Effective Prefetch for Mark-Sweep Garbage Collection *

Robin Garner Stephen M Blackburn Daniel Frampton

Department of Computer Science
Australian National University
Canberra, ACT, 0200, Australia
{Robin.Garner,Steve.Blackburn,Daniel Frampton}@anu.edu.au

Abstract

Garbage collection is a performance-critical feature of most modern object oriented languages, and is characterized by poor locality since it must traverse the heap. In this paper we show that by combining two very simple ideas we can significantly improve the performance of the canonical mark-sweep collector, resulting in improvements in application performance. We make three main contributions: 1) we develop a methodology and framework for accurately and deterministically analyzing the tracing loop at the heart of the collector, 2) we offer a number of insights and improvements over conventional design choices for mark-sweep collectors, and 3) we find that two simple ideas — edge order traversal and software prefetch — combine to greatly improve garbage collection performance although each is unproductive in isolation.

We perform a thorough analysis in the context of MMTk and Jikes RVM on a wide range of benchmarks and four different architectures. Our baseline system (which includes a number of our improvements) is very competitive with highly tuned alternatives. We show a simple marking mechanism which offers modest but consistent improvements over conventional choices. Finally, we show that enqueuing the *edges* (pointers) of the object graph rather than the *nodes* (objects) significantly increases opportunities for software prefetch, despite increasing the total number of queue operations. Combining edge ordered enqueuing with software prefetching yields average performance improvements over a large suite of benchmarks of 20-30% in garbage collection time and 4-6% of total application performance in moderate heaps, across four architectures.

* This work is supported by ARC DP0452011, ARC DP0666059 and Intel. Any opinions, findings and conclusions expressed herein are those of the authors and do not necessarily reflect those of the sponsors.

ISMM'07, October 21–22, 2007, Montréal, Québec, Canada.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors—Memory management (garbage collection)

General Terms Design, Performance, Algorithms

Keywords Java, Mark-Sweep, Software Prefetch

1. Introduction

The mark-sweep algorithm dates back to 1960 [10] and in a variety of forms it remains widely used today. The basic algorithm is simple. The collector starts with a set of roots and traverses the graph of objects, making a *mark* against each object encountered in the transitive closure, to indicate that it is live. The collector then performs a *sweep* of the heap, reclaiming all unmarked objects. The most performance-critical component of the algorithm is the tracing loop which performs the transitive closure over the graph of live objects. In this paper we present a detailed examination of the performance of this loop and propose a number of optimizations which lead to signicant improvements in garbage collection performance and total application running time.

We begin by describing a methodology and framework for analyzing the performance of the tracing loop. Our approach is at each collection, to capture the exact visit order of a trace of the heap, and then during the collection, *replay* that trace, each time measuring the cost of performing different aspects of the tracing loop, including touching, marking, scanning, and tracing through each object. By suitably combining the replay operations, we are able to tease apart tracing costs and examine in detail the pros and cons of standard variations on the mark-sweep trace algorithm. Since we piggyback this process on a conventional mark-sweep collector, we are able to analyze actual costs in the context of real applications. Our focus here is on the tracing loop of a mark-sweep collector, but our methodology and framework could be extended to copying collectors. We will make our framework publicly available.

Using *replay tracing*, we draw a number of insights. First, we find that performance is limited by the memory costs of two distinct phases; marking and scanning, corresponding to temporally distinct visits to the object and its metadata (each of which may be co-located or spatially disjoint). This observation leads to our new approach to prefetching, described

later in the paper. Second, we confirm the conventional wisdom that densely packed mark bits improve the locality of the mark phase. However, we find that although performance improves when marking is done in isolation, this advantage is completely neutralized by the cache-displacing proprieties of other parts of the tracing loop. Third, we measure the synchronization costs associated with different mark schemes. Finally, we propose and evaluate a simple hybrid marking scheme that out-performs each of the primary alternatives.

Having analyzed and optimized the performance of a basic tracing loop, we then explore opportunities for prefetching values in the tracing loop. The conventional tracing loop visits each object and/or its metadata twice in each collection, corresponding to the two primary costs identified in our replay tracing analysis. A trivial variation on the conventional tracing algorithm is to enqueue an object for each *edge* in the graph rather than an object for each *node* in the graph. This variation, which we call *edge enqueuing*, while performing exactly the same number of marks and scans, obviously increases queue work from N nodes being enqueued to E edges being enqueued, where $N \leq E$. We find that the modest locality advantage associated with edge enqueuing outweighs the queuing overhead. However, edge enqueuing greatly increases opportunities for software prefetching [5, 6]. We then perform a systematic analysis of the effect of prefetching on four architectures using both queuing algorithms and two metadata mechanisms. We find that by combining our hybrid metadata scheme with edge enqueuing and prefetching, we are able to improve garbage collection performance by as much as 25% and total application performance by up to 15% in space-constrained heaps, on average, across a large benchmark suite. This is a substantial improvement over prior work, which was only able to show modest average improvements on a small number of benchmarks [5, 6]. We compare directly with the approach used in the prior art in Section 7. Edge enqueuing has been previously identified in the literature [9], but to our knowledge has not been used and has never been carefully studied.

We finish the paper with insights and lessons we learned when porting our prefetch mechanisms to a different VM code base. We hope these will help anyone wishing to implement prefetching in their own code base.

2. Related work

The basic mark-sweep algorithm has been continuously refined since 1960 [10]. Our work builds directly upon prior work on lazy sweeping [5] and prefetching [5, 6].

Boehm [5] first describes and evaluates *lazy sweeping* in the context of the Boehm-Demers-Weiser (BDW) collector. Instead of sweeping the whole heap immediately after each mark phase, the GC only sweeps completely free (unmarked) blocks of memory, and the remainder are lazily swept on demand by the allocator. This approach is effective when mark bits are maintained on the side, allowing free blocks to be cheaply identified. Boehm saw up to 17% net performance win from lazy sweeping.

In the same paper, Boehm is the first to apply software prefetching to garbage collection. He introduces a prefetch strategy called *prefetch on gray*, where an object is prefetched upon insertion into the mark stack. This strategy is somewhat effective in a heap implementation using a mark bitmap on the side. Boehm saw speedups of up to 17% in synthetic GC-dominated benchmarks and 8% on a real application (ghostscript). However, when mark bit metadata is embedded in the object header, objects are guaranteed to be in cache when they are pushed on the mark stack, obviating the need for a prefetch. Prefetch on gray has two key limitations: many items are prefetched too soon, and by the time the depth-first search pops the stack back to the item, it is no longer in cache; and secondly, many items are prefetched too late, since the last object pointed to by any given object is accessed immediately after it is pushed on the stack and prefetched, allowing no time for the prefetch to take effect. Boehm measures costs using profiling, and reports results for a small number of C benchmarks.

Cher et al. [6] build on Boehm's investigation, using simulation to measure costs and explore the effects of prefetching in the BDW collector. They find that when evaluated across a broad range of benchmarks, Boehm's prefetch on gray strategy attains only limited speedups under simulation, and no noticeable speedups on contemporary hardware. Cher et al. introduce the buffered prefetch strategy that we also adopt (see Figure 1). Buffered prefetching observes that the LIFO discipline used in the mark stack when performing depth-first search is unsuitable for prefetching as the pattern of future accesses in a LIFO structure is hard to predict. They recover predictability by placing a small FIFO prefetch buffer between the mark stack and the tracing process. When the tracing loop pops the next entry from the mark stack (line 5), a prefetch is issued on its referent (line 6) and the entry is inserted at the tail of the prefetch FIFO (line 7). The entry at the head of the prefetch FIFO is then selected for scanning (line 8). The depth of the FIFO defines the prefetch distance.

Cher et al. validate their simulated results using a PowerPC 970 (G5), almost identical to the system that we obtain our results on. They obtain significant speedups on benchmarks from the jolden suite, but less impressive results for the SPECjvm98 suite, with their best result being 8% on _202_jess, and 2% on _213_javac. All of these results were

```
1  void add(Address item)
2    stack.push(item)
3
4  Address remove()
5    Address pf = stack.pop()
6    pf.prefetch()
7    fifo.insert(pf)   // FIFO buffer allows
8    return fifo.pop() // prefetch time
```

Figure 1. The FIFO-Buffered Prefetch Queue [6].

achieved in very space-constrained heaps; about 1.125× the minimum heap size, sufficiently small that GC time is a large fraction of total time, amplifying the effect of any GC improvements. In Section 7 we show that by combining Cher et al.'s approach with edge ordered enqueuing, we see consistent, sizable performance improvements across a large number of benchmarks.

Hicks et al. [7] use a technique similar to our replay collector to study the costs of copying garbage collection mechanisms. Their tool, *Oscar*, captures snapshots of heaps for *offline* analysis, and is language-independent. They use Oscar to analyse the performance of mechanisms for copying collectors. Our approach uses online analysis and measures multiple collections across the lifetime of a benchmark.

3. Replay Tracing

One of the contributions of this work is a methodology for detailed performance analysis of the tracing loop. The remainder of this paper builds on the insights we gained through this methodology.

The tracing loop is the most performance-critical element of many modern garbage collection algorithms. Previous work has used sample-based profiling [5] and simulation [6] to analyze the mechanism, each of which have shortcomings. Simulation has the disadvantage of long running times, making it difficult to use benchmarks with a significant memory load. It also limits the available target architectures to those supported by the simulation packages available, and is entirely dependent on the fidelity of the simulation infrastructure with respect to real hardware. Sample-based profiling limits analytical flexibility: in order to sample a collector, it must be a working real-world collector; this makes it time consuming to experiment with algorithmic and implementation variations, and very hard to tease apart the contributions of various details of the implementation. Furthermore, sampling is inherently probabilistic rather than exact.

3.1 The Replay Tracing Framework

We call our solution *replay tracing*. Our approach allows us to experiment with a great many variations on the tracing loop, and by using timers and hardware performance counters we can analyze the various costs in great detail. Our initial implementation uses a modified mark-sweep garbage collector. The system works by modifying the garbage collector so that at *every* collection, in addition to performing collection work, the collector gathers a trace of visited objects and then *replays* and measures that trace multiple times for analytical purposes.

At each collection, we first trace the live objects in the heap, exactly as the unmodified mark-sweep collector would, except that whenever an object is processed (popped from the mark stack), we record a pointer to the object in a *replay buffer*. The replay buffer gives us a record of the objects accessed during the trace, in exactly the order in which they are accessed. We then use the replay buffer to execute multiple replay *scenarios*. Each scenario performs different operations on each object in the buffer, in exactly the same order each time.

```
1  for p in root-set
2    mark(p)
3    queue.add(p)
4
5  while !queue.isEmpty()
6    obj = queue.remove()
7    gcmap = obj.getGcMap()
8    for p in gcmap.pointers()
9      child = p.load()
10     if child != null
11       if child.testAndMark()
12         queue.add(child)
```

Figure 2. The Standard Tracing Loop

For example, a scenario which just performs a mark operation on each object allows us to isolate the cost of marking and thus evaluate different marking strategies. Likewise, a scenario could just touch each object, scan each object, or perform a complete mark, scan and trace of each object. By carefully constructing scenarios and measuring their costs, we can break down the contributions of the various elements of the tracing loop and systematically explore alternatives.

In order to minimize distortion of results due to cache pollution, we flush the cache between each use of the replay buffer by reading a large (8MB) table sequentially. We also need to take care when setting mark bits—their state must be flipped after each phase in which they are changed. We repeat all of this—creating the replay buffer, replaying scenarios, and flushing the cache—each time a collection is triggered. We aggregate results across collections so that at the end of the program we have measurements for each scenario with respect to the entire GC workload of the program.

3.2 Replay Scenarios

We now describe a number of example replay scenarios, including those we use in our subsequent analysis.

Figure 2 shows pseudo-code for the standard mark-sweep tracing loop, from which most of the scenarios in Figure 3 are derived. We have described the queue operations that maintain the *work list* abstractly as *add* and *remove*. If a collector implements these operations as push and pop (LIFO–a stack), it will perform a depth-first traversal of the heap graph, while tail-insert and pop (FIFO–a queue) will produce a breadth-first traversal. The major components of the basic tracing loop in Figure 2 are: a) queuing costs (lines 3, 5, 6 & 12), b) accessing the pointer map for each object (line 7), c) enumerating pointers (line 9), and d) the test and mark of referenced objects (line 11).

In order to evaluate the relative costs of these operations, we use the scenarios shown in Figure 3. The *harness cost* scenario measures the cost of the replay buffer and thus the overhead of our framework. The *queue cost* scenario

measures the approximate queue management cost of the standard tracing loop, alternately inserting N items onto the queue and popping N-1 items (we use N = 10) so that the effects of growing and shrinking the queue across block boundaries are measured.

The *object touch* scenario measures the cost of accessing the first word of each reachable object in the heap. The GC map, describing the location of any pointers within the object, is typically only found via touching (and possibly dereferencing) the header of the object to be scanned. The *scan* scenario measures the cost of visiting each object in the heap, iterating its GC map, and loading each pointer field. Comparing the cost of this scenario with a scenario that simply visits each heap object can tell us about the cost of the operations that have been added to the scenario. The *trace* scenario adds to the *scan* scenario a dereference of every non-null child of the scanned object. Finally, the *mark* scenario performs a mark on each non-null child of every object in the replay buffer. The *mark* scenario thus performs all of the work of the standard tracing loop except the final enqueuing operation; compare lines 7 to 11 in Figure 2 with lines 3 to 7 in Figure 3(f).

```
1 for item in buffer
2   item.load()
```

(a) Harness Cost Scenario

```
1 i=0
2 for item in buffer
3   queue.add(item.load())
4   if ++i == N
5     while --i > 0 && !queue.isEmpty()
6       queue.remove()
7     --i
8 while !queue.isEmpty()
9   queue.remove()
```

(b) Queue Cost Scenario

```
1 for item in buffer
2   item.load().load()
```

(c) Object Touch Scenario

```
1 for item in buffer
2   obj = item.load()
3   gcmap = obj.getGCMap()
4   for p in gcmap.pointers()
5     child = p.load()
```

(d) Scan Scenario

```
1 for item in buffer
2   obj = item.load()
3   gcmap = obj.getGCMap()
4   for p in gcmap.pointers()
5     child = p.load()
6     if (child != null)
7       child.load()
```

(e) Trace Scenario

```
1 for item in buffer
2   obj = item.load()
3   gcmap = obj.getGCMap()
4   for p in gcmap.pointers()
5     child = p.load()
6     if (child != null)
7       child.testAndMark()
```

(f) Mark Scenario

Figure 3. Replay Scenarios

One might be tempted to assume that the change in workload of two scenarios where the second scenario strictly adds work to the first one can be measured by simple subtraction. This is true for wall clock time (at least in our observations), but not necessarily true for other measures such as cache misses. In fact, the *object touch* scenario (Figure 3(c)) that visits the first word of each object in the heap actually has a higher L2 miss rate than the *scan* scenario (Figure 3(d)), on some architectures, even though the scan scenario does strictly more work (`obj.getGCMap()` involves at least one `load()`). We suspect that the additional work in the scan scenario is allowing a hardware prefetch mechanism time to take effect, eliminating some of the misses seen by the first scenario.

4. Key Mark-Sweep Design Choices

Although the basic mark-sweep algorithm is well documented and well understood, there are several design choices that have the potential to significantly affect performance [14, 9, 5]. These include: a) the *allocation* mechanism and free-list structure, b) the mechanism for storing *mark state*, c) the technique used to *sweep* the heap, and d) the structure of the collector's *work queue*.

Allocation We use the same allocation mechanism and free-list structure for all mark-sweep configurations we evaluate in this paper. The free list is structured as a *two level segregated fit* free-list structure [14]. The allocator divides memory into coarse grain *blocks* of which there are several distinct sizes. When required, it assigns individual blocks to a single *object size class* and divide them into *N* equal sized cells. The allocator always satisfies requests by the first entry on the free list for the smallest size class that is large enough. This approach reduces worse case fragmentation while ensuring a fast simple allocation path as any request is always satisfied by the head of the free-list.

Mark state The collector's mark phase depends on being able to store mark state for each object. Mark state is typically stored either as a field within the *header* of each object, or in a dense *bitmap* on the side. We refer to these two approaches as *header* and *side bitmap* respectively. The side bitmap has a potential locality advantage because it is far more tightly packed than mark bits embedded within headers of objects which are dispersed throughout the heap. An important optimization for *header* state is to change the sense of the mark bit at each garbage collection, which avoids having to reset all mark bits after every collection. Side bitmaps

can be trivially and cheaply zeroed in bulk, thereby avoiding this issue.

An alternative to side bitmaps is a *bytemap*, a byte-grained data structure on the side. Bytemaps trade an $8\times$ space overhead to avoid synchronizing on each set operation (if we assume support for atomic byte-grained stores). In a parallel collector, it is possible to lose updates in a race to set a bit unless the bit can be set atomically. We include both a side bitmap and the header approach for comparison throughout our evaluation. We also evaluate the overhead of synchronization on the bitmap.

Sweep A classic mark-sweep collector will *sweep* the entire heap at the end of each collection, identifying unmarked memory and returning it to free lists. The sweep comprises two components: examining each block's metadata to identify marked objects, and populating the free lists with any freed memory. Scanning a side bitmap will easily reveal blocks which are entirely free, allowing them to be freed up entirely, avoiding the construction of a free list. Building freelists typically involves a full walk through each block, which is a memory-bound operation. Sweeping the entire heap at each collection is known as *eager sweeping*.

Boehm [5] noted significant advantages to *lazy sweeping*. A lazy sweeper sweeps blocks only when they are required by the allocator. This has two significant advantages. First, it saves sweeping work for many blocks which are unchanged from collection to collection. Second, free list construction occurs immediately prior to use of the cells, which has measurable temporal locality benefits. Blocks which are entirely free can be identified by examining a side bitmap at the end of a collection. Lazy sweeping is therefore normally used with a side bitmap.

In the process of teasing apart the various design choices for mark-sweep collectors, we wanted to explore lazy sweeping with header mark bits. As we show in Section 6.3 this combined approach is quite effective, although to our knowledge has not been explored before. In order to free unused blocks eagerly while lazily sweeping partially used blocks, we developed *hybrid marking*, which uses mark bits in object headers and a single byte of side metadata for each block. Each block's side metadata is set whenever an object within the block is marked. Any block with an unmarked metadata byte is completely free, and may be eagerly reclaimed at collection time. This way we are able to combine header metadata with lazy sweeping.

Recall that one of the two advantages of lazy sweeping is that unmarked objects only lazily make their way onto free lists. When a side bitmap is used, this delay is of no consequence beyond the obvious saving of work. Recall however, that header mark state changes sense at each collection. Therefore an object with header mark state will not be recognized as unmarked if it is swept an even number of collections since it was last marked. This situation can never lead to a live object being collected since every live object is visited at every collection, but it can lead to "floating garbage." Of course this situation does not arise with a side bitmap since the sense of the mark bit does not alternate.

We compared two basic solutions to the problem of floating garbage when lazy sweeping. One approach is to simply sweep the unswept portion of the heap at the beginning of each collection cycle. This approach will still see the locality advantages of lazy sweeping, but loses the advantage of avoiding redundant sweeping. Our second approach uses multiple bits to record mark state within each object header and stores the mark state as the collection number modulo the largest number that can be stored in the header bit field. With a bit field size of 8, we reduce the worst case amount of floating data by $1/256$. We could not measure any floating data in practice. Note that the data will only float in the case that it is finally swept by the mutator exactly $N \times 256$ collections since it was last marked without that block being swept in any of the previous 255 collections. Section 6.3 evaluates these approaches.

Work queue Section 3.1 described the basic tracing loop (Figure 2), and pointed out that the collector's work queue could be maintained in either LIFO or FIFO disciplines, leading to depth-first or bread-first traversals of the heap respectively. In a copying collector the traversal order affects the relative position of objects after collection, and therefore impacts application locality [8, 15]. In such collectors, depth first orders are generally accepted as more efficient [8, 15]. Although there is little performance impact in a non-copying collector, we use a LIFO discipline since it is widely used.

5. Methodology

We now describe the methodology used throughout the rest of the paper, including the virtual machine we use, and the hardware platforms and benchmark suites we evaluate.

Hardware Platforms We use four different hardware platforms in our analysis, which are described in detail in Table 1. The systems were running Linux 2.6.x kernels with Debian unstable. All CPUs were operated in 32-bit mode, although the 3 64-bit capable processors (all except the Pentium M) were running 64-bit kernels. Each of the CPUs we use supports hardware performance counters, which we access using the perfctr [11] library. We count the following metrics:

RI Retired instructions

L1 Level 1 D-cache miss rate

L2 Level 2 cache miss rate

DTLB Data TLB miss rate

Jikes RVM and MMTk We use Jikes RVM and MMTk for all of our experiments. Jikes RVM [1] is an open source high performance Java virtual machine (VM) written almost entirely in a slightly extended Java. Jikes RVM *does not have*

Platform	Clock frequency	Main memory	L1 cache	L2 cache
AMD Athlon 64 3500+	2.2GHz	400MHz DDR2	64KB 64B 2-way	512KB 64B 16-way
Pentium-4 *Prescott*	3.0GHz	533MHz DDR2	16KB 64B 8-way	1MB 64B 16-way
Pentium-M *Dothan*	2.0GHz	533MHz DDR2	32KB 64B 4-way	2MB 128B 8-way
PowerPC 970 *G5*	1.6GHz	333MHz DDR	32KB 128B 2-way	512KB 128B 8-way

Table 1. Hardware Platforms

a bytecode interpreter. Instead, a fast template-driven compiler produces machine code when the VM first encounters each Java method. The adaptive compilation system then judiciously optimizes the most frequently executed methods [2]. Using a timer-based approach, it schedules periodic interrupts. At each interrupt, the adaptive system records the currently executing method. Using a threshold, it then selects frequently executing methods to optimize. Finally, the optimizing compiler thread re-compiles these methods at increasing levels of optimizations. All of our experiments were run using Jikes RVM's *replay compilation* feature, which provides deterministic hot method compilation using adaptive compilation profiles gathered on previous runs. We used 'FastAdaptive' builds, which remove assertion checks and *fully optimize* all code for the virtual machine (and hence the garbage collector). Experiments were performed 5 times in standalone mode, and the fastest of 5 runs chosen. MMTk is Jikes RVM's memory management sub-system. It is a composable memory management toolkit that implements a wide variety of collectors that reuse shared components [3]. We use MMTk's mark-sweep collector (*MarkSweep*).

Credibility of MMTk As An Experimental Platform The previous work on software prefetching in garbage collectors was performed using the Boehm-Demers-Weiser (BDW) collector, and most recently in the context of gcj, the GNU Java compiler, which uses the BDW collector. We have compared the performance of the tuned version of MMTk used in this paper with gcj using the standard Jikes RVM GC performance benchmark, FixedLive. The results of this comparison are given in Table 2, with the numbers representing a tracing rate in MB/s. This shows that MMTk outperforms gcj by between 11% and 37% on the x86 platforms, but lags gcj on the PPC by 33%. The tracing performance of gcj on the PPC is considerably faster than on the x86 architectures, despite the fact that on most benchmarks the PPC 970 is slower than the other machines we used. We believe that the prefetching in the BDW collector is much more effective on the PPC, and that this allows it to outperform MMTk with no prefetching.

In this comparison we use the same baseline MMTk MarkSweep configuration that is used throughout the rest of the paper—the node-enqueuing collector with a mark bit in the object header (see Section 7 for details). We use gcj version 4.0.2, and compile using the '-O2' flag. The BDW collector is a conservative (ambiguous roots) collector, thus some of the performance difference may be due to it being unable to take advantage of the fast object scanning techniques used by MMTk. Our goal here, however, is simply to assert that MMTk is a well tuned platform and a credible basis for experimentation, relative to prior work.

Platform	Jikes RVM	GCJ 4.0
AMD Athlon64 300+	266	194
Pentium-4 *Prescott*	265	214
Pentium-M *Dothan*	210	189
PowerPC 970 G5	218	291

Table 2. GC Throughput in MB/s

Benchmarks We use the DaCapo and SPECjvm98 benchmark suites, and pseudojbb. The DaCapo suite [4] is a recently developed suite of non-trivial real-world open source Java applications. We use version beta051009. pseudojbb is a variant of SPEC JBB2000 [12, 13] that executes a fixed number of transactions to perform comparisons under a fixed garbage collection load.

6. Analysis of Tracing Costs

We now use the replay tracing framework described in Section 3.1 to evaluate the dominant costs of the mark-sweep tracing loop shown in Figure 2. We perform our measurements using Jikes RVM and MMTk with DaCapo and SPEC benchmarks, and the AMD and Pentium-M architectures, as described in Section 5.

Framework Overhead We begin by measuring the overhead of the replay tracing harness. We compare the cost of the simple *harness cost* scenario shown in Figure 3(a) against the cost of a full collection of the heap. We found that in terms of wall clock time, the cost of the harness was less than 2% that of a full heap collection. Hardware performance counters revealed negligible L1, L2 and DTLB misses, while the harness accounted for 6% of the instructions executed.

6.1 Experiments

To analyze the cost of the mark-sweep tracing loop, we use the methodology described in Section 3, evaluating a series of incrementally more complex replay scenarios to build up a picture of total costs. The replay scenarios we use are as follows:

enq-deq Enqueue/dequeue operations in the ratio 10:9, as reflected by the *queue cost* scenario in Figure 3(b). This identifies the cost of the queuing mechanism on which the standard tracing loop is based. All of the subsequent scenarios do not use a queue, as they are driven by the replay buffer, which captures and replays an exact object visit order (see Section 3).

touch Load the first word of every object pointed to by the trace buffer, as reflected by the *touch* scenario in Figure 3(c). This identifies the cost of touching every live object in the heap. In the standard tracing loop, the first step after obtaining a new reference to work on is to fetch the GCmap for the object (line 7 of Figure 2). In Jikes RVM (and many other Java virtual machines) this involves fetching a per-class data structure pointed to by a word in the object's header. By loading the first word of each object, we access memory in a pattern very similar to that of the first action in fetching a GC map. Note that this scenario does not place any dependencies on the load, so the costs of the loads may be understated by wall clock time. However, this scenario will help understand the locality properties of a full heap trace.

scan Scan every object in the trace buffer, loading the value of its pointer fields, as shown in Figure 3(d). This builds on the *touch* operation by using the GC map to enumerate the pointer fields in the object and load the value of each pointer.

trace Trace 'through' every object in the trace buffer, *dereferencing* each of its non-null pointer fields, as shown in Figure 3(e). This builds on the *scan* operation by accessing the word pointed to by each pointer field in the object, i.e. touching each object referenced by the current object. In an implementation where mark bits are kept in the header of an object, this operation will access memory in an analogous pattern.

mark Perform `testAndMark()` on every non-null child of every object in the replay buffer. If mark state is implemented in the header (see Section 4), this scenario only differs from the *trace* scenario by using a `testAndMark()` on each object's header, rather than a `load()`. If mark state is implemented in a bitmap on the side, this scenario will not touch the child object, and therefore is similar to the *scan* scenario, differing only in that it touches the side bitmap associated with the child. In either case, this scenario differs only from the full tracing loop in that unmarked pointers are not enqueued for later tracing.

Another cost that we evaluate is performing the mark operation directly on every object in the replay buffer. While this does not directly correspond to any single operation in the trace loop, it provides an interesting point of comparison between different implementations of marking.

6.2 Tracing Costs: Results

We now present a detailed analysis of the tracing costs for two orthogonal variations on the standard marking mechanism: a) mark state is either in the object *header* or in a *side* bitmap; and b) mark state is set with a simple store (*unsync*) or with architecture-specific atomic update instructions (*sync*). Table 3 shows elapsed times for each of the scenarios with each of the four header and synchronization variants, on the Pentium-M and AMD platforms, with all data normalized to the *header*/*sync* time on the respective platform. Recall that the *mark* scenario only differs from the full tracing loop in that it does not enqueue the marked child (compare Figure 2 lines 7 to 11 and Figure 3(f) lines 3 to 7). The *mark* scenario thus provides a reasonable baseline. Table 5 includes and expands upon this data by providing retired instructions (RI), L1 cache misses, and L2 cache misses. In Table 5 the data is normalized against the *mark* scenario for each respective implementation variant (i.e. each column is normalized to the bottom row).

(a) Pentium-M

	Header		Side	
	Sync	Unsync	Sync	Unsync
Traverse	0.02	0.03	0.02	0.03
Enq-deq	0.10	0.10	0.10	0.10
Touch	0.15	0.15	0.15	0.15
Scan	0.40	0.40	0.39	0.40
Trace	0.59	0.59	0.58	0.59
Mark	**1.00**	0.84	0.98	0.83

(b) AMD Athlon64

	Header		Side	
	Sync	Unsync	Sync	Unsync
Traverse	0.02	0.02	0.02	0.02
Enq-deq	0.11	0.11	0.11	0.11
Touch	0.14	0.14	0.14	0.14
Scan	0.46	0.46	0.46	0.45
Trace	0.63	0.63	0.64	0.63
Mark	**1.00**	0.89	1.00	0.88

Table 3. Comparative Cost of Various Scenarios for Four Design Points, Normalized to Header/Synchronized.

(a) Pentium-M

	Header		Side	
	Sync	Unsync	Sync	Unsync
Time	0.47	0.35	0.37	0.25
L1 Misses	0.50	0.50	0.36	0.36
L2 Misses	0.53	0.58	0.29	0.32
RI	0.27	0.25	0.29	0.27

(b) AMD Athlon64

	Header		Side	
	Sync	Unsync	Sync	Unsync
Time	0.46	0.27	0.38	0.22
L1 Misses	0.52	0.53	0.40	0.40
L2 Misses	0.54	0.54	0.41	0.41
RI	0.27	0.25	0.28	0.26

Table 4. Cost of The Mark Mechanism Alone for Four Design Points, Each Normalized to Cost of Entire Mark Scenario.

A number of observations can be drawn from Tables 3 and 5:

1. Synchronization accounts for approximately 17% of the cost of tracing the heap on a Pentium-M, and 11% on the AMD Athlon. This is evident when comparing normalized *mark* times for *sync* and *unsync* columns in Table 3. Unsurprisingly, this shows that designing data structures

to avoid synchronization is a worthwhile goal, given current hardware implementations.

2. The cost of the *mark* operation between *header* and *side* mark state implementations shows no significant difference when measured in the context of the tracing loop. Compare *header* and *side* columns in Table 3: there is no measurable difference on either platform when the operation is synchronized, and *side* is 1% slower on the Pentium-M and 1% faster on the AMD when unsynchronized. This result is discussed below.
3. The overhead of queue management (enq/deq) is at most 13% of the running time of the tracing loop (Table 5).
4. *Touch* data in Table 5 shows that the initial visit of an object header accounts for 14–18% of time, but 45–60% of cache misses. This is perhaps accounted for by a lack of any dependency on the load, and combination of compiler instruction scheduling and out-of-order execution overlapping the initial fetch of an object with the mark operation on the previous object. This result helps to illuminate the limited success that other approaches have had with prefetch strategies that target this aspect of the tracing loop—even halving these misses would not yield a significant performance gain.
5. Comparing *scan* and *touch* L2 numbers in Table 5 shows that cache miss rates are non-monotonic; scenarios that perform more work sometimes have lower cache miss rates than simpler scenarios. Note that time and RI numbers remain monotonic. We suspect that this is accounted for by hardware prefetching—in particular the *scan* scenario performs a great many register/ALU operations in between memory fetches, allowing time for prefetched cache lines to arrive.

6.3 Evaluating Mark State Implementations

One of the reasons why mark state is often stored in side bitmaps rather than in object headers is the locality advantage of greatly increased density of the mark state (Section 4 and [5]). It seems intuitively likely that a denser data structure will lead to lower cache misses than a scheme where mark bits are held in object headers and distributed over the whole heap. However, comparing the numbers in Table 3 shows that there is no significant performance difference. The overhead of synchronization dominates, and when synchronization is not used, the difference is only 1%, with one result in either direction on the two architectures examined.

To validate this result, we also measured the performance of a scenario that simply performs the `testAndMark()` operation on each object in the replay buffer. These results are shown in Table 4 and are normalized to the entire *mark* scenario. As expected, this shows significant differences between the header and the side bitmap implementation, and as expected, both L1 and L2 cache miss rates decrease markedly with a side bitmap.

This result seems to confirm the intuition that a side bitmap provides a real locality advantage, but it contradicts the results in Table 3. We suspect that the discrepancy can be accounted for by the cache-displacing properties of the remainder of the tracing loop. The basis for the locality argument is that when marking an object, the required cache line will already be in cache with some probability which increases greatly as the metadata is densely packed into cache lines (spatial locality is *greatly* amplified by the dense bitmap). This argument depends on subsequent marks to the same line being relatively near to each other temporally, which is somewhat true when the mark is considered in isolation. However, when the mark is examined in the context of the entire tracing loop, subsequent marks to the same cache line will on average be much further apart in terms of memory accesses due to the significant memory activity of the remainder of the tracing loop (particularly due to the scan). This analysis suggests that any locality advantage due to dense metadata is almost entirely lost due Amdahl's law and the predominance of memory activity in the scan portion of the trace loop, explaining the results in Table 3.

7. Software Prefetching

We demonstrate in the previous section that poor locality is the principal bottleneck to the performance of the tracing loop. Because of the significant miss penalties imposed by modern architectures, it is clear that improving the cache behavior of the trace will be fruitful in terms of improving overall trace performance.

Motivation for Prefetching Broadly, there are two main techniques that can be used to improve memory behavior of software. The first is to minimize memory footprint by maximizing data density and thereby increase the probability that requests for data are served from the fastest levels of cache. This typically involves trying to combine spatial and temporal locality of data operations. An example of this is the use of side bitmaps for mark state, as discussed in the previous section. The second technique is to detect ahead of time that an item of data will be required and use a *software* prefetch instruction to *prefetch* the data into the processor's cache. The potential benefits of this approach are limited by the ability to predict accurately what memory will be required in the near future. Modern hardware typically includes very aggressive *hardware* prefetching engines. However, while well suited to regular data access patterns, these are not amenable to the irregular patterns of *pointer chasing*, such as those that dominate a garbage collection trace.

Prefetching For GC Tracing Previous work [5, 6] has focused on the potential for prefetching objects from the marking stack with a tracing loop similar to the one in Figure 2. Ignoring memory accesses directly associated with the queuing mechanism (which we show in Section 6 to be insignificant), the two sources of memory access are a)

(a) Pentium-M

	Header								Side							
	Sync				Unsync				Sync				Unsync			
	Time	RI	L1	L2	Time	RI	L1	L2	Time	RI	L1	L2	Time	RI	L1	L2
Traverse	0.02	0.06	0.00	0.00	0.03	0.06	0.00	0.00	0.02	0.06	0.00	0.00	0.03	0.06	0.00	0.00
Enq-deq	0.11	0.30	0.06	0.01	0.13	0.31	0.06	0.01	0.10	0.29	0.06	0.01	0.13	0.31	0.06	0.01
Touch	0.15	0.10	0.45	0.59	0.18	0.10	0.45	0.58	0.15	0.10	0.45	0.61	0.17	0.10	0.45	0.60
Scan	0.39	0.54	0.53	0.43	0.47	0.55	0.53	0.43	0.39	0.53	0.53	0.44	0.47	0.56	0.53	0.44
Trace	0.59	0.63	0.98	1.04	0.71	0.65	0.97	1.02	0.59	0.61	0.98	1.06	0.70	0.65	0.98	1.04
Mark	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**

(b) AMD Athlon64

	Header								Side							
	Sync				Unsync				Sync				Unsync			
	Time	RI	L1	L2	Time	RI	L1	L2	Time	RI	L1	L2	Time	RI	L1	L2
Traverse	0.02	0.06	0.00	0.00	0.02	0.06	0.00	0.00	0.02	0.06	0.00	0.00	0.02	0.06	0.00	0.00
Enq-deq	0.11	0.30	0.06	0.01	0.12	0.31	0.06	0.01	0.11	0.29	0.06	0.01	0.12	0.31	0.06	0.01
Touch	0.14	0.10	0.51	0.54	0.15	0.10	0.51	0.54	0.14	0.10	0.48	0.55	0.16	0.10	0.50	0.55
Scan	0.46	0.54	0.58	0.56	0.51	0.55	0.58	0.56	0.46	0.53	0.60	0.56	0.52	0.56	0.58	0.56
Trace	0.63	0.63	0.99	1.01	0.71	0.65	0.99	1.01	0.63	0.61	0.99	1.02	0.72	0.65	0.99	1.02
Mark	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**

Table 5. Costs for Four Designs, Showing Time, Retired Instructions, L1 & L2 Misses, Normalized to Each Mark Scenario.

marking each object (line 11), and b) *scanning* each marked object (lines 7, 8 and 9). The first requires the object's mark metadata, which may be in the object's header, or held on the side. The second requires accessing the object's GCmap (normally in the object's header or accessed via indirection from the object's header), and scanning each of the pointers within the object.

7.1 Edge Enqueuing

The standard tracing loop depicted in Figure 2 has been optimized to reduce the number of objects that need to be enqueued and dequeued from the marking stack. Other alternative tracing loops have been discussed [9] but discarded due to the additional stack operations they entail. However, as the analysis in the previous section has shown, *queuing operations are not the bottleneck to improving performance*. We observe that it is possible to remove one of the memory access points in the loop by enqueuing *all* non-null objects during the *trace* rather than inspecting these objects and filtering out marked objects. Figure 6 gives pseudo-code for this approach, which we call *edge* enqueuing, as the referents

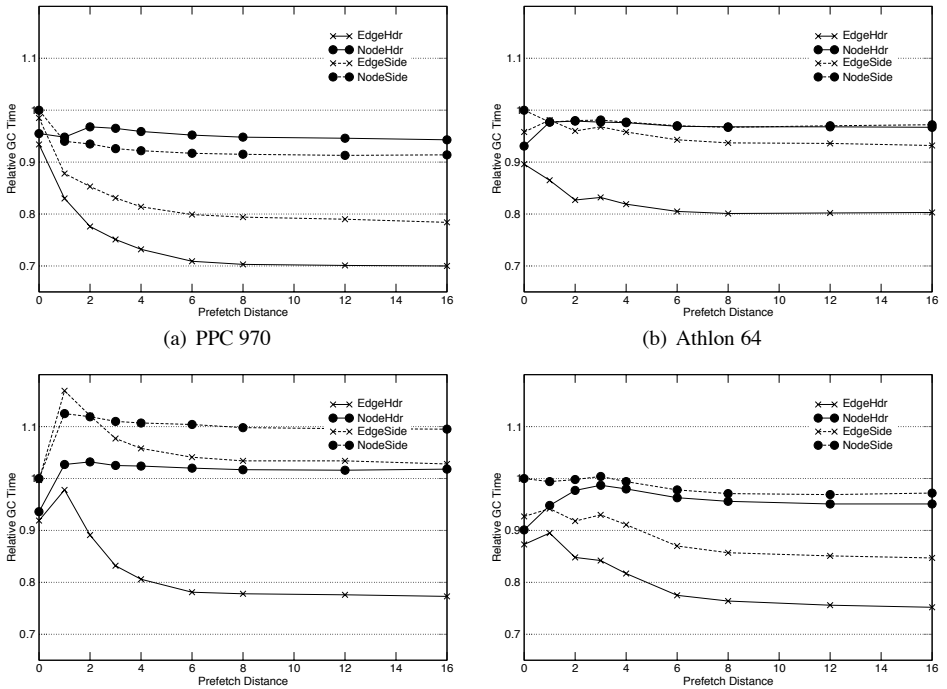
```
1  for p in root-set
2    mark(p)
3    queue.add(p)
4
5  while !queue.isEmpty()
6    obj = queue.remove()
7    if obj.testAndMark()   // hoist from 12
8      gcmap = obj.getGcMap()
9      for p in gcmap.pointers()
10       child = p.load()
11       if child != null
12         queue.add(child) // weaker guard
```

Figure 6. The *Edge-Enqueuing* Tracing Loop

of all non-null edges in the graph are placed on the stack. We refer to the traditional technique as *node* enqueuing (since each node in the graph is enqueued only once). Comparing Figures 2 and 6, the only difference between the two loops is that the mark operation (line 11 of Figure 2) is hoisted to line 7 in Figure 6.

By hoisting the mark operation, edge enqueuing weakens the guard on line 12 of Figures 2 and 6, so that children are eagerly enqueued, and the conditional mark operation is only performed later, immediately before the child is scanned. This increases the number of queue operations from the number of *nodes* in the live object graph to the number of *edges* in the live object graph, but does not affect the number of objects which are marked or scanned. As we have already shown, queuing operations form a negligible part of the cost of tracing. The benefit of edge enqueuing is that the *mark* (line 7 of Figure 6), *scan* (lines 8–10) and *trace* (lines 11 and 12) for a given object now occur contemporaneously, providing a far more predictable access pattern, which is more amenable to prefetching. The additional stack requirements of edge enqueuing are also reasonable. We found that the SPEC and DaCapo benchmarks used in this paper have on average 40% more edges than nodes, leading to about 40% more queue operations.

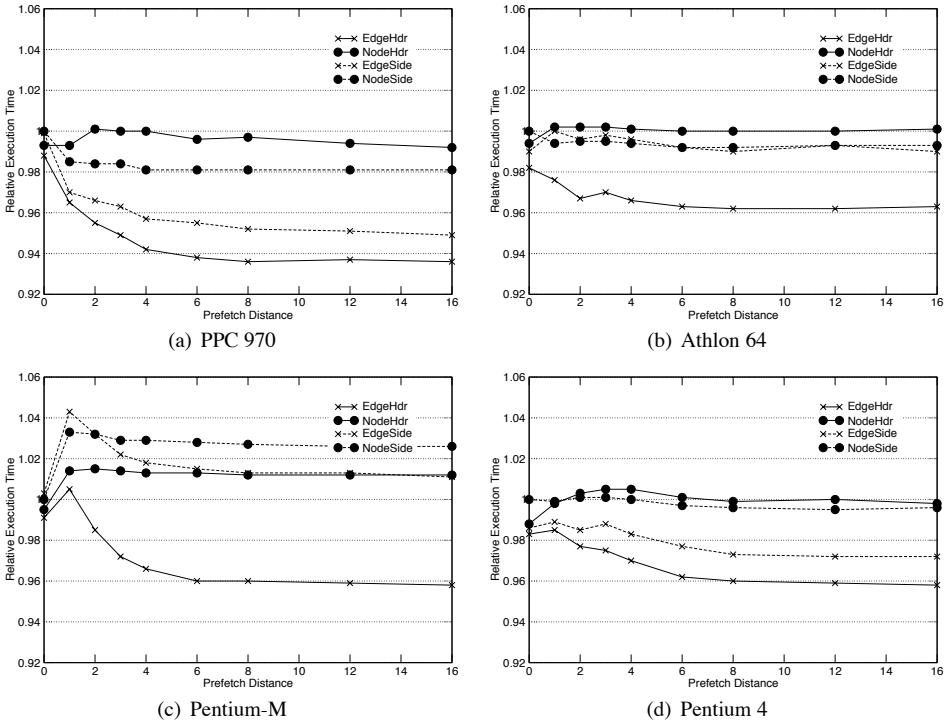
The contemporaneous *mark*, *scan* and *trace* mean that edge enqueuing has better temporal locality than node enqueuing. This addresses the first of our identified techniques for improving locality, and also provides a better environment for prefetching, since all accesses to each object occur together when that object is removed from the queue (in line 7). We implemented the FIFO-buffered mark queue (Figure 1, Cher et al. [6]) in our infrastructure to explore the effect of node and edge enquing on prefetcing. We control the prefetch distance by changing the size of the FIFO buffer.



(a) PPC 970 (b) Athlon 64

(c) Pentium-M (d) Pentium 4

Figure 4. Normalized GC Time vs. Prefetch Distance, as Geometric Mean of DaCapo, SPECjvm98 and Psuedojbb. Four Combinations of Edge and Node Enqueuing and Side and Header Metadata Are Shown.



(a) PPC 970 (b) Athlon 64

(c) Pentium-M (d) Pentium 4

Figure 5. Normalized Total Time vs. Prefetch Distance, as Geometric Mean of DaCapo, SPECjvm98 and Psuedojbb. Four Combinations of Edge and Node Enqueuing and Side and Header Metadata Are Shown. Measured In a Generous Heap (3 $\times$ Minimum).

7.2 Prefetching Results

We now evaluate the effectiveness of software prefetching in the tracing loop under both *edge* and *node* enqueuing models. Given their effect on locality, we compare with both *header* and *side bitmap* implementations of mark state, yielding four configurations.

Figures 4 and 5 show garbage collection and total time respectively, as we vary the prefetch distance from 0 to 16 for each of the four configurations (EdgeHdr, NodeHdr, EdgeSide, NodeSide), measured on four modern architectures. Each graph shows the geometric mean of performance for the full set of 17 benchmarks drawn from DaCapo and SPEC. Results are normalized to the time for NodeSide with no prefetch. NodeSide is the configuration which most closely follows prior work [5, 6]. We gathered results for different heap sizes, but found that the effect of prefetch was independent of heap size. We report here the results for a fairly generous heap; three times the minimum heap size for each benchmark. We performed identical experiments on $2\times$ and $4\times$ heaps, and found the GC-time results were almost indistinguishable, although the overall impact of this GC-time optimization on total time obviously increases as time spent in GC goes up in smaller heaps. For completeness, Figure 7 shows total time as a function of heap size. In order to maintain a consistent number of collections across the different configurations and fairly assess the effect of prefetch, we did not allow the header meta-data configurations to make use of the minor space saving due to avoiding the side bitmap.

The results in Figure 4 show a clear win for EdgeHdr (edge order enqueuing with mark state in the header), which out performs all other configurations on all four architectures. The impact of prefetching ranges from modest to poor for node order enqueuing. This poor result for node enqueuing is consistent with prior work which saw only modest improvements with prefetching [6, 5]. The difference in effectiveness of the prefetch operations across the architectures is significant, with GC-time performance improvements ranging from 30% on the PowerPC 970, to around 20% on the Athlon 64, and with notable degradations seen for a number of configurations on the Pentium-M. We also see that prefetch distances greater than eight provide little advantage, with longer prefetch distances degrading performance slightly on the Athlon 64.

In order to perform a direct comparison with Cher et al. [6], we followed their approach and measured the speedup in total time in a very space-constrained heap, $1.125\times$ the minimum heap (left-most data in graphs in Figure 7). In this situation, we obtain total running time speedups of 18%, 15%, 11% and 16% on the PowerPC 970, Pentium-M, AMD and Pentium 4 respectively. This compares very favorably with Cher et al's result of 6% speedup on the PowerPC 970 (achieved with arguably more amenable benchmarks), and is consistent with our results which show that edge enqueuing is essential to effective software prefetch.

7.3 Robustness: Experiences With Other Code Bases

Since submitting this work for publication, we have ported the work to a substantially different version of the Jikes RVM code base. Our efforts to reproduce our original results were frustrating, but ultimately illuminating. Our experience may be relevant to anyone wishing to implement prefetching in their own garbage collector.

First, poor code quality in the compiled code for the tracing loop can dominate any prefetching advantage. Through painstaking detective work we established that the absence of a number of aggressive optimizations, the addition of an extraneous virtual dispatch, and the addition of an extra register spill each reduced the tracing loop performance sufficiently that the prefetching advantage was negligible or zero. These were all (unintended) artifacts of changes in the underlying Jikes RVM code base.

Second, the prefetch optimization only improves the tracing loop. In a garbage collector where the tracing loop does not dominate performance, the usefulness of this optimization will be correspondingly diminished. The way Jikes RVM collects its 'boot image' was changed from tracing (which utilizes the prefetch) to an explicit enumeration of pointer fields (which does not). We found that in small benchmarks where the Jikes RVM boot image formed a large fraction of the workload, the effectiveness of the prefetch optimization was significantly diminished.

Finally, we found that on some architectures the FIFO structure placed in front of the mark stack [6] gave a performance advantage even *without* the prefetch (this is evident when performance improves at $x = 1$ in Figure 4, and we saw it on an Intel Core 2 Duo). Our best explanation for this surprising behavior is that the FIFO structure enabled more aggressive hardware speculation through more predictable access patterns, just as it facilitates software prefetch.

8. Conclusion

The most performance-critical element of most modern garbage collection algorithms is the tracing loop which performs a transitive closure over the live objects. The performance of this loop well known to limited by the poor memory locality associated with the irregular walk of the heap performed by the trace. In this paper we address the performance of this tracing loop, first analytically, then with a simple optimization which yields considerable performance advantages over the state of the art.

This paper makes three main contributions: 1) we develop a methodology and framework for accurately and deterministically analyzing the performance-critical garbage collection tracing loop, 2) we offer a number of insights and improvements over conventional design choices for mark-sweep collectors, and 3) we find that by combining two simple ideas which each offer modest gains—edge enqueuing and mark stack prefetching—we can greatly improve the performance of the tracing loop on four architectures on a

large benchmark suite. While previous work was only able to show a modest advantage from software prefetching, we are able to show average improvements over 17 benchmarks of 20-30% of collection time and 4-6% of total time when running in moderate heaps.

Acknowledgments

We thank Hans Boehm, Tony Hosking, Richard Jones, and Eliot Moss for their assistance, feedback and encouragement.

References

[1] B. Alpern et al. The Jalapeño virtual machine. *IBM Systems Journal*, 39(1):211–238, February 2000.

[2] M. Arnold, S. J. Fink, D. Grove, M. Hind, and P. Sweeney. Adaptive optimization in the Jalapeño JVM. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 47–65, Minneapolis, MN, October 2000.

[3] S. M. Blackburn, P. Cheng, and K. S. McKinley. Myths and realities: The performance impact of garbage collection. In *ACM SIGMETRICS Conference on Measurement & Modeling Computer Systems*, pages 25–36, NY, NY, June 2004.

[4] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. In *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-Oriented Programing, Systems, Languages, and Applications*, New York, NY, USA, Oct. 2006. ACM Press.

[5] H.-J. Boehm. Reducing garbage collector cache misses. In *The 2000 International Symposium on Memory Management*, pages 59–64, 2000.

[6] C.-Y. Cher, A. L. Hosking, and T. N. Vijaykumar. Software prefetching for mark-sweep garbage collection: hardware analysis and software redesign. In *ASPLOS-XI: Proceedings of the 11th international conference on Architectural support for programming languages and operating systems*, pages 199–210, New York, NY, USA, 2004. ACM Press.

[7] M. W. Hicks, J. T. Moore, and S. M. Nettles. The measured cost of copying garbage collection mechanisms. In *ICFP '97: Proceedings of the second ACM SIGPLAN international conference on Functional programming*, pages 292–305, New York, NY, USA, 1997. ACM Press.

[8] X. Huang, S. M. Blackburn, K. S. McKinley, J. E. B. Moss, Z. Wang, and P. Cheng. The garbage collection advantage: Improving mutator locality. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, Vancouver, BC, 2004.

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

[10] J. McCarthy. Recursive functions of symbolic expressions and their computation by machine. *Communications of the ACM*, 3(4):173–197, Apr. 1960.

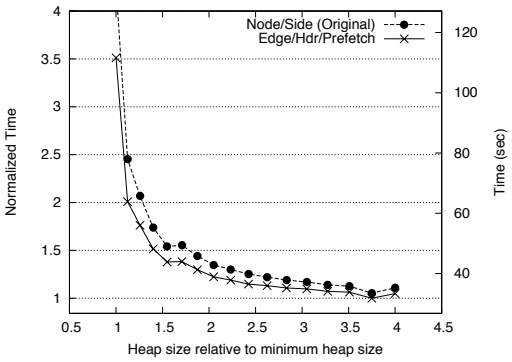
[11] M. Pettersson. Linux Intel/x86 performance counters, 2003. http://user.it.uu.se/ mikpe/linux/perfctr/.

[12] Standard Performance Evaluation Corporation. *SPECjvm98 Documentation*, release 1.03 edition, March 1999.

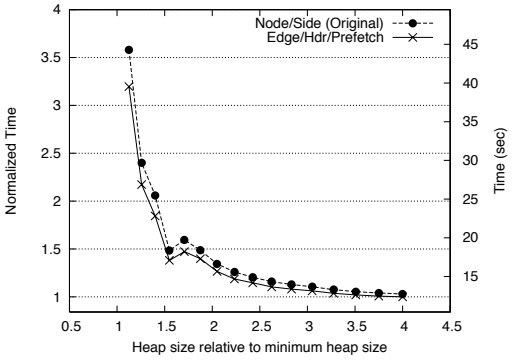
[13] Standard Performance Evaluation Corporation. *SPECjbb2000 (Java Business Benchmark) Documentation*, release 1.01 edition, 2001.

[14] P. R. Wilson, M. S. Johnstone, M. Neely, and D. Boles. Dynamic storage allocation: A survey and critical review. In *ACM International Workshop on Memory Management*, 1995.

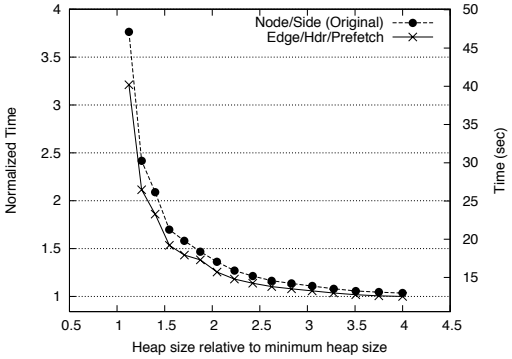
[15] P. R. Wilson, M. S. Lam, and T. G. Moher. Effective static-graph reorganization to improve locality in garbage-collected systems. In *ACM SIGPLAN Conference on Programming Languages Design and Implementation*, pages 177–191, Toronto, Canada, June 1991.



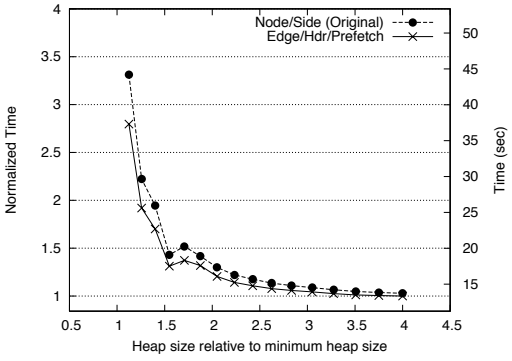
(a) PPC 970



(b) Athlon 64



(c) Pentium-M



(d) Pentium 4

Figure 7. Relative Total Execution Time As a Function of Heap Size, Comparing Node Order and Side Mark Against Prefetching With Edge Order and Header Mark. Geometric Mean of DaCapo, SPECjvm98 and Psuedojbb.