Prefetch Injection Based on Hardware Monitoring and Object Metadata

Ali-Reza Adl-Tabatabai, Richard L. Hudson, Mauricio J. Serrano, Sreenivas Subramoney

Programming Systems Laboratory
Microprocessor Technology Laboratory
Intel Corporation
Santa Clara, California USA

{Ali-Reza.Adl-Tabatabai | Rick.Hudson | Mauricio.J.Serrano | Sreenivas.Subramoney}@Intel.com

ABSTRACT

Cache miss stalls hurt performance because of the large gap between memory and processor speeds – for example, the popular server benchmark SPEC JBB2000 spends 45% of its cycles stalled waiting for memory requests on the Itanium® 2 processor. Traversing linked data structures causes a large portion of these stalls. Prefetching for linked data structures remains a major challenge because serial data dependencies between elements in a linked data structure preclude the timely materialization of prefetch addresses. This paper presents *Mississippi Delta* (MS Delta), a novel technique for prefetching linked data structures that closely integrates the hardware performance monitor (HPM), the garbage collector's global view of heap and object layout, the type-level metadata inherent in type-safe programs, and JIT compiler analysis. The garbage collector uses the HPM's data cache miss information to identify cache miss intensive traversal paths through linked data structures, and then discovers regular distances (*deltas*) between these linked objects. JIT compiler analysis injects prefetch instructions using deltas to materialize prefetch addresses.

We have implemented MS Delta in a fully dynamic profile-guided optimization system: the StarJIT dynamic compiler [1] and the ORP Java virtual machine [9]. We demonstrate a 28-29% reduction in stall cycles attributable to the high-latency cache misses targeted by MS Delta and a speedup of 11-14% on the cache miss intensive SPEC JBB2000 benchmark.

Categories and Subject Descriptors

D.3.3 [**Programming Languages**]: Language Contructs and Features – *Dynamic storage management; Classes and objects.*

General Terms

Algorithms, Measurement, Performance, Design, Experimentation, Languages.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

PLDI'04, June 9–11, 2004, Washington, DC, USA.

Copyright 2004 ACM 1-58113-807-5/04/0006…$5.00.

Keywords

Prefetching, compiler optimization, garbage collection, cache misses, profile-guided optimization, virtual machines.

1. INTRODUCTION

Memory systems performance remains one of the biggest bottlenecks in processor performance. Figure 1 illustrates this problem, showing the percentage of execution cycles attributed to data cache stalls running the SPEC JBB2000 and SPEC JVM98 benchmarks [29] on the Itanium® 2 processor. For memory-intensive programs – such as SPEC JBB2000 and db – the processor spends up to 45% of its execution cycles stalled waiting for memory. This problem will only worsen as processor speed continues to outpace memory speed, and as the demand on the memory subsystem increases due to chip-level multiprocessing.

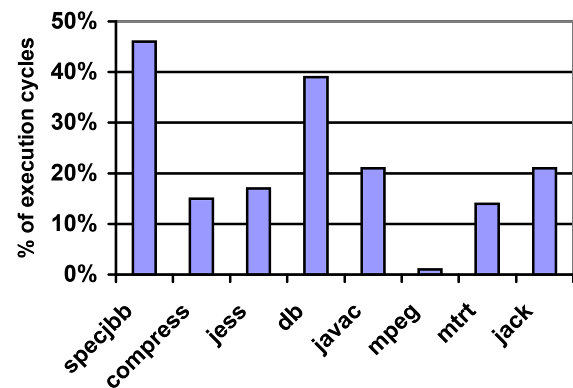

Figure 1. Percentage of execution cycles stalled due to data cache misses.

Prefetching tackles the memory latency problem by fetching data into processor caches in advance of their use. To prefetch in a timely fashion, the processor must materialize the prefetch address early enough to overlap the prefetch latency with other computations or latencies. For both hardware and software-based strategies, prefetching for linked data structures remains a major challenge because serial data dependencies between elements in a linked data structure preclude the timely materialization of prefetch addresses.

Figure 2 illustrates the challenge of prefetching for linked data structures. This figure shows three linked objects. Assume that traversal of this linked structure causes all three objects to miss in the cache. The processor cannot load the String object until it has loaded

the brandInfo field of Item. Similarly, it cannot load the Char Array object until it has loaded the value field of String. Thus data dependences serialize cache misses during traversal, as illustrated by the chart.

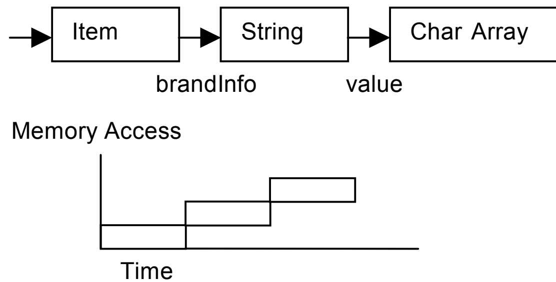

Figure 2. Serial data dependences in linked data structures.

This paper presents *Mississippi Delta (MS Delta)*, a novel technique for prefetching linked data structures that starts with information from the hardware performance monitor (HPM), and uses the garbage collector (GC) to discover cache miss intensive paths through linked data structures and regular distances (*deltas*) between objects in these structures. The JIT compiler subsequently uses the deltas to materialize prefetch addresses in a timely fashion.

Figure 3 shows how MS Delta uses deltas to predict prefetch addresses, thus avoiding data dependences and permitting timely prefetches. The GC discovers a regular delta between Item and String, as well as Item and Char Array. The JIT compiler uses this delta to inject prefetches of String and Char Array such that the prefetch latency overlaps the miss latency of Item, as illustrated by the chart.

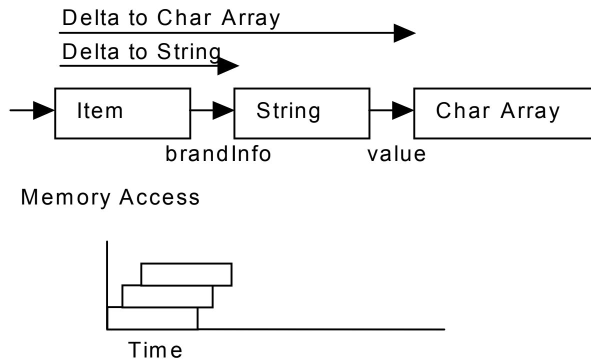

Figure 3. Prefetching using deltas.

Figure 4 shows the high-level MS Delta algorithm as it abstracts from raw HPM samples up to prefetches. The HPM provides samples of high-latency cache misses. Each sample includes the instruction pointer address (IP) and the referent effective address (EA) of the memory access. MS Delta abstracts these raw samples first into the *delinquent objects* that caused the misses and then into a high level *metadata graph*, whose nodes represent object types, and whose edges represent relations induced by fields and array elements containing references. During heap traversal, the GC uses the delinquent objects to discover edges in the metadata graph that approximate the high-latency traversals between linked data. It then composes these edges into *delinquent paths* representing linked data structure traversals that cause high- latency cache misses. Taking advantage of object placement, the GC determines regular deltas between objects along the paths. Knowing the deltas and the paths simplifies the JIT compiler analysis needed to schedule prefetches along a traversal path: The JIT combines the address of the first object in the path with the deltas to materialize prefetch targets. This means that the miss latency experienced by the first object in a traversal path hides the miss latency of subsequent objects along the path.

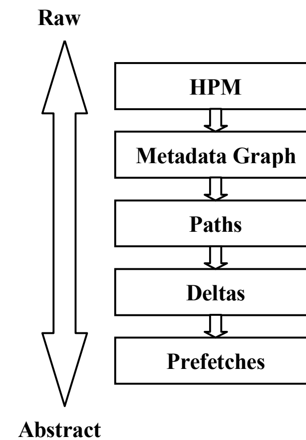

Figure 4. High-level MS Delta algorithm.

This paper makes the following novel contributions. First, MS Delta uses dynamic hardware-based data cache miss profiling instead of software-based memory access profiling to greatly reduce the cost of profiling and to concentrate analysis on loads that actually cause cache misses instead of loads that fall along frequently executed yet cache resident paths. Second, MS Delta leverages the GC's heap traversal to infer object layout properties (in the form of deltas between linked objects) useful for generating prefetch addresses in linked data structures. By using deltas to materialize prefetch addresses, MS Delta avoids the inherent serial dependences that make prefetching for linked data structures difficult. Third, MS Delta uses the GC's object placement ability to maintain the deltas used for prefetching. Fourth, MS Delta abstracts the raw cache miss data up to the type and type connectivity level instead of tracking individual addresses and objects. By abstracting the voluminous miss address information up to type-level metadata, MS Delta concisely models cache miss behavior, eliminating the need to maintain potentially large historic address specific structures. In summary, MS Delta couples hardware data cache profiles and global heap properties with metadata to enhance and guide dynamic JIT compiler analysis, recompilation, and prefetch injection decisions.

The rest of this paper is organized as follows. The next section describes our experimental framework. Section 3 describes the algorithm for building the metadata graph using the HPM samples. Sections 4 and 5 describe how the GC builds delinquent paths and discovers deltas between objects, respectively. Section 6 describes the JIT algorithm for injecting prefetches and Section 7 presents experimental results on SPEC JVM98 and SPEC JBB2000. Section 8 discusses related work.

2. EXPERIMENTAL FRAMEWORK

We have implemented MS Delta in the context of the StarJIT dynamic compiler [1] and the ORP JVM [9] running on the Itanium® 2 platform [19]. The ORP JVM supports dynamic profile-guided feedback to the JIT compiler, and contains a GC tool kit capable of supporting multiple collection algorithms [24] (e.g., generational, sliding compaction, parallel, concurrent). To improve memory performance, ORP and StarJIT can compress pointers from 64 bits to 32 bits [2]. We have optimized the GC to take advantage of the Itanium® 2 processor features [16].

StarJIT features an SSA-based intermediate representation and performs aggressive profile-guided global optimizations such as devirtualization, inlining, bounds-check elimination, and others. StarJIT also includes aggressive profile-guided code generator optimizations such as trace scheduling, speculation, code layout, and others. StarJIT and ORP currently support two profiling strategies: control-flow edge profiling using software instrumentation and hardware cache miss sampling. The Itanium® processor supports sampling of hardware cache misses via an HPM called the Performance Monitoring Unit (PMU).

All measurements reported in this paper were gathered by running the SPEC JVM98 and SPEC JBB2000 benchmarks on a commercially available 4 processor 1.5 GHz Itanium® 2 machine with 16 gigabytes of memory and 6 megabytes of 3rd-level cache (with 128-byte cache lines) running Microsoft's Windows 2003 Enterprise Edition. Figure 5 compares the performance of StarJIT and ORP with a leading edge commercial JVM for the Itanium® processor (BEA WebLogic JRockit™ 1.4.1 SDK Developer Release). On these benchmarks, our baseline system performs 5-59% faster than the commercial JVM.

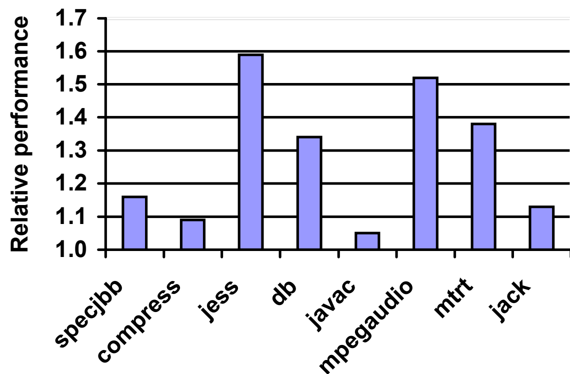

Figure 5. Performance of ORP relative to a commercial JVM.

3. BUILDING THE METADATA GRAPH

MS Delta represents cache miss information in terms of types and their relationships using a concise metadata graph. This section describes how MS Delta abstracts HPM samples first into objects, then into types and related loads, and finally into the edges needed to build this graph.

3.1 Delinquent Objects

Each PMU sample forms a tuple comprising the load instruction pointer (IP), target effective address (EA), and latency of the load causing the miss. MS Delta programs the PMU to sample only high-latency cache misses so that it can concentrate on loads that access off-chip memory. MS Delta further eliminates samples whose EA do not fall within the contiguous garbage collected heap.

Each sample refers to an object that caused a cache miss; we call such objects *delinquent objects*. Each object starts with a header containing a virtual function table pointer (*vtable*) that identifies the type of the object. MS Delta abstracts a sample into a delinquent object by scanning backwards in memory starting from the EA of the sample, looking for a word that looks like a valid vtable. To improve accuracy, the search probes a hash table containing valid vtables recorded by the class loader. The search ensures constant time by bounding the number of 8-byte words it visits to 100. If the search fails to locate a valid vtable it simply discards the sample. Of course a random sequence of bits can masquerade as a vtable (rare in practice) – we discuss later how to deal with this inaccuracy. After abstracting the effective addresses up into delinquent objects, MS Delta constructs the *delinquent object set* from the delinquent objects and their respective samples.

Limiting the vtable search to at most 100 captures 97% of the interesting types in all of the SPEC JVM98 and SPEC JBB2000 benchmarks except for compress, which exhibits misses in large arrays. MS Delta focuses on misses in linked data structures, which usually consist of relatively small objects instead of large arrays. Standard loop prefetching techniques can typically address misses to large arrays (see [30] for an overview).

3.2 Delinquent Types and Loads

Experimental data shows that only a few types and a few loads cause the majority of cache misses [27]. We refer to these types and loads as *delinquent types* and *delinquent loads* [10], respectively. The set of delinquent types and loads concisely characterizes an application's cache misses. After identifying delinquent objects, MS Delta further abstracts the HPM data by rolling up the information contained in the delinquent object set to the set of delinquent types.

To compute delinquent types the algorithm iterates through the delinquent object set accumulating the total miss latency for each type encountered. It then sorts these types by their latency and retains the topmost types whose cumulative latency contributes to most of the overall latency (types contribute to 90% of the latency in SPEC JBB2000, 5 delinquent types contribute to 99% of the latency in db, and 9 delinquent types contribute 96% in mtrt. Similarly, 88 loads contribute 88% of the latency in SPEC JBB2000, and 24 loads contribute 90% of the latency in db.

A delinquent load can access more than one delinquent type because of subtyping. We observed that in our benchmarks, each delinquent load accesses a dominant delinquent type. To filter out errors from misidentifying the correct vtable in a sample, MS Delta discards samples whose loads access non-delinquent types. for example, 98%). These types form the set of delinquent types. A similar algorithm computes the set of delinquent loads, looking at sample IPs instead of types.

Figure 6 and 7 show the cumulative cache miss latency contributed by the top sorted delinquent loads and types, respectively. These figures show that the set of delinquent loads and delinquent types is tractable. For example, 10 delinquent

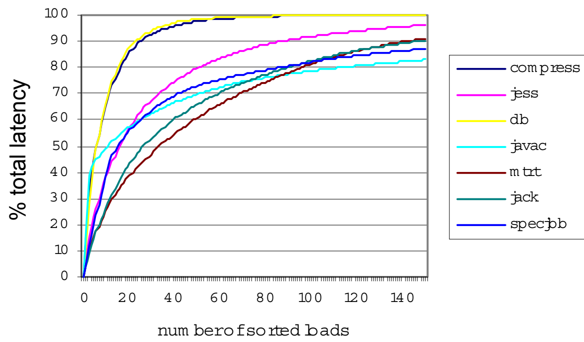

Figure 6. Cumulative contribution of delinquent loads to total cache miss latency.

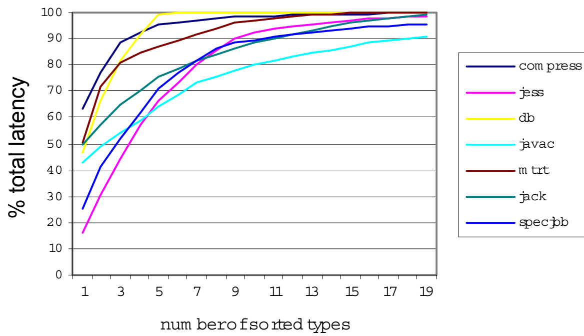

Figure 7. Cumulative contribution of delinquent types to total cache miss latency.

3.3 The Metadata Graph

The metadata graph consists of delinquent types and the edges between them called *delinquent edges*. MS Delta piggybacks on the GC's heap traversal to collect information needed to identify the metadata graph. When MS Delta encounters an object of delinquent type during heap traversal, it iterates through each of the reference fields in the object to see if they point to a child object of delinquent type. If MS Delta finds such a child object the pair has one of 4 possible *delinquent relationships*:

- Delinquent object to delinquent object (O→O) if both objects are delinquent objects.
- Delinquent object to delinquent type (O→T) if only the parent is a delinquent object.
- Delinquent type to delinquent object (T→O) if only the child is a delinquent object.
- Delinquent type to delinquent type (T→T) if neither is a delinquent object.

MS Delta maintains counts for each of the relationships in a sparse NxN matrix called the *Dynamic Metadata Table* (DMT), indexed using ids assigned to delinquent types. Each element in the matrix contains a linked list of nodes. Each node represents a field pointing from the parent type to the child type, and records delinquent relationships corresponding to that particular field. At the end of the heap traversal the matrix concisely characterizes the dynamic connectivity between each of the delinquent types. Note, that both the DMT and the metadata graph are small data structures; for example, SPEC JBB2000 contains only 10 delinquent types and thus produces a sparse 10 X 10 matrix. Furthermore, even though the object identification technique discussed in Section 3.1 occasionally misidentifies objects, the GC only encounters valid objects, thus making misidentification benign.

Figure 8 shows a portion of the metadata graph for SPEC JBB2000. Type Item points to type String via fields name and brandInfo. Type String points to type Char Array, via field
value.

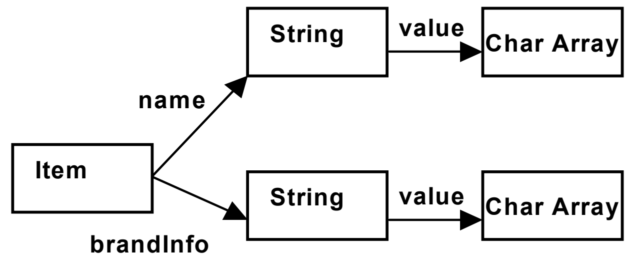

Figure 8. Portion of SPEC JBB2000 metadata graph.

Figure 9 shows the DMT for the metadata graph in Figure 8. The y-axis contains the parent delinquent types and the x-axis contains the child delinquent types. Each element of the matrix contains a list of the fields connecting that element's parent type to its child type, or null if no fields connect the two types. For example, the element DMT[Item, String] consists of a linked list of two nodes, one of which corresponds to the brandInfo field connecting type Item to type String and the second corresponds to the name field between the same two types. These nodes contain the delinquent relationship counters updated during heap traversal.

Child Types

	String	char array	Item
String	Null	field ("value"); int O_to_O; int O_to_T; int T_to_O; int T_to_T; edge *next_edge;	Null
char array	Null	Null	Null
Item	field("brandInfo"); int O_to_O; int O_to_T; int T_to_O; int T_to_T; edge *next_edge → field("name"); ...	Null	Null

These counts are updated during heap traversal

Figure 9. Portion of SPEC JBB2000 dynamic metadata table.

4. DETERMINING DELINQUENT PATHS

A delinquent edge connects a parent type to a child type in the metadata graph and represents a delinquent load of the child using a reference field in the parent. A *delinquent path* comprises one or more delinquent edges and represents a traversal of a linked data structure that frequently misses. The first type in a delinquent path is the *delinquent base type.* Discovering the delinquent base type and the associated load is key to locating where to inject prefetches.

MS Delta uses the DMT to identify delinquent edges. For each pair of delinquent types MS Delta examines the DMT to see if heap traversal discovered an edge connecting the pair. If so, MS Delta sums the O→O and the T→O delinquent relationships for that edge. If this is greater than some lower bound N (a small fraction of the total number of distinguished objects), then MS Delta considers this edge a delinquent edge candidate and further calculates a *complex edge weight (CEW)*:

$$CEW = 100*O \rightarrow O + 10*T \rightarrow O + O \rightarrow T$$

CEW gives the highest weighting to the O→O component because it represents good evidence that the application followed this edge, causing a cache miss. The T→O component is important because it indicates how we actually reached a known delinquent object following an edge (field dereference) from a delinquent type. The O→T component is less important since it gives less assurance that the edge being followed leads to a delinquent object. This is particularly true if multiple fields in a delinquent type have the same type. MS Delta sorts the delinquent edge candidates based on their CEW, filtering all but the topmost edges. This information is then rolled up into the metadata graph.

MS Delta builds delinquent paths by composing delinquent edges. Initially, each delinquent edge is a delinquent path. To lengthen a delinquent path, MS Delta recursively finds an edge whose parent type matches the child type of the last edge in the path. The algorithm terminates once it finds the longest path for each delinquent edge, it encounters an array, or the path reaches a length limit. Each path has a weight equal to the sum of its edge's CEWs. This delinquent path information is then rolled up into the metadata graph.

The algorithm for building delinquent paths misses some paths because it requires all types along the path to be delinquent. If a delinquent object sits adjacent to its next downstream object so that they often reside in the same cache line, the second object will rarely be delinquent. Further analysis by the GC can detect when two connected types tend to reside on the same cache line, allowing the second type to be considered as part of the delinquent path.

5. DETERMINING PREFETCH DELTAS

For each delinquent path, the delta determination algorithm computes deltas relative to the path's delinquent base type. The algorithm iterates through the delinquent object set and each time it encounters an object whose type matches a delinquent base type, it traces the objects along delinquent paths starting from the delinquent object. For each edge it traverses along the path (i.e., each field it dereferences) it calculates the delta from the base object, binning the delta into a *delta histogram* for that edge. After collecting the deltas into histograms, the algorithm discards deltas whose bins comprise less that 10% of the deltas for that edge and annotates the metadata graph's edges with the remaining deltas. Note, that by only traversing paths starting from delinquent objects, MS Delta effectively samples paths and deltas.

The deltas indicate distances between bases of two objects that miss. Cache misses, however, typically occur on an access to a field at some offset within the object; therefore, MS Delta also determines the *effective offsets* that cause misses by iterating over the delinquent object set, subtracting the object base from the effective address the HPM delivered. MS Delta then annotates the metadata graph's edges with these effective offsets. The JIT adds the deltas and the effective offsets to the address of the base object to determine the prefetch target. This refinement proved more effective than simply prefetching the entire referent object.

The delta determination algorithm computes path-specific deltas, improving prefetch accuracy. For example, depending on the path, the character array associated with each string object is sometimes located before the string object and sometimes after it. One path wants to use the negative delta to prefetch the character array while another path wants to use a positive delta. The delta determination algorithm distinguishes between these two paths.

5.1 Maintaining Deltas during GC

The ORP GC allocates using a frontier pointer scheme [16] resulting in allocation order object placement. For many applications (e.g., SPEC JBB2000 and db) allocation order results in delinquent objects having regular deltas along delinquent paths. To maintain allocation order and also deal with fragmentation the ORP GC employs sliding compaction.

Performing compaction prior to calculating deltas results in more regular deltas between objects. This seems primarily due to short-lived objects being interspersed with the longer living delinquent objects. Because MS Delta abstracts deltas up to the type level, improving delta consistency improves prefetch effectiveness. Compaction (or any other object movement during garbage collection), however, requires the GC to update the delinquent object set during the repoint phase.

5.2 Computing Prefetch Deltas

The benefit of prefetching a cache line is the total cache miss latency avoided. To estimate this, MS Delta combines delta information from all delinquent paths starting at each base delinquent type. It adds each delta and effective offset and divides the sum by the cache line size to compute a delta in terms of cache lines. It then bins the associated latency into a histogram for that base delinquent type. Each histogram bin reflects the benefit from prefetching that bin's cache line. MS Delta then assigns a low, medium, or high confidence rating based on this benefit, and does not prefetch low confidence cache lines. MS Delta prefetches medium confidence cache lines using a non-temporal prefetch instruction [19], which minimizes the risk of cache pollution, and prefetches high confidence cache lines using a regular prefetch instruction.

Figure 10 shows the layout for the metadata graph shown in Figure 8. Consider the two paths starting at Item containing Char Arrays of variable size. The varying size of the Char Arrays reduces the accuracy of the deltas along the second path. If, however, the deltas are abstracted to cache lines as above, the prefetch will help along at least one of the paths.

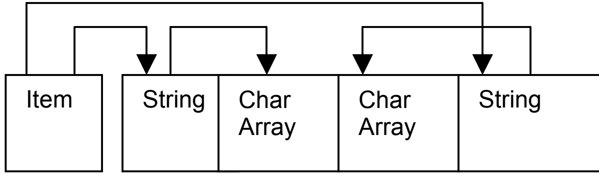

Figure 10. Example layout for Figure 8.

6. INJECTING PREFETCHES

The compiler identifies the candidate methods in which MS Delta injects prefetches, and the intermediate representation (IR) for the delinquent loads, using a map from IPs to loads in the IR. For each delinquent load of a base type, the compiler follows use-def links to track down the operation that produces that load's effective address – that is, the operation that adds the field offset to the object's base address. It then generates instructions that add the base address to the constant prefetch deltas, thus materializing prefetch addresses. Finally, the JIT injects prefetches before the base type's delinquent load. Before invoking the compiler, MS Delta filters out any delinquent load of a base type that accounts for less than 10% of all misses of its delinquent type, thus focusing the compiler analysis.

In contrast to prefetch techniques in scientific codes, which typically overlap prefetch latency with computation, MS Delta overlaps prefetch latency with the miss latency of the base type's delinquent load. This approach eliminates microarchitecture-specific memory latency calculations, and generalizes across microarchitecture generations (which may have different memory latency) because both the miss and prefetch latency increase at the same rate.

7. RESULTS

We have run MS Delta against SPEC JBB2000 and the SPEC JVM98 suite. SPEC JBB2000 shows considerable improvement, while the SPEC JVM98 benchmarks show neither improvements nor degradations.

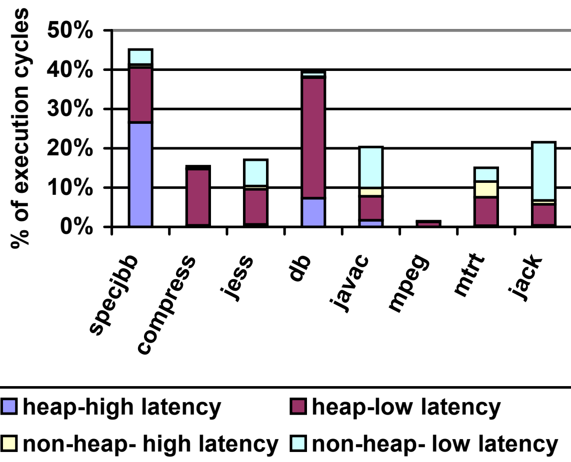

Figure 11. Cycles spent on memory stalls.

Figure 11 shows that the SPEC JVM98 benchmarks do not exhibit many high-latency cache misses to the heap, whereas SPEC JBB2000 spends 27% of its execution stalled on high-latency cache misses to the heap. This figure categorizes each cache miss latency cycle according to whether it was to the heap, and whether it was a high or low-latency miss. High-latency misses typically access DRAM, which has a latency of 300-400 cycles. MS Delta, therefore, should aim to improve on SPEC JBB200 without degrading the other applications.

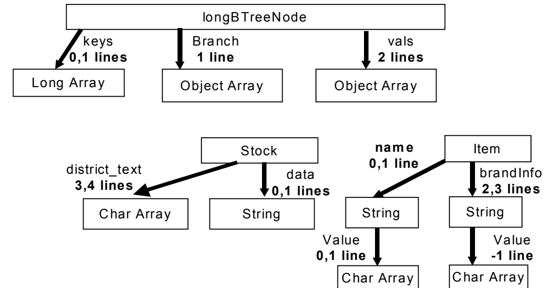

Figure 12. SPEC JBB2000 metadata graph with deltas.

Figure 12 shows the metadata graph that MS Delta builds for SPEC JBB2000 when run using compressed 32-bit pointers [2]. The edges contain the deltas (in cache lines) and field names. SPEC JBB2000 has seven delinquent types, some lying along multiple paths. MS Delta injects prefetches for seven delinquent paths at four locations in two methods.

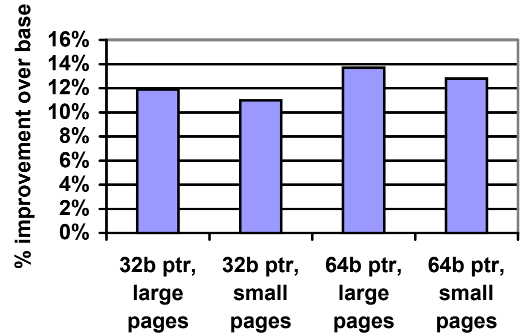

Figure 13. Improvements in SPEC JBB2000.

Figure 13 shows that MS Delta speeds up SPEC JBB2000 by 11-14% over our baseline system. This figure shows speedups for 4 different system configurations: using 64-bit versus compressed 32-bit pointers, and small (8 kilobyte) versus large (16 megabyte) pages.

Figure 14 shows the non-heap, low-latency heap, and high-latency heap stall cycles for the baseline system and MS Delta, using 64-bit and 32-bit pointers with a small page size. With 32-bit pointers, MS Delta reduces high-latency heap miss stalls from 25% down to 18% of the total cycles, a 28% reduction in high-latency heap miss stalls. Similarly, with 64-bit pointers, MS Delta reduces high-latency heap miss stalls from 28% down to 20% of the total cycles, a 29% reduction in latency in high-latency heap miss stalls.

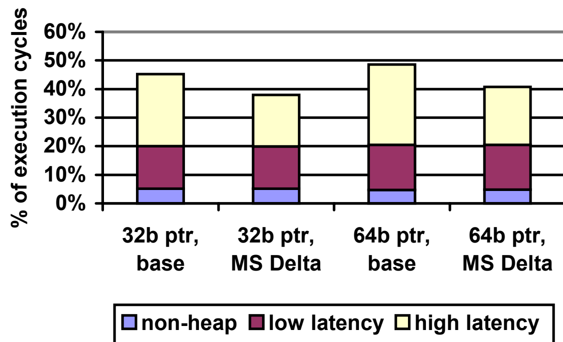

Figure 14. Data stall reduction in SPEC JBB2000 with 64-bit and compressed 32-bit pointers.

The ORP GC uses a frontier pointer based allocation scheme and a sliding compaction algorithm. We had assumed that the GC would need to proactively place objects based on delinquent paths but discovered that this didn't appear to be required. Further investigation revealed that allocation order placement already arranged objects in an appropriate order. It is important to note that object placement schemes that segregate individual objects based on size or schemes that result in random placement will not produce deltas usable by MS Delta.

The SPEC JVM98 benchmark db is an example of a benchmark that is sensitive to cache size. On an earlier generation Itanium® processor with a smaller 3-megabyte 3rd-level cache, MS Delta located two delinquent paths, and prefetch injection resulted in an overall improvement of 2%. When we moved to the later generation processor a noticeable reduction in high-latency cache misses reduced the opportunity for MS Delta and while our performance did not degrade we no longer saw any improvement.

Db is interesting in another aspect since it shows one of the drawbacks to the current algorithm, that of misidentifying the IP associated with the base of a delinquent path. MS Delta detected two delinquent loads that loaded the same type. At one delinquent load this type formed the base of a delinquent path and the inserted prefetch produced a 5% improvement. Unfortunately at the other IP the detected delinquent path did not actually exist. Injecting prefetches at the second IP reduces performance from 5% down to only 2%, a graphic example of the dangers of over aggressive prefetching. We leave for future work how to deal with this problem.

7.1 Overheads

Any system relying on dynamic profile-guided optimization includes a feedback loop that profiles the running code followed by recompiling the code followed by executing the code. The value of the optimization is the difference between the benefit of the optimizations and the cost of the profiling, analysis, and recompilation. Reducing profiling cost or frequency can control the cost. MS Delta uses the HPM, which has a very low sampling overhead, to reduce the cost. Not surprisingly, the cost related to GC heap traversal dominates.

For SPEC JBB2000, a sampling rate of 1400, which generates one sample every 1400 high-latency cache misses, results in the delinquent paths shown in Figure 12. Increasing the rate produces the same delinquent types and paths. (Similarly for db on the previous generation Itanium® processor.) Obtaining a sample costs about 1000 cycles. We measured a negligible overhead (i.e., much less than 1%) due to collecting samples on SPEC JBB2000.

The samples are collected in buffers of 5000 samples each. GC consumes full buffers, flushing any partially-filled buffers. If an application does not exhibit enough cache misses between GCs, MS Delta will not see sufficient cache misses to apply its analysis (and will not incur any overhead other than sample collection). To characterize SPEC JVM98, which has few high-latency cache misses, we had to increase the sampling rate to 100.

GC takes up approximately 2% of execution time for SPEC JBB2000. Performing path and delta determination at each GC (and then discarding the results) increases this time to 4%. So for SPEC JBB2000 the upper bound on the cost is 2% or double the normal cost of GC.

Performing delta and path analysis at a GC is necessary only if the program moves to a new phase that could benefit from prefetching. Therefore, once MS Delta has samples it determines the potential benefit from prefetching. If the benefit is too low, or there is no significant change in miss behavior since the last prefetch optimization (i.e., no phase change), then there is no point in performing any further analysis regardless of the cost. We detect phase changes using the following metrics:

- Changes in the set of delinquent types.
- Changes in the set of delinquent loads.
- Increase in the rate of high-latency cache misses.
- Changes in the number of threads producing samples.

On SPEC JBB2000, MS Delta processes the samples, calculates the paths and deltas during GC, and recompiles only once, resulting in minimal overhead during warm-up. The total compilation cost was approximately 0.08 seconds, negligible for SPEC JBB2000, which runs for several minutes.

7.2 Filtering Characterization

Figure 15 shows the effectiveness of filtering samples for our benchmarks; the only benchmark not shown is compress for which the vtable search algorithm could not find a vtable for almost all samples because most misses were to large arrays.

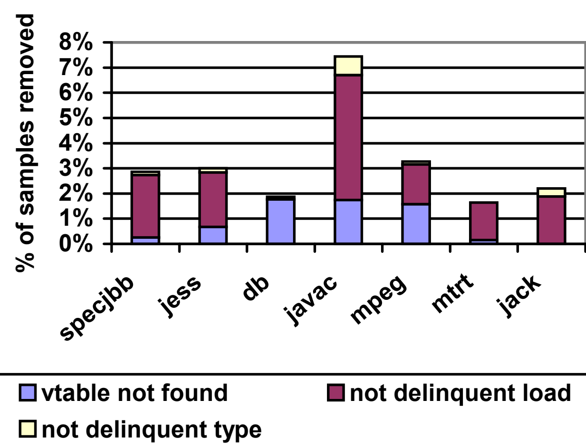

Figure 15. Effectiveness of filtering.

This figure shows the percentage of samples eliminated if (a) the object vtable was not found, which eliminates less than 2% of the samples; (b) the sample's load was not a delinquent load, which eliminates up to an additional 5% of the samples; and (c) the sample's delinquent load did not access a delinquent type, which eliminates less the 1% of the samples. These filters combined eliminate 2 to 7.5% of the samples.

7.3 Delta Characterization

To determine whether allocation order produces predictable deltas between objects, we instrumented the GC to look at each reference and determine the number of cache lines separating the referencing object from the referent object. From this we construct a histogram of deltas for each field of reference type, and extract the most common cache line delta from each histogram. We sum these most common deltas to determine the percent of total references that exhibit the most common deltas for their field. Figure 16 shows that for a GC that maintains allocation order placement the percent of common deltas ranges from 20% to 52% across the SPEC JVM98 and SPEC JBB2000 benchmarks suite, a surprising high number. This data shows how often one can correctly predict the value of a reference field using the most common delta for that field; for example, 20% of SPEC JBB2000's references can be correctly predicted using deltas.

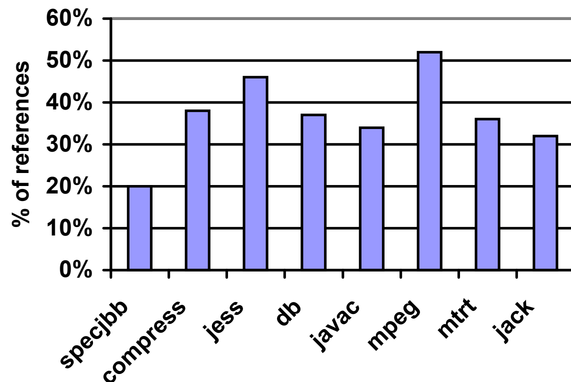

Figure 16. Percent of references with frequent field-level deltas.

7.4 Delinquency Characterization

Figure 17 breaks out total execution cycles stalled due to memory latency according to object age. This includes both high and low-latency stalls as well as non-heap stalls. An object is young until it has survived a garbage collection at which point it matures. For SPEC JBB2000 and db, the two benchmarks with the highest memory stalls, mature objects account for most of the memory stalls. This indicates that objects causing most of the miss latency survive a garbage collection, and are thus available for delta and delinquency analysis by the GC. This further indicates that delinquent objects are available to the GC in case it decides to induce deltas between objects along a delinquent path as it moves them during a GC cycle.

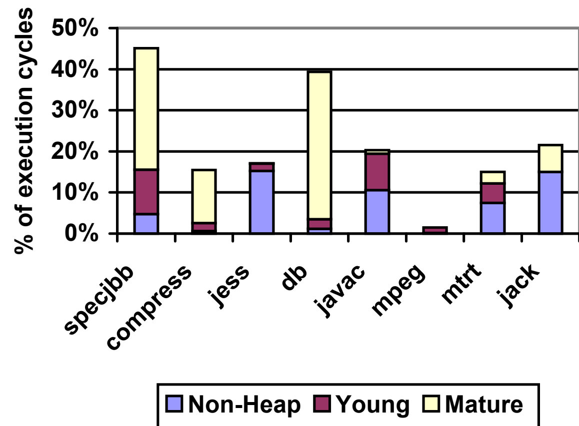

Figure 17. Age of objects causing latency.

8. RELATED WORK

Several authors have recognized the importance of garbage collection techniques to improve memory performance by positioning objects. One of earliest works employed *cdr coding* in the Lisp Machine [14]. White [31] suggested that paging performance should be a primary task of garbage collection. Chilimbi and Larus [5] improved cache line packing by using software to monitor loads and having the GC place objects based on temporal locality. Yefim et al [28] used allocation frequency to identify prolific types and then to place objects based on prolificacy. Wilson et al [32] focused on ameliorating the negative paging effects associated with garbage collection by improving the placement of objects using connectivity. Inagaki et al [18] also recognize the benefits of maintaining allocation order at garbage collection time to maintain deltas between objects.

Early prefetching work concentrated on improving performance in scientific code with densely packed arrays; see [30] for a survey. Prefetching linked data structures requires predicting the access patterns, a challenging problem. Luk et al [23] and Wu et al [34] use stride prefetching to exploit regular patterns of access in linked-list traversals. The literature for hardware stride prefetching is abundant; for example, see Sair et al [26]. Several authors have recently recognized the existence of deltas between loads or between objects that can be used for prefetching. Inagaki et al [18] explore software delta prefetching within loops and Zhou et al [35] explore hardware value prediction.

Helper or speculative threads [20] attempt to prefetch data by forcing cache misses ahead of the worker thread. These techniques can use MS Delta techniques to materialize prefetch addresses in the helper thread without loss of generality of either technique.

Roth and Sohi [25] prefetch linked data structures by augmenting objects with fields containing the addresses of objects to prefetch. MS Delta materializes prefetch addresses using constant deltas compiled into the code rather than augmenting objects with a new field.

Chilimbi and Hirzel [6] predict hot data streams using a finite state machine that represents a program's frequently executed address streams. In contrast, MS Delta concisely models the cache miss behavior of a program using metadata, and predicts

miss addresses using deltas, which allow MS Delta to prefetch addresses that have not been profiled.

Similar to MS Delta, other authors have also used metadata to concisely model memory behavior. Wu et al [33] use types to characterize cache misses. Their characterization of cache misses replaces effective address with the object type, thus achieving a more concise representation to compress traces of data cache misses. Calder at al [4] recognized other forms of metadata that allow concise modeling; for example allocation units managed by 'malloc-like' allocators and memory regions (stack, heap, constant area, etc). Abstracting addresses up to metadata dramatically reduces the amount of information needing analysis, and exposes patterns in address streams.

MS Delta uses hardware monitoring, which has fewer overheads than software monitoring and accurately identifies delinquent loads, which benefit from prefetch. Other authors [23][22] have also used hardware monitoring of cache misses to concentrate prefetch on delinquent loads. In contrast, several authors have used software instrumentation to predict cache misses. Chilimbi and Hirzel [6] used whole program instrumentation to track memory references, using bursty profiling to reduce instrumentation overhead. Chilimbi and Larus [5] reduce instrumentation overhead by tracking only the base address of an object. Wu et al [34] and Inagaki et al [18] used software-based profiling to guide prefetching optimizations in loops, where cache misses are likely to occur.

9. CONCLUSIONS

In this paper, we have presented Mississippi Delta, a novel technique for prefetching linked data structures that closely integrates the hardware performance monitor, the garbage collector's global view of heap and object layout, the type-level metadata inherent in type-safe programs, and JIT compiler analysis. We have shown how Mississippi Delta's dynamic closed loop system abstracts raw addresses and instruction pointers delivered by the hardware up into a concise metadata graph where reasoning can be done at the type level instead of at the raw address level. We have shown how Mississippi Delta guides the JIT analysis in inserting timely prefetches by finding delinquent paths through the metadata graph and calculating deltas. Finally, we have shown how these prefetch techniques result in a 11-14% speedup on the cache miss intensive SPEC JBB2000 benchmark.

Mississippi Delta further expands the garbage collector's role to include observing memory system performance and guiding memory optimizations using global heap properties related to object placement and connectivity. Mississippi Delta demonstrates that researchers should view the garbage collector as an integral part of the dynamic profile feedback loop that produces highly optimized code.

10. ACKNOWLEDGMENTS

The authors wish to thank the members of the StarJIT dynamic compiler and ORP virtual machine teams, Jay Bharadwaj, Dong-Yuan Chen, Michal Cierniak, Marsha Eng, Anwar Ghuloum, Neal Glew, Brian T. Lewis, Vijay Menon, Brian R. Murphy, Bratin Saha, Tatiana Shpeisman, James Stichnoth, and Weldon Washburn, for providing a high-performance Java infrastructure that allowed us to do this research. The authors also thank Jesse Fang for his encouragement and insights, which made this research possible.

11. REFERENCES

[1] Adl-Tabatabai, A., Bharadwaj, J., Chen, D-Y, Ghuloum, A., Menon, V., Murphy, B., Serrano, M., and Shpeisman, T. The StarJIT Compiler: A Dynamic Compiler for Managed Runtime Environments. *Intel Technology Journal*, February 2003.

[2] Adl-Tabatabai, A., Bharadwaj, J., Eng, M., Fang, J., Lewis, B.T., Murphy, B.R., Stichnoth, J., and Cierniak, M. Improving 64-bit Java IPF Performance by Compressing Heap References. In *Code Generation and Optimization (CGO),* 2004.

[3] Cahoon, B., and McKinley, K. Data Flow Analysis for Software Prefetching Linked Data Structures in Java. In *Parallel Architectures and Compilation Techniques (PACT),* 2001.

[4] Calder, B., Krintz, C., John, S., and Austin, T. Cache-Conscious Data Placement. In *Architecture Support for Programming Languages and Operating Systems* (*ASPLOS)*, 1998.

[5] Chilimbi, T., and Larus, J. Using Generational Garbage Collection To Implement Cache-Conscious Data Placement, In *Proceedings of International Symposium on Memory Management (ISMM)*, 1998.

[6] Chilimbi, T., and Hirzel, M. Dynamic Hot Data Stream Prefetching for General Purpose Programs. In *Programming Languages Design and Implementation (PLDI*), 2002.

[7] Chilimbi, T., and Larus, J. Cache-conscious Structure Layout. In *Programming Languages Design and Implementation (PLDI),* 1999.

[8] Chilimbi, T. Efficient Representations and Abstractions for Quantifying and Exploiting Data Reference Locality. In *Programming Languages Design and Implementation (PLDI*), 2001.

[9] Cierniak, M., Eng, M., Glew, N., Lewis, B., and Stichnoth, J. The Open Runtime Platform: A Flexible High-Performance Managed Runtime Environment. *Intel Technology Journal*. http://developer.intel.com/technology/itj/2003/volume07issue01, February 2003.

[10] Collins, J., Wang, H., Tullsen, D., Hughes, C., Lee, Y-F., Lavery, D., and Shen, J. Speculative Precomputation: long-range prefetching of delinquent loads. In *International Symposium of Computer Architecture (ISCA),* 2001.

[11] Dieckmann, S., and Hölze, U. A Study of the Allocation Behavior of the SPECjvm98 Java Benchmarks. In *European Conference on Object-Oriented Programming (ECOOP),* 1999.

[12] Dimpsey, R., Arora, R., and Kuiper, K. Java Server Performance: A Case Study of Building Efficient, Scalable JVMs. *IBM Systems Journal*, 39(1): 151-174, 2000.

[13] Fenichel, R., and Yochelson, J., A Lisp Garbage Collector for Virtual Memory Computer Systems. *Communications of the ACM*, 12(11):611-612, November 1969.

[14] Greenblatt, R. The LISP Machine; Working Paper No. 79, M.I.T. A. I. Lab, Cambridge MA (Nov 1974).

[15] Hirzel, M., Diwan, A., and Hertz, M. Connectivity-Based Garbage Collection. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA),* 2003.

[16] Hudson, R., Moss, J.E.B., Subramoney, S., and Washburn, W. Cycles to Recycle: Garbage Collection on the IA-64. In *International Symposium on Memory Management (ISMM),* 2000.

[17] Hudson, R., and Moss, J.E.B. Copying Garbage Collection without stopping the world. *Concurrency and Computation: Practice and Experience* 15(3-5), 2003.

[18] Inagaki, T., Onodera, T., Komatsu, H., and Nakatani, T. Stride Prefetching by Dynamically Inspecting Objects. In *Programming Languages Design and Implementation (PLDI),* 2003.

[19] Intel Corp. IA-64 Application Developers Architecture Guide. May 1999.

[20] Kim, D., Liao, S., Wang, P., Cuvillo, J., Tian, X., Zou, X., Wang, H., Yeung, D., Gikar, M., and Shen, J. Physical Experimentation with Prefetching Helper Threads on Intel's Hyper-Threaded Processors. In *Code Generation and Optimization (CGO),* 2003.

[21] Lam, M., Wilson, P., and Moher, T. Object Type Directed Garbage Collection to Improve Locality. In *International Workshop of Memory Management (IWMM)*, 1992.

[22] Lu, J., Chen, H., Fu, R., Hsu, W-C, Othmer, B., Yew, P-C, and Chen, D-Y. The Performance of Runtime Data Cache Prefetching in a Dynamic Optimization System. In *International Symposium on Microarchitecture (MICRO),* Dec. 2003.

[23] Luk, C-K, Muth, R., Patil, H., Weiss, R., Lowney, P.G., and Cohn, R. Profile-guided post-link stride prefetching. In *International Conference on Supercomputing (ICS),* 2002.

[24] Jones, R., and Lins, R. Garbage Collection, Algorithms for Automatic Dynamic Memory Management. John Wiley and Sons, 1996.

[25] Roth, A., and Sohi, G. Effective Jump Pointer Prefetching for Linked Data Structures. In *International Symposium on Computer Architecture (ISCA),* 1999.

[26] Sair, S., Sherwood, T., and Calder, B. Quantifying Load Stream Behavior. In *International Symposium on Microarchitecture (MICRO),* 2002.

[27] Shuf, Y., Serrano, M.J., Gupta, M., and Singh, J.P. Characterizing the Memory Behavior of Java Workloads: A Structured View and Opportunities for Optimizations. In *Conference on Measurement and Modeling of Computer Systems (SIGMETRICS),* 2001.

[28] Shuf, Y., Gupta, M., Franke, H., Appel, A., and Singh, J.P. Creating and Preserving Locality of Java Applications at Allocation and Garbage Collection Times. In *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, 2002.

[29] Standard Performance Evaluation Corporation. http://www.specbench.org.

[30] VanderWiel, S. and Lilja, D. Data Prefetch Mechanisms. In *ACM Computing Surveys*, 2000.

[31] White, J.L. Address/Memory Management for a Gigantic LISP Environment or, GC Considered Harmful. Conference Record of the 1980 LISP Conference, Stanford University, Palo Alto California.

[32] Wilson, P.R., Lam, M.S., and Moher, T.G. Effective "Static-graph" Reorganization to Improve Locality in Garbage-Collected Systems. In *Programming Languages Design and Implementation (PLDI),* 1991.

[33] Wu, Q., Pyatakov, A., Spiridonov, A., and August, D. Exposing Memory Access Regularities for Effective Memory Profiling. In *Code Generation and Optimization (CGO),* 2004.

[34] Wu, Y., Serrano, M.J., Krishnaiyer, R., Li, W., and Fang, J. Value-Profile Guided Stride Prefetching for Irregular Code. In *Compiler Construction Conference (CC),* 2002.

[35] Zhou, H., Flanagan, J., and Conte, T. Detecting Global Stride Locality in Value Streams. In *International Symposium on Computer Architecture (ISCA),* 2003.