the optimization thread tends to have low execution resource demands.

To measure the cost of our dynamic prefetch optimizer to see how much it affects the performance of the main execution thread, we run Trident with our prefetch optimization without actually using the optimized traces. That is, the runtime optimizer is triggered to construct and optimize hot traces, but it does not alter the original binary to jump to the optimized trace. The goal of this analysis is to determine the overhead the optimization code imposes on the system while it executes concurrently with the main thread.

We observe the total cost to be only 0.6%. This is much lower than what would be expected in a traditional dynamic optimization system such as Dynamo [2], which would require runtime profiling and frequent switches between the main thread and the optimizer and profiler to enable a similar optimization.

In our current study, an idle hardware context is assumed ready for dynamic optimization. However, we don't have to reserve an entire hardware context for dynamic optimization. Helper threads are invoked for optimization subject to the availability of hardware contexts. Hardware contexts are released after helper threads finish the optimization. Figure 3 shows the percentage of each benchmark's total execution cycles when the optimization thread runs concurrently with the benchmark. The results show that the helper threads are active for a small fraction of the main thread's total execution time, on average 2.2%. Since the optimization time is relatively small, our optimization technique should have opportunities to run even in a real multithreaded system with multiple threads running (e.g., even if contexts only become available during I/O).

Note that the cost of our optimizations will increase with our adaptive techniques; however, the optimization threads with self repairing prefetching are typically active at most 25% more than the base case. Therefore, the total cost is still under 1%. The cost of the interference between the main thread and the helper threads are fully reflected in subsequent results.

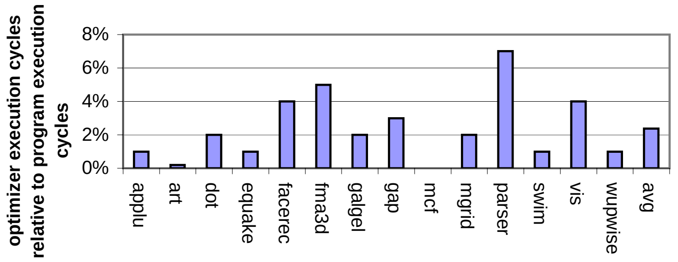

Figure 3: The execution time of optimization threads relative to the program's execution time

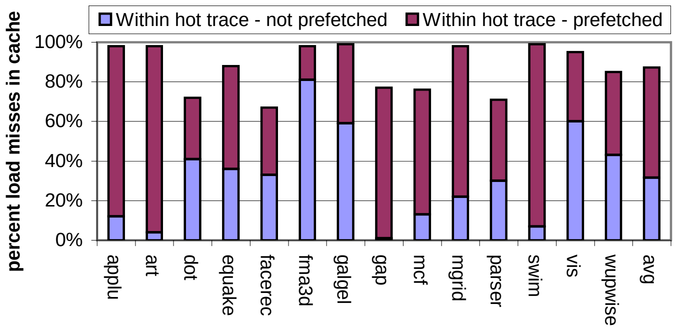

Figure 4: Percentage of load missed covered by hot traces and the prefetcher

5.2 Load Coverage by Software Prefetching

To gage the potential of our software prefetching, we first measure the dynamic load miss coverage. Only load instructions within hot traces can be potentially prefetched by our current optimization technique. Figure 4 shows the percent of cache misses which occur within hot traces, and those that can be potentially software prefetched. The difference between the height of the bar and 100% represents cache misses that do not occur while executing hot traces.

The results show that our hot trace scheme covers over 85% of load misses, and nearly 55% of all misses are potentially covered by our prefetcher. Note that *dot* and *parser* have relatively low miss coverage. This is in large part because of the low dynamic coverage of the hot traces. In contrast, *gap* has low hot trace coverage, but nearly all its hot trace load misses are prefetched.

5.3 Performance of Software Prefetching

In this section, we want to show the performance improvement from basic software prefetching, whole object prefetching, and our adaptive self-repairing prefetcher. The performance improvement over the baseline hardware prefetching is shown in Figure 5.

The first bar shows the performance improvement with a configuration similar to prior dynamic prefetching schemes [14, 5] (*basic*). This divides the average cache miss latency by the average trace execution time to estimate the prefetching distance. We refer to this scheme as the baseline software prefetching approach (even though it includes some features unique to our system, such as strided prefetching of pointer loads). As shown in Figure 5, the average

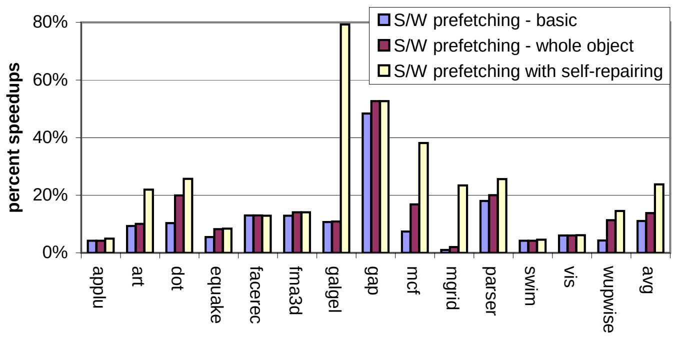

Figure 5: Performance improvement of software prefetching with and without self-repairing relative to hardware prefetching (8X8)

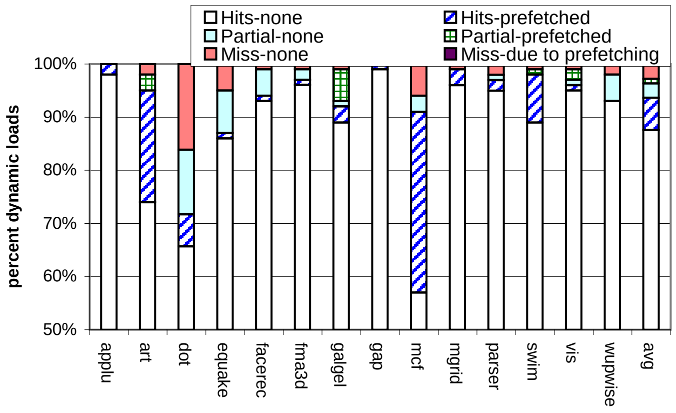

Figure 6: Percentage breakdown of all dynamic loads

speedup for the baseline software prefetching technique is about 11% (over the baseline – hardware prefetching alone). This means the prefetch distance estimation works reasonably well.

The second bar represents the performance improvement of the stride-based same object prefetching over the baseline hardware prefetching (*whole object*). It achieves higher performance improvement than the baseline scheme on the pointer intensive applications such as *dot* and *mcf*. Performance improvement is mainly due to the jump-pointer type prefetching.

The third bar in the figure shows performance of software prefetching with our adaptive self-repair technique. In this approach, the baseline software prefetcher is initiated first, with a default prefetch distance of 1. Its prefetch distance is gradually repaired by the runtime optimizer. As observed in Figure 5, software prefetching with self-repairing significantly boosts the prefetching performance. This is because our adaptive prefetching technique can dynamically correct the prefetch distance as the latency of the hot trace is dynamically tuned. Note that *applu*, *facerec*, and *fma3d* do not show any further performance improvement with self-repairing, because the naive estimates were sufficient – for example, *applu* has such a large inner loop (over 1000 instructions) that a prefetch distance of 1 is optimal.

As noted previously, we also examine an alternate strategy where the initial prefetch distance is estimated more carefully and repaired/incremented from there. We found it achieves performance almost identical to the results shown here. This demonstrates the efficiency with which the system adapts, as the initial value becomes irrelevant. The simpler scheme eliminates certain hardware overheads (also necessary for the non-adaptive results shown) needed to estimate the prefetch distance.

Overall, self-repairing outperforms the basic software prefetching by increasing the speedup from 11% to 23% on average. This comes from a combination of (1) doing a better job of getting the prefetch distance right, and (2) adapting to changes in the hot trace and cache behavior. This demonstrates that our low-overhead, adaptive prefetch approach enables us to overcome the difficulty in calculating statically, or even at runtime, the appropriate prefetch distance.

More insight into our software prefetching approach is provided by Figure 6. Each bar represents the percentage breakdown of all dynamic loads which hit, partially hit, or miss in the cache. When a cache block is fetched due to a prefetch instruction, the first load access to this block is counted as a *Hit-prefetched*, but any subsequent accesses are counted as *Hits-none* rather than prefetching hits. When a line is displaced by a prefetch, we record the tag so that we can identify a *Miss due to prefetching* if a subsequent miss matches that tag. Figure 6 shows two key results that indicate the power of our adaptively repairing prefetcher. Misses due to prefetching rarely occur, and we have a very low incidence of partial prefetch hits.

5.4 Software Prefetching Sensitivity

This section shows the sensitivity of our self-repairing prefetcher to the DLT sizes and our delinquent-load identifying thresholds. We show the results for three load monitoring window sizes (128, 256, and 512). We also show results for the miss rate thresholds of 1%, 3%, 6%, and 12%. This is the miss rate that needs to occur within the load monitoring window to classify the load as a delinquent load. Figure 7 shows the average performance improvement of software prefetching for these different configurations. We found that at least 8 misses during the load's monitoring window provides an adequate indication to classify the load as delinquent. If this number is too small, then it may be overly aggressive with its prefetching. On the other hand, if this number is too big, it may miss delinquent loads. Overall, a cache miss rate threshold of 3% (at least 8 misses out of 256 accesses) works best for the program's we examined.

Figure 8 shows the performance improvement of software prefetching with different delinquent load table (DLT) sizes. We found for most programs that the performance only slightly increases when the table size doubles. However, for benchmarks with large working sets, such as *dot* and *parser*, performance is boosted with a large DLT size. We anticipate the DLT with 1024 entries should work well for most programs.

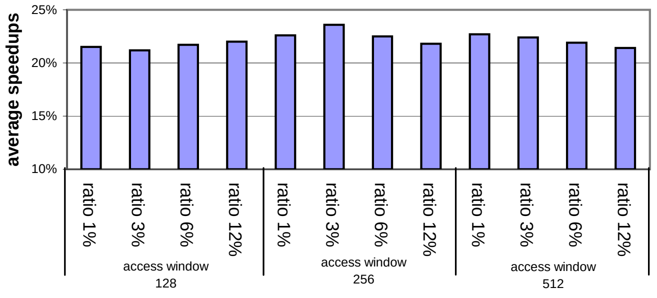

Figure 7: Average performance improvement of software prefetching with different load monitoring window sizes cache miss rate thresholds

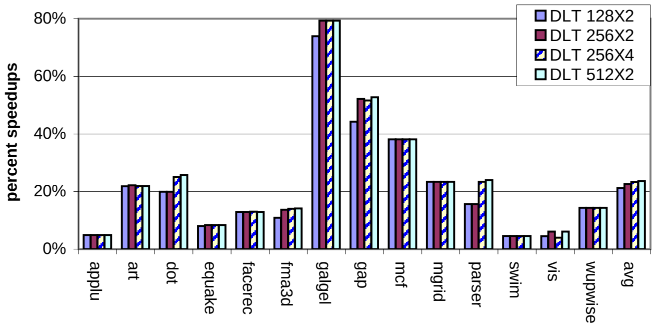

Figure 8: Average performance improvement of software prefetching with different DLT sizes

Since our prefetching technique relies on some new hardware structures, we also want to evaluate how much these hardware resources would boost performance if we simply used them to increase the size of the data cache. If all hardware resources of the DLT table and the watch table are used to increase the size of the L1 data cache, we observe merely a 0.8% performance boost over the baseline.

5.5 Comparison with Hardware Prefetching

Finally, we compare the performance of software prefetching and hardware prefetching alone in Figure 9. Relative to the baseline without any prefetching, our software prefetching with self-repairing outperforms hardware prefetching (*the 8X8 configuration*) in most of benchmarks, with average 11% more speedups. We notice that software prefetching achieves relatively moderate speedups for *dot*, *equake*, and *swim*. This is due to these two factors: (1) Dynamic software prefetching can only target delinquent loads within hot traces. Thus, low coverage of dynamic loads such as in *dot* limits software prefetching performance. (2) Software prefetching has cost usually due to the reduction of effective instruction issue bandwidth, competition of execution resources, and cache capacity conflict. Thus, when the programs such as *equake* and *swim* exhibit simple stride patterns with short prefetching distances, hardware prefetching may be more advantageous. However, when software prefetching is combined with hardware prefetching, the cost is minimized since software prefetching now targets delinquent loads which cannot be handled efficiently by hardware prefetching.

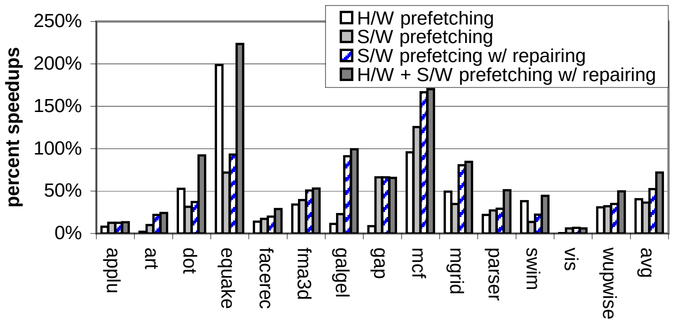

Figure 9: Performance comparison with hardware prefetching

6 Conclusion

Software prefetching is a promising technique to tolerate long memory latencies and achieve full performance from modern processors. Software prefetching has to be accurate and timely in order to be effective.

In this paper, we extend the event-driven, multithreaded dynamic optimization framework, *Trident*, to perform software prefetching by dynamically inserting prefetch instructions into hot traces. The low overhead of the Trident framework allows the runtime optimizer to repeatedly optimize the same trace to adjust prefetching either because existing prefetching is not effective or because the program's behavior changes.

Our prefetching technique also performs stride-based object prefetching to not only hide the latency of the first access of the object, but all of the fields touched in that object. This technique combines the effectiveness of software prefetching, which can analyze the code to recognize access patterns, with many of the advantages of hardware prefetching, which can exploit some patterns static software systems cannot, and which can adapt to the actual runtime behavior of individual loads.

With our dynamic self-repairing prefetcher, which finds the proper prefetch distance by trying multiple distances until the correct one is found, we achieve an average 23% speedup relative to the baseline which includes a hardware stride based prefetcher. In addition, our self-repairing prefetching mechanism achieves 12% better performance than prior dynamic prefetching techniques without repairing.

Acknowledgments

The authors would like to thank the anonymous reviewers for their comments and suggestions. This research was supported by NSF grants CNS-0311683 and CCF-0311710, Semiconductor Research Corporation grant 2005-HJ-1313, and grants from Microsoft and Intel.

References

[1] M. Annavaram, J. Patel, and E. Davidson. Data prefetching by dependence graph precompuation. In *Annual International Symposium on Computer Architecture*, July 2001.

[2] V. Bala, E. Duesterwald, and S. Banerjia. Dynamo: A transparent dynamic optimization system. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2000.

[3] B. Cahoon and K.S. McKinley. Data flow analysis for software prefetching linked data structures in java. In *International Conference on Parallel Architectures and Compilation Techniques*, September 2001.

[4] B. Callahan, K. Kennedy, and A. Porterfield. Software prefetching linked data structures in java. In *Proceedings of the Symposium on Architectural Support for Programming Languages and Operating Systems*, April 1991.

[5] H. Chen, J. Lu, W. Hsu, and P. Yew. Continuous adaptive object-code re-optimization framework. In *Ninth Asia-Pacific Computer Systems Architecture*, 2004.

[6] T. Chilimbi and M. Hirzel. Dynamic hot data stream prefetching for general purpose programs. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2002.

[7] J.D. Collins, D.M. Tullsen, H. Wang, and J.P. Shen. Dynamic speculative precompuation. In *34th International Symposium on Microarchitecture*, December 2001.

[8] J.D. Collins, H. Wang, D.M. Tullsen, C.J. Hughes, Y.-F. Lee, D. Lavery, and J.P. Shen. Speculative precomputation: Long-range prefetching of delinquent loads. In *28th Annual International Symposium on Computer Architecture*, July 2001.

[9] K.I. Farkas, P. Chow, N.P. Jouppi, and Z. Vranesic. Memory-system design considerations for dynamically-scheduled processors. In *24th Annual International Symposium on Computer Architecture*, June 1997.

[10] IIya Ganusov and Martin Burtscher. Future execution: A hardware prefetching technique for chip multiprocessors. In *International Conference on Parallel Architectures and Compilation Techniques*, September 2005.

[11] Tatsushi Inagaki, Tamiya Onodera, Kideaki Komatsu, and Toshio Nakatani. Stride prefetching by dynamically inspecting objects. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 2003.

[12] N. Jouppi. Improving direct-mapped cache performance by the addition of a small fully associative cache and prefetch buffers. In *17th Annual International Symposium on Computer Architecture*, May 1990.

[13] D. Kim and D. Yeung. Design and evaluation of compiler algorithm for pre-execution. In *Proceedings of the Symposium on Architectural Support for Programming Languages and Operating Systems*, October 2002.

[14] Jiwei Lu, Howard Chen, Pen-Chung Yew, and Wei-Chung Hsu. Design and implementation of a lightweight dynamic optimization system. In *The Journal of Instruction-Level Parallelism*, Jun 2004.

[15] C.-K. Luk. Tolerating memory latency through software-controlled pre-execution in simultaneous multithreading processors. In *28th Annual International Symposium on Computer Architecture*, July 2001.

[16] C.-K. Luk and T. Mowry. Compiler-based prefetching for recursive data structures. In *Proceedings of the Symposium on Architectural Support for Programming Languages and Operating Systems*, October 1996.

[17] A. Moshovos, D. Pnevmatikatos, and A. Baniasadi. Slice processors: An implementation of operation-based prediction. In *International Conference on Supercomputing*, June 2001.

[18] T.C. Mowry, M.S. Lam, and A. Gupta. Design and evaluation of a compiler algorithm for prefetching. In *Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 62–73, October 1992.

[19] S. Palacharla and R. Kessler. Evaluating stream buffers as secondary cache repacement. In *21st Annual International Symposium on Computer Architecture*, April 1994.

[20] R. Rabbah, H. Sandanagobalane, M. Ekpanyapong, and W. Wong. Compiler orchestrated prefetching via speculation and prediction. In *ASPLOS*, October 2004.

[21] A. Roth, A. Moshovos, and G. Sohi. Dependence based prefetching for linked data structures. In *Proceedings of the Symposium on Architectural Support for Programming Languages and Operating Systems*, October 1998.

[22] A. Roth and G. Sohi. Effective jump-pointer prefetching for linked data structures. In *Annual International Symposium on Computer Architecture*, May 1999.

[23] A. Roth and G. Sohi. Speculative data-driven multithreading. In *Proceedings of the Seventh International Symposium on High-Performance Computer Architecture*, July 2001.

[24] R. H. Saavedra and D. Park. Improving the effectiveness of software prefetching with adaptive execution. In *International Conference on Parallel Architectures and Compilation Techniques*, 1996.

[25] T. Sherwood, E. Perelman, and B. Calder. Basic block distribution analysis to find periodic behavior and simulation points in applications. In *International Conference on Parallel Architectures and Compilation Techniques*, September 2001.

[26] T. Sherwood, E. Perelman, G. Hamerly, S. Sair, and B. Calder. Discovering and exploiting program phases. *IEEE Micro: Micro's Top Picks from Computer Architecture Conferences*, December 2003.

[27] T. Sherwood, S. Sair, and B. Calder. Predictor-directed stream buffers. In *33rd International Symposium on Microarchitecture*, 2000.

[28] J.E. Smith and W.-C. Hsu. Prefetching in supercomputer instruction caches. In *Proceedings of Supercomputing*, November 1992.

[29] D.M. Tullsen, S.J. Eggers, J.S. Emer, H.M. Levy, J.L. Lo, and R.L. Stamm. Exploiting choice: Instruction fetch and issue on an implementable simultaneous multithreading processor. In *23rd Annual International Symposium on Computer Architecture*, May 1996.

[30] D.M. Tullsen, S.J. Eggers, and H.M. Levy. Simultaneous multithreading: Maximizing on-chip parallelism. In *22nd Annual International Symposium on Computer Architecture*, June 1995.

[31] Youfeng Wu. Efficient discovery of regular stride patterns in irregular programs and its use in compiler prefetching. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 2002.

[32] Y. Yadama, J. Gyllenhaal, G. Haab, and W.M. Hwu. Data relocation and prefetching for programs with large data sets. In *27th International Symposium on Microarchitecture*, 1994.

[33] Weifeng Zhang, Brad Calder, and Dean M. Tullsen. An event-driven multithreaded dynamic optimization framework. In *International Conference on Parallel Architectures and Compilation Techniques*, September 2005.

[34] Z. Zhang and J. Torrellas. Speeding up irregular applications in shared-memory multiprocessors: memory binding and group prefetching. In *Annual International Symposium on Computer Architecture*, June 1995.