Software Prefetching for Mark-Sweep Garbage Collection: Hardware Analysis and Software Redesign

Chen-Yong Cher
School of Electrical and Computer Engineering
Purdue University
West Lafayette, IN 47907
chenyong@ecn.purdue.edu

Antony L. Hosking
Department of Computer Sciences
Purdue University
West Lafayette, IN 47907
hosking@cs.purdue.edu

T. N. Vijaykumar
School of Electrical and Computer Engineering
Purdue University
West Lafayette, IN 47907
vijay@ecn.purdue.edu

ABSTRACT

Tracing garbage collectors traverse references from live program variables, transitively tracing out the closure of live objects. Memory accesses incurred during tracing are essentially random: a given object may contain references to any other object. Since application heaps are typically much larger than hardware caches, tracing results in many cache misses. Technology trends will make cache misses more important, so tracing is a prime target for prefetching.

Simulation of Java benchmarks running with the Boehm-Demers-Weiser mark-sweep garbage collector for a projected hardware platform reveal high tracing overhead (up to 65% of elapsed time), and that cache misses are a problem. Applying Boehm's default prefetching strategy yields improvements in execution time (16% on average with incremental/generational collection for GC-intensive benchmarks), but analysis shows that his strategy suffers from significant timing problems: prefetches that occur too early or too late relative to their matching loads. This analysis drives development of a new prefetching strategy that yields up to *three times* the performance improvement of Boehm's strategy for GC-intensive benchmarks (27% average speedup), and achieves performance close to that of perfect timing (*ie*, few misses for tracing accesses) on some benchmarks. Validating these simulation results with live runs on current hardware produces average speedup of 6% for the new strategy on GC-intensive benchmarks with a GC configuration that tightly controls heap growth. In contrast, Boehm's default prefetching strategy is ineffective on this platform.

Categories and Subject Descriptors

B.3.2 [**Memory Structures**]: Design Styles—*cache memories*;
B.3.3 [**Memory Structures**]: Performance Analysis and Design Aids—*simulation*;
D.3.4 [**Programming Languages**]: Processors—*memory management (garbage collection), run-time environments*

General Terms

Algorithms, management, measurement, performance, design, experimentation, languages

Keywords

Cache architecture, prefetching, garbage collection, mark-sweep, prefetch-on-grey, buffered prefetch, depth-first, breadth-first

1. INTRODUCTION

Automatic dynamic memory management, commonly referred to as *garbage collection* (GC) [13], is an important aspect of run-time systems for modern applications. Languages such as Java^{TM} [10] impose automatic dynamic memory management as a mandatory feature of the language (*ie*, there is no *free* operation in Java). Languages like C, while not forcing automatic memory management on programmers, are still amenable to modern GC techniques, making GC a realistic option for them as well.

The comparative benefits of automatic versus manual memory management are often hotly-debated among programmers, yet GC is now a well-accepted approach for developing reliable large-scale applications that allocate and manipulate complex linked data structures. Its key advantages derive from the fact that GC relieves programmers from the need to free allocated objects explicitly, resulting in separation of concerns between data producers and consumers, promoting modularity and composition of rich application libraries, and eliminating "dangling pointer" errors that result when objects are freed prematurely.

The basic operating principle of GC is that an object allocated in the dynamic *heap* will not be freed while there remain *live* references to that object. Any object referred to by live references is itself live, and cannot be freed. At a given point in a program's execution, a reference is said to be live if it *may* be used by the program at some point in the future to access its target object. GC techniques typically use a conservative approximation of reference *liveness*, simply assuming that all references held in the registers, static areas, and thread activation stacks are live. Live heap objects are those reachable (transitively) via live references; references held in live objects are themselves live (transitively). Thus, *tracing* GC algorithms operate by traversing references from known *roots* (*eg*, registers, globals, and threads) to heap-allocated objects, and then object-to-object for references held in those objects. All objects (transitively) reachable from the *roots* are considered to be live. Objects not transitively reachable are considered to be dead, and can be freed.

Applications typically allocate objects from the heap up to some

GC threshold, at which point tracing is used to enumerate the live objects; the remaining objects are freed. The space consumed by freed objects is recycled for use in satisfying future allocation requests.

While GC incurs overhead in terms of extra instructions as well as cache misses, technology trends are increasing the gap between processor speed and memory latency and expanding on-chip execution resources (*ie*, functional units). These trends will make extra cache misses more significant. Despite the overheads of GC, it is becoming increasingly prevalent, not just for Java and other modern languages, but even for C-based applications (*eg*, even the popular gcc compilers now use GC internally for their memory management).

1.1 Our contribution

In this paper, we focus on reducing the cache miss overhead of GC, specifically targeting misses incurred by tracing collectors as they enumerate the live objects. We analyze the performance of the portable and widely-used Boehm-Demers-Weiser garbage collector [6, 5], including Boehm's own prefetching strategy for reducing GC cache misses while tracing [4]. Our results, obtained via simulation of standard Java benchmarks, reveal the overhead of tracing (up to 65% of elapsed time), and demonstrate the improvements obtained using Boehm's prefetching approach (16% on average for incremental/generational GC on GC-intensive benchmarks).

The keys to effective prefetching are knowing *what* to prefetch and *when* to prefetch it. For GC, *what* to prefetch is known: we must trace the live objects of the graph, and their references are discovered during the trace. *When* to prefetch is trickier: prefetching an object too early means it may be displaced from the cache before we can trace it; prefetching too late exposes memory latency. Thus, the key issue is timeliness. Prefetches can occur only as object references are discovered, while an object is scanned, whereas tracing algorithms impose traversal orders that are often different from that order.

Our experimental methodology uses simulation of Java benchmarks running on projected hardware to obtain detailed analysis of GC prefetching behavior. Our key observation is that Boehm's prefetching approach suffers from severe timing problems: many prefetches occur too early or too late. The reason is that prefetches occur essentially breadth-first (FIFO), while BDW uses depth-first traversal (LIFO) to access objects. This difference in ordering implies that the time between prefetch and access can be arbitrarily short/long leading to prefetches that are too late/early. Based on our analysis, we devise and implement a new prefetching strategy, which imposes limited FIFO processing over a small window of the BDW mark stack, so that the amount of tracing work performed, and hence time elapsed, between prefetch and access is *bounded*. The size of this window can be tuned for memory latency, permitting tight control over prefetch timeliness. Our new approach yields up to *three times* the simulated performance improvement of Boehm's strategy for GC-intensive benchmarks (27% on average), and achieves performance close to that of perfect timing (*ie*, no misses for tracing accesses) for some benchmarks. Validating our simulation results on current hardware, the new strategy also yields performance improvement (from 2% to 6% on average, depending on GC configuration), even though current hardware has shorter memory latency and tighter restrictions on the number of in-flight cache misses than the simulated platform.

1.2 Outline

The remainder of the paper is organized as follows. In Section 2, we describe the basics of tracing GC and its fundamental costs, as well as the Boehm-Demers-Weiser collector used as the basis of our study. Section 4 describes our experimental framework and methodology. Results and observations follow in Section 5. We conclude with a discussion of related work in Section 6, and a summary of our findings in Section 7.

2. TRACING GARBAGE COLLECTION

Tracing garbage collection algorithms can be categorized into two basic approaches: *mark-sweep* collectors, and *copying* collectors. Mark-sweep collectors operate in two phases, first marking the live objects as they are traced, and then sweeping up the unmarked dead objects, gathering them onto free lists for use in subsequent allocations. Copying collectors *evacuate* live objects as they are discovered, copying them into a new heap space; dead objects remain behind in the old space, which can then be freed wholesale.

2.1 GC overheads

The overhead of tracing GC breaks down into two components: (1) enumerating the live objects (*tracing*), and (2) deallocating the dead objects (*freeing*). These costs are inherent to tracing GC regardless of whether liveness is captured explicitly, by marking, or implicitly, by copying. Freeing the dead objects in a copying collector incurs no real overhead, since freeing the old space is a constant-time operation. This fact is sometimes cited as an argument against mark-sweep collectors since the sweep phase must examine the whole heap, whereas the cost of a copying collector is proportional to the size of the live data (ignoring the overhead of copying the objects). However, *lazy* sweeping [12, 4] reduces the asymptotic complexity of mark-sweep collectors to that of copying collectors. Lazy sweeping effectively transfers the cost of the sweep phase to allocation, piggy-backing some small amount of sweep work onto each allocation request, sufficient to discover enough free space to satisfy the request. Moreover, because some of the swept space is itself used to satisfy the allocation request, lazy sweeping exhibits better locality. Thus, the cost of mark-(lazy)sweep GC reduces to the cost of enumerating the live objects, as for copying collection.

Enumerating the live objects is thus the principal cost of garbage collection; tracing must complete fully before any object can be freed. Moreover, it is memory-intensive since the references contained in each live object must be enumerated and traversed. Even worse, the memory accesses incurred for tracing are essentially random: a given object may contain references to *any* other object in the heap. Since application heaps are typically much larger than the capacity of the hardware cache hierarchy, tracing results in large numbers of cache misses. Tracing also displaces the application's working set from the caches and pollutes them with data not germane to the working set; our results show pollution is not a big problem, even for incremental GC.

2.2 Generational GC

To reduce the intrusiveness of GC, notably extended pauses due to GC, researchers have devised algorithms that exploit demographic invariants in typical programs. One invariant is that most objects die young. As a result, focusing tracing effort on the youngest objects in the heap is likely to yield a larger fraction of dead objects than from the heap as a whole. Thus, *generational* GC algorithms [24] segregate objects by age (temporally, though not necessarily spatially), and focus tracing effort on the youngest objects, tracing them more frequently than the older objects. All that is needed for generational collection is a mechanism for tracking references from old objects to young objects, and treating those references as GC roots (in addition to the usual roots) when tracing the young objects independently of the old objects. Tracing only the young generation reduces GC pauses, while usually yielding sufficient free space for the application to proceed. If tracing the young objects does not yield sufficient space, then GC can revert to a whole-heap trace.

2.3 Incremental GC

Also to reduce intrusiveness, *incremental* GC algorithms interleave tracing work with application execution. Using the standard graph analogy, the object heap is a directed graph, with objects as nodes and references as edges. The collector traces the reachability of nodes in the graph from the roots. *Stop-the-world* collectors assume that the object graph remains static while the collector runs. In contrast, incremental collectors view the application program as a *mutator* of the object graph, that can arbitrarily transform the graph before the collector has completed tracing it. As a result, the collector must compensate for mutations that change the reachability of objects. Generally, it is safe to ignore objects losing a reference while the collector is tracing the graph – objects that lose their last references will be collected at the next GC. More problematic is the creation of new references *from* already-traced objects to an object whose other references are overwritten by the mutator. If the collector never sees those old references then the object will never be traced. Thus, incremental collectors require synchronization with the mutator.

Dijkstra's *tri-color* abstraction [9] serves as a useful basis for understanding synchronization approaches for incremental GC. Objects in the heap are painted one of three colors:

Black nodes have already been traced (including all the references they contain);

Grey nodes contain references still to be traced;

White nodes are untraced; at the end of tracing they are garbage.

Tracing is the process of partitioning the objects into black and white nodes. Tracing is complete when there are no grey nodes; nodes left white are garbage. Tracing correctness means preventing the mutator from writing references to white objects into black objects without the collector knowing about it. There are two possible approaches:

1. Prevent the mutator from seeing white objects; or
2. Inform the collector about creation of any reference from black to white objects so that it can be traced.

The first approach imposes a *read barrier* on pointer traversals, while the second requires a *write barrier* on pointer stores. So long as barriers are used to ensure correctness, incremental tracing can proceed independently of the mutator. One approach piggybacks a small amount of tracing onto each allocation request, to ensure tracing progress is tied to mutator progress (as measured by allocation). Alternatively, tracing can run concurrently with the mutator in a separate thread or in parallel on a separate processor.

Incremental tracing minimizes intrusiveness by reducing collector pause times, at the expense of synchronization overheads and the churning effect of tracing on the memory hierarchy. As a result, while responsiveness for interactive or other (soft) real-time activities may improve, overall throughput may suffer.

3. PREFETCHING IN GC

Boehm [4] was the first to apply prefetching techniques within GC, implemented within the Boehm-Demers-Weiser (BDW) collector. We begin with an overview of BDW before discussing how it incorporates prefetching.

3.1 The Boehm-Demers-Weiser collector

The Boehm-Demers-Weiser (BDW) mark-(lazy)sweep collector is popular due its portability and language-independence. It epitomizes a class of collectors known as *ambiguous roots* collectors. Such collectors are able to forego precise information about roots and knowledge of the layout of objects, by assuming that any word-sized value is a potential heap reference. Any value that ambiguously *appears* to refer to the heap (while perhaps simply having a value that looks like a heap reference) is treated as a reference – the object to which it refers is considered to be live.[1] The upshot of ambiguity is that ambiguously-referenced objects cannot move, since their ambiguous roots cannot be overwritten with the new address of the object; if the ambiguous value is not really a reference then it should not be modified. The BDW collector treats registers, static areas, and thread activation stacks ambiguously. If object layout information is available (from the application programmer or compiler) then the BDW collector can make use of it, but otherwise values contained in objects are also treated ambiguously.

The advantage of ambiguous roots collectors is their independence of the application programming language and compiler. The BDW collector supports garbage collection for applications coded in C and C++, which preclude accurate garbage collection because they are not type-safe. BDW is also often used with type-safe languages whose compilers do not provide the precise information necessary to support accurate GC. The minimal requirement is that source programs not hide references from GC, and that compilers not perform transformations that hide references from GC. Thus, BDW is used in more diverse settings than perhaps any other collector. As a result, the BDW collector has been heavily tuned, both for basic performance, and to minimize the negative impact of ambiguous roots [3].

The BDW collector supports generational/incremental collection using a simple write barrier based on knowing which virtual memory pages have been modified recently. For generational GC, only pages modified since the last collection can contain references to objects allocated since that last collection. Treating these newly allocated objects as the young generation for generational GC means tracing from the roots and *modified* older pages in the heap. BDW interleaves incremental marking with the mutator, piggy-backing a little marking work onto every allocation. The amount of marking done is adjusted to avoid the mutator getting too far ahead of marking, on the basis of bytes marked versus bytes allocated. When the incremental marking phase finishes, the only thing preventing GC completion is the fact that the mutator may have created references from black to white objects (as discussed above). BDW simply treats all pages modified by the mutator since the start of marking as if they are grey, and completes tracing from those pages before permitting the mutator to run. This last "stop-the-world" marking phase is usually relatively brief compared to the overall cost of tracing.

The BDW write barrier is implemented in one of two ways, depending on the availability of support from the operating system. One approach is to use virtual memory page protection primitives to record page modifications [1]. This approach can be expensive given the overhead of fielding protection traps from the operating system via user-level signal handlers to record updates. A more efficient, though less portable, alternative is to obtain dirty-page information directly from the operating system's virtual memory

[1]Ambiguous roots collectors are also sometimes referred to as *conservative* collectors, since they treat potential references conservatively. However, the undecidability of variable liveness means all collectors are effectively conservative. We use the term ambiguous roots to dispel such confusion.

```
Ensure that all objects are white.
Grey all objects pointed to by a root.
While there is a grey object g
    Blacken g.
    For each pointer p in g
        If p points to a white object
            Grey that object.
```

Figure 1: The BDW tracing algorithm

```
Push all roots on the mark stack, making them grey.
While there is a pointer to an object g on the stack
    Pop g from the mark stack.
    If g is still grey (not already black)
        Blacken g.
        For each pointer p in g's target
            Prefetch p.
            If p is white
                Grey p and push p on the mark stack.
```

Figure 2: Prefetch-on-grey (PG)

page tables via system calls or access to per-process information maintained in the `/proc` file-system. BDW supports both approaches depending on the host system.

3.2 Prefetching in BDW

The BDW collector supports a rudimentary form of prefetching during tracing. The basic approach is to prefetch the target of a reference at the point that the target object is discovered to be grey. An abstract formulation of the BDW tracing algorithm is given in Figure 1 [4].

In reality, BDW maintains a table *separate* from the objects, containing one mark bit per object. A mark bit value of zero corresponds to a white object; a value of one indicates a grey or black object. A separate stack contains the addresses of all grey objects. When an object is greyed its mark bit is set, and it is pushed onto the mark stack. Blackening an object corresponds to removing it from the mark stack.

Having observed via profiling that a significant fraction of the time for this algorithm is spent retrieving the first pointer p from each grey object, Boehm [4] introduced a prefetch operation at the point where p is greyed and pushed on the mark stack. The resulting tracing algorithm, with concrete mark stack and Boehm's prefetch-on-grey (PG), appears in Figure 2.

Boehm also permutes the code in the marker slightly, so that the last pointer contained in g to be pushed on the mark stack is prefetched first, so as to increase the distance between the prefetch and the load through that pointer. Thus, for every pointer p inside an object g, the prefetch operation on p is separated from any dereference of p by most of the pointer validity (necessary because of ambiguity) and mark bit checking code for the pointers contained in g. Boehm notes that this allows the prefetch to be helpful even for the first pointer followed, unlike the cases studied by Luk and Mowry using "greedy prefetching" on pointer-based data structures [16].

In addition, Boehm linearly prefetches a few cache lines ahead when scanning an object. These *scan-prefetches* help with large objects, as well as prefetching other relevant objects in the data structure being traced.

Boehm reported prefetch speedups between 1% and 17% for a small set of C-based synthetic benchmarks and real applications.

3.3 Observations on prefetch-on-grey

Before presenting our experimental evidence for timing problems with Boehm's prefetch-on-grey, we first make some observations on how such problems may arise. As noted earlier, the success of software prefetching in hiding cache miss latencies depends on timeliness. If a prefetch is issued too early, then the data it prefetches may be displaced from the cache by other accesses that force its eviction, before the data is used. If a prefetch is issued too late, then the data will not be available in-cache when needed, exposing the miss latency.

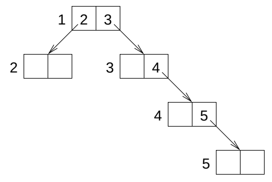


Figure 3: Tree example

Prefetch-on-grey is intimately tied to the depth-first access order imposed by the stack-based BDW marking algorithm: prefetch occurs when a node is pushed on the stack, while access occurs when it is popped. The delay between pushing and popping a reference is highly variable, and depends on the data structure being traced. Consider a tree data structure as illustrated in Figure 3. Tracing from root object 1, BDW with prefetch-on-grey issues prefetches in the order 3-2-4-5, but the stack-based BDW tracing algorithm visits these objects in depth-first order 3-4-5-2. The prefetch of 2 occurs early, but the access to 2 occurs late. If the work performed in tracing 3-4-5 delays the access to 2 for too long (*ie*, the prefetch is issued too early) then the prefetched object 2 may be displaced from the cache.

3.4 Our improvement: buffered prefetching

An alternative approach might be to use a (FIFO) mark queue instead of a (LIFO) mark stack, for breadth-first traversal. Prefetch order would then be 2-3-4-5, matching the traversal order 2-3-4-5. However, breadth-first search suffers from poorer locality than depth-first search, tending to group traversal of unrelated objects in a data structure (*eg*, cousins, rather than parents and children).[2] Also, prefetch-on-grey with breadth-first search still suffers from lack of control over timeliness: objects containing many references (such as large arrays) result in prefetching all the targets before the first target can be scanned.

Instead, we propose a new software prefetching approach using buffering (in software) to defer prefetching until grey references are popped from the stack. In effect, we use a hybrid of queue-based (FIFO) and stack-based (LIFO) processing, by interposing a fixed-size window over references at the top of the mark stack, prefetching and processing references in the window in FIFO order. The size of the window is tune-able with varying memory latency (*ie*, larger window for longer latency). When a grey reference is popped from the stack, we prefetch its target, and drop the reference into the software prefetch buffer. We pop and prefetch as many references from the top of the stack as will fill the buffer. When the buffer fills, or when the mark stack is empty, we scan the object at the head of the buffer queue. Thus, at any given time, the maximum number of prefetch instructions in flight is limited to the size of the prefetch buffer. Moreover, we scan objects from the prefetch buffer in FIFO order. In combination, we thus retain a tight bound on the time between issue of a prefetch instruction and its associated data access. Our results show that this algorithm eliminates nearly all

[2]For this same reason, copying algorithms often cluster related objects together in depth-first order, to improve object locality [20, 22, 25].

```
Push all roots on the mark stack, making them grey.
Loop
  While there is a pointer to an object $g$ on the stack
    If the prefetch buffer is full
      Dequeue a pointer $b$ from the buffer.
      If $b$ is still grey (not already black)
        Blacken $b$.
        For each pointer $p$ in $b$'s target
          If $p$ is white
            Grey $p$ and push $p$ on the mark stack.
    Pop $g$ from the mark stack.
    Prefetch $g$ and enqueue it in the prefetch buffer.
  If the prefetch buffer is empty
    Exit loop.
  Dequeue a pointer $b$ from the buffer.
  If $b$ is still grey (not already black)
    Blacken $b$.
    For each pointer $p$ in $b$'s target
      If $p$ is white
        Grey $p$ and push $p$ on the mark stack.
  Continue loop.
```

Figure 4: Buffered-prefetch (BP)

cache misses in the marking code. Our revised algorithm appears in Figure 4.

Returning to the example of Figure 3, let us assume use of a 2-entry prefetch buffer. Beginning with object 1 on the stack, the buffered-prefetch algorithm executes as follows: pop-prefetch-enqueue 1, dequeue-scan 1, push 2, push-pop-prefetch-enqueue 3, pop-prefetch-enqueue 2, dequeue-scan 3, push-pop-prefetch-enqueue 4; dequeue-scan 2; dequeue-scan 4, push-pop-prefetch-enqueue 5, dequeue-scan 5. Considering simply the prefetches and scans from this sequence we have: prefetch 1, scan 1, prefetch 3, prefetch 2, scan 3, prefetch 4, scan 2, scan 4, prefetch 5, scan 5.

Despite reducing most cache misses in the marking code (as revealed in our experiments below), this algorithm does delay processing of *cache-resident* objects, since their references must now transition through the buffer queue as well as the stack before they can be scanned. This may hurt performance. To study the impact of this deferral, our experiments also consider use of *informing* prefetches [11] that notify if the prefetch hits immediately in the cache. If so, then the collector can scan the object immediately, and avoid the overhead of managing the buffer. However, our experiments show that delaying the processing of the blocks on hits does not hurt performance.

4. EXPERIMENTS

Our experiments use the gcc-based static compiler for Java (gcj) to compile a set of standard Java benchmarks into static Alpha executables. We then simulate these benchmarks using the SimpleScalar architecture simulator [2]. Our methodology is simulator-based because detailed analysis of prefetch timing behavior is not possible with live runs, even using hardware performance counters. We compare several different configurations of the BDW collector, with and without incremental/generational support and prefetch-on-grey(PG)/buffered-prefetch(BP). We use the default parameters for BDW as shipped with gcj, such as the free-space divisor (FSD) used to trigger GCs and the GC rate variable that dictates when incremental marking should occur. All prefetches load into L1 cache as defined by the Alpha ISA.

To ensure best GC performance on our simulated machine, we tuned the scan-prefetch distance for linear prefetches used during object scanning. Since both PG and BP use the same scanning code, both benefit from this tuning. Trial-and-error measurements led to selection of the scan-prefetch distance at 384 bytes (three L2 cache blocks ahead). A side-effect of this optimal scan-prefetch distance is that the second and third L2 cache blocks are not covered by scan-prefetches. To eliminate cache misses while scanning these two blocks, we can issue two additional prefetches, one each for the second and third blocks of an object, at the same time as the PG or BP prefetch, depending on the algorithm. In effect, these *gap-prefetches* increase the effective prefetch granularity for PG/BP prefetches to cover the first three cache blocks of an object; they fill the gap between PG/BP prefetches and scan-prefetches. Our experiments show that gap-prefetches improve both PG and BP, so our experiments use gap-prefetching unless otherwise mentioned.

Table 1: Simulated hardware parameters

Processor	8-way issue, 300 RUU, 60 LSQ, 8 integer ALUs, 2 integer mul/div units, 2 memory ports, 4 FP ALUs, 2 FP mul/div units
Branch prediction	8K/8K/8K hybrid predictor; 32-entry RAS, 8192-entry 4-way BTB, 8-cycle misprediction penalty
Caches	64KB 2-way 2-cycle I/D L1, 1MB 8-way 12-cycle L2, both LRU, 32 L1 MSHRs and 32 L2 MSHRs.
Memory	Infinite capacity, 300 cycle latency
Memory bus	32-byte wide, pipelined, split transaction

4.1 Experimental platform

We modified SimpleScalar version 3.0 [2] to simulate an 8-way issue, out-of-order processor. To simulate accurately the timing effects of cache misses, we incorporated support for simulation of Miss Status History Registers (MSHRs). SimpleScalar simulates Alpha binaries directly, including standard system calls, but we added emulation of system calls that mimic OS-maintained page-level dirty bits for BDW incremental/generational GC. The overhead to maintain and deliver these dirty bits is not directly simulated. As a result, our results give an ideal impression of incremental/generational executions, for which querying dirty bit information costs nothing. Table 1 lists the default hardware parameters we use for all experiments except for those we specify otherwise.

4.1.1 Capitulating loads

As noted earlier, BP delays processing of blocks that hit in the cache, which may slow down the GC mark phase. If tracing immediately processes a block that *hits* in the cache without entering the block into the prefetch queue, then performance may improve by not delaying dependent tracing and allowing additional overlap of tracing work with earlier prefetches (*ie*, buffering will degrade overall performance if all blocks hit in the cache). To see whether deferral of tracing in BP hurts performance, we also simulate a new kind of *informing* load instruction [11] that we call a *capitulating* load (c-load). A c-load is one that attempts the load operation, but which returns immediately if the access is a miss, indicating the miss status with an additional output register operand to the load operation. On a hit, a c-load returns the accessed data along with the hit status. Applications can use the status information as a condition, to choose whether to process the targeted block if the load hits, or to do something else if it misses. In our case, we can use c-loads to decide dynamically whether to use BP-deferred tracing of a non-resident object, or immediate tracing of a resident object. We implement c-loads in the simulator to return hit status on L1 cache hit and miss status otherwise, and modify the marking code to *conditionally* prefetch (and buffer) objects that miss.

Table 2: Benchmark inputs

SPECjvm98		Java Olden	
Benchmark	Parameters	Benchmark	Parameters
compress	1 iteration	bh	-b 4096 -m
jess	*standard*	bisort	-s 250000 -m
raytrace	*standard*	em3d	-n 2000 -d 100 -m
db	*standard*	mst	-v 256 -m
javac	*standard*	perimeter	-l 16 -m
mpegaudio	*standard*	power	-m
jack	*standard*	treeadd	-l 20 -m
		tsp	-c 10000 -m
		voronoi	-n 20000 -m

4.2 Benchmarks

For our experiments we use the standard SPECjvm98 benchmarks [21] as well as Java versions of the Olden benchmark suite [7, 8]. The Java Olden benchmarks are heap-intensive programs with large footprints that stress the memory system. We used gcc version 3.0, with optimization level `-O2`, targeting Alpha Tru64 version 5, to compile and statically link the benchmarks. Since Zilles [26] points out that the Olden benchmark health is inefficient in both algorithm and implementation, making it unrepresentative as a benchmark, we omit health from our results. Also, because our simulator lacks support for Java threads, we omit mtrt of SPECjvm98. Instead, we use the single-threaded benchmark raytrace, which mtrt invokes as its per-thread workload. Table 2 lists the benchmarks and their parameters for our experiments.

5. RESULTS

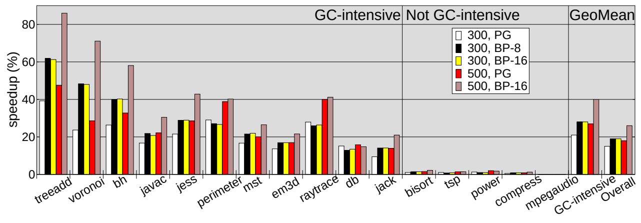
We now present our simulation results. Section 5.1 shows overall speedup of Boehm's PG and our new BP, while also varying memory latency. Section 5.3 explores the effect of varying the MSHR count for BP. Finally, we show the effect of using c-loads to choose dynamically between BP-deferred and immediate tracing.

For completeness, we show results for three different GC configurations: whole-heap stop-the-world (*ie*, non-generational, non-incremental), generational only, and incremental/generational. We refer to these as STW, GEN, and INCGEN, respectively. We do not consider an incremental/non-generational collector because it is unlikely one would forego the benefits of generational collection when the write barrier support needed for it is available, as it is with incremental GC.

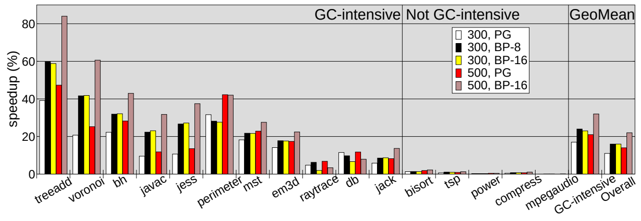
5.1 Speedup

Speedups for PG and BP appear in Figure 5, for each of the three GC configurations. To show the effect of increasing memory latency, we report results for memory latencies of 300 and 500 cycles. The graphs show elapsed time speedup as percentage improvement over the respective base case without prefetching. Benchmarks are sorted from left to right in decreasing order of GC-intensiveness measured as the proportion of total time that is spent tracing. We compute mean speedup for benchmarks that have high GC-intensiveness (greater than 5% of total execution time) in order to focus on benchmarks for which prefetching in the mark phase can have any significant impact. Note that, for the incremental/generational collector, marking is interleaved with mutator activity, but that GC-intensiveness considers only marking overheads.

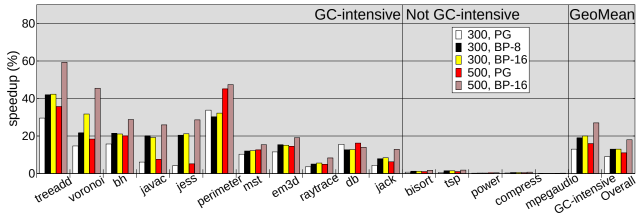
Each set of bars represents a benchmark. Within a set, the first bar represents PG. The second and third bars represent BP with queue sizes of 8 (BP-8) and 16 (BP-16), respectively. The first three bars are for 300-cycle memory latency, normalized against a non-prefetching base case with 300-cycle memory latency. The fourth and fifth bars represent PG and BP-16, both for 500-cycle memory latency, normalized against a non-prefetching base case with 500-cycle memory latency. The mean speedups represent the geometric mean of speedups for the GC-intensive benchmarks and all benchmarks, respectively.



(a) STW



(b) GEN



(c) INCGEN

Figure 5: Speedup

We see that BP performs better than PG in most cases. For 300-cycle memory latency, BP-8 achieves 28%, 24%, and 19% mean speedups on GC-intensive benchmarks for STW, GEN, and INCGEN, respectively. BP-16 achieves 28%, 23%, and 20% mean speedups, respectively. These speedups are substantially higher compared to PG, with 21%, 17%, and 13%.

With increased latency of 500 cycles, because the L2-cache-miss penalty consumes a larger portion of total execution time, both algorithms achieve higher speedups, though BP-16 outperforms PG. BP-16 (fifth bar) achieves 40%, 32%, and 27% mean speedups on GC-intensive benchmarks for STW, GEN, and INCGEN, respectively. These are substantially higher than for PG (fourth bar), which achieves 27%, 21%, and 16%. This shows that BP-16 is more effective in reducing cache misses for longer memory latency than PG.

Trends across GC configurations are similar, though we note that GEN and INCGEN, which both collect only a portion of the heap at each GC cycle have less opportunity for speedup, since marking consumes a smaller fraction of total time. Also, note that INCGEN which need not complete a full GC mark cycle before the benchmark terminates, has correspondingly less opportunity for improvement.

Because trends among GCs are similar, we focus our discussion for individual benchmarks on GEN. With 300-cycle latency, treeadd, voronoi and bh obtain speedups of 60%, 42% and 32%, respectively, for BP-8, compared to 39%, 21% and 22% for PG.

Increasing the latency to 500 cycles causes treeadd, voronoi and bh to obtain higher speedups of 84%, 61% and 43%, respectively for BP-16, compared with 47%, 25% and 28% for PG. When comparing ratios between these speedups, treeadd, voronoi and bh achieve 15%, 13% and 8% better speedup for BP-16 going from 300- to 500-cycle latency, compared to the lower 6%, 4% and 5% for PG. Because the mark phase is not a significant fraction of total execution time for compress, mpegaudio, bisort, tsp and power, prefetching does not improve performance for these benchmarks. At 300-cycle latency, db and perimeter perform better for PG, achieving speedups of 11% and 32%, compared to 10% and 28% with BP-8. We discuss these degradations in the next section. None of the benchmarks we simulated has performance lower than its non-prefetching base case.

5.2 Detailed comparisons

We now show detailed comparisons between PG and BP-16 for execution times, and statistics for STW, GEN, and INCGEN GC configurations.

To show the potential performance improvements for different prefetching algorithms, we implement an oracle simulation which we call perfect prefetching. This treats the matching load for every prefetch as if it is a hit (*ie*, as if the prefetch was perfectly timed). The implementation simply tags the target cache block for each prefetch, and makes that block immediately available in L1 and L2 caches when a demand-load requests it. The tags are cleared after the matching demand-load. Because the tags are maintained separately from the cache simulation data structures, they are not affected by replacement events in the caches. Notice that perfect prefetching does not show the potential for eliminating cache misses in marking, because there are misses that are not covered by prefetching, including misses that occur because their cache blocks are evicted after first use despite being prefetched earlier. We only show perfect prefetching for PG and BP-16, both with gap-prefetching.

Figure 6 shows normalized overall execution times. Each set of bars represent a benchmark. Within a set, the first bar shows execution time for the non-prefetching base case without prefetching. The second and third bars represent PG without gap-prefetching and PG with gap-prefetching, respectively. The fourth bar shows perfect PG. The fifth and sixth bars correspond to BP-16 without gap-prefetching and BP-16 with gap-prefetching. The seventh bar shows perfect BP-16. All the bars are normalized to the same non-prefetching base case (the first bar).

Each bar has two parts: the lower shaded part represents marking overhead (*ie*, the portion of execution time that prefetching directly impacts) shown as a fraction of total execution time. The upper white part corresponds to the non-marking portion of execution time (including both mutator code and sweeping).

From the first bar in Figure 6, we see that marking overhead is a significant portion of total execution time for many benchmarks. Using GEN as an example, marking constitutes as much as 70% for voronoi, 60% for treeadd, 53% for javac, and 44% for jess. It is this significant overhead that represents any opportunity for improving performance using prefetching during marking. Section 5.1 mentions that prefetching cannot improve compress, mpegaudio, bisort, tsp and power because they are not GC-intensive. Figure 6 illustrates that their GC marking overhead is less than 5% of total execution time. Other benchmarks have marking overhead ranging from 17% to 70%.

We now turn to the impact of gap-prefetching. Comparing PG without gap-prefetching (second bar) versus PG with gap-prefetching (third bar), there is clear improvement. For example, gap-prefetching improves javac from 10% to 17%, raytrace from 19% to 28%, treeadd from 8% to 39% and voronoi from 17% to 24% in STW GC. Except for bh in GEN, and javac, jess, and jack, in INCGEN, which see 1-3% degradation, most benchmarks with PG gain or maintain speedups with the addition of gap-prefetching. Similarly, BP-16 (sixth bar) also benefits from gap-prefetching, showing reduction in execution times from BP-16 without gap prefetching (fifth bar).

Now, compare PG (third bar) with BP-16 (sixth bar). As shown earlier, BP-16 performs better than PG. Figure 6 shows that these improvements come from the reduction in marking overhead. Comparing these prefetching algorithms to their perfect prefetching cases, PG's performance is often far away from PG perfect (fourth bar), while BP-16's performance is often closer to BP-16 perfect (seventh bar). Summarizing, BP-16 achieves 93%, 96% and 95% of its potential across all benchmarks in STW, GEN and INCGEN configurations, respectively. In contrast, PG achieves only 85%, 83% and 87% of its potential, respectively.

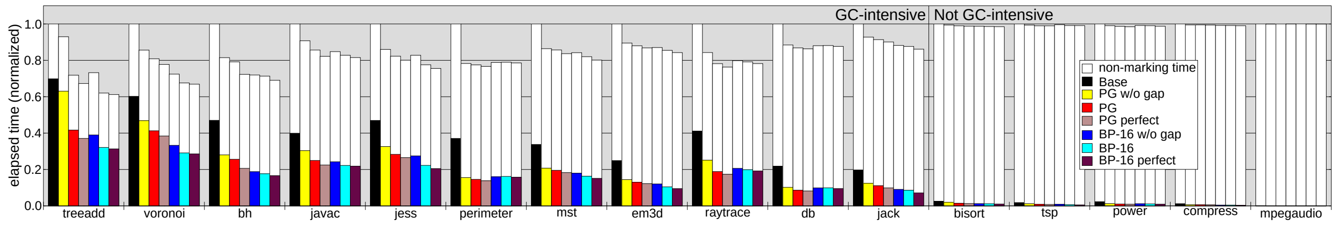
One interesting observation in Figure 6 is that perfect BP-16 performs better than perfect PG in many cases. We attribute this to improved instruction-level parallelism for BP-16's hybrid depth-first/breadth-first traversal (strict depth-first must chase dependent pointers while breadth-first follows independent sibling pointers).

Table 3 shows the timing of prefetching, categorizing prefetches as: *late* if the load latency experienced by the corresponding demand loads are larger than the L2 latency, *timely* if the load latency experienced is smaller than the L2 latency, and *early* if the prefetched block is neither in-flight nor in L1/L2 caches. To put things in perspective, too early prefetches typically incur more penalty because they fully expose memory latency to demand loads. The table shows the percentage of all prefetches occurring in each category for each benchmark/GC combination.

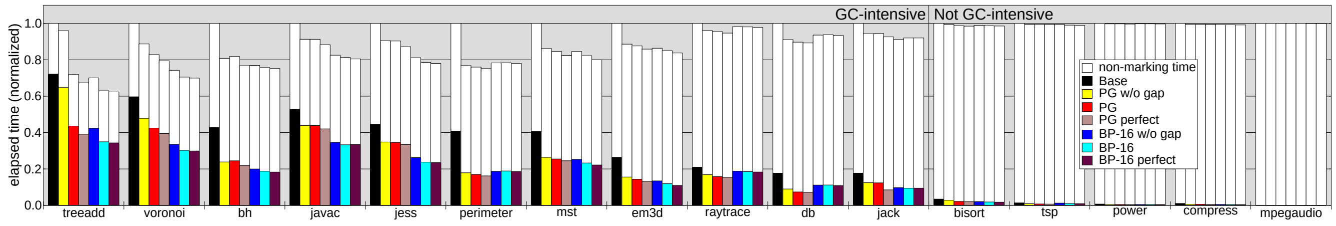
Table 3 shows that a large fraction of prefetches using PG are either early or late, whereas BP-16 successfully minimizes early/late prefetches to increase the percentage of timely prefetches. In the case of GEN (Figure 6(b)), javac's timely prefetches increase from 68% with PG to 92% for BP-16. Similarly, jess increases from 71% to 94%, voronoi increases from 76% to 96%, and treeadd increases from 91% to 96%. In db's case, timely prefetches do not increase significantly and BP-16 pays the price of instruction overhead, resulting in performance degradation.

Table 4 lists the GC statistics for PG and BP-16. The first column shows execution time for the non-prefetching base case. The second column pair shows instruction overhead for PG and BP-16, expressed as a percentage over the non-prefetching base case. The third column pair shows maximum heap size for PG and BP-16, respectively. Note that PG and the base case are the same since they trace objects in the same order. BP-16 can differ since different tracing orders can produce differences in GC behavior because of BDW's treatment of ambiguous roots. The fourth column pair shows the percentage increase in *non-marking* time with respect to the base case for PG and BP-16, respectively. The fifth column pair gives the number of times the collector is invoked (*ie*, number of mark phases) for PG and BP-16, respectively. Table 4 shows that BP-16 tends to have higher instruction overhead than PG, because of the extra code to manage the buffer. Because BP-16 eliminates more cache misses than PG, BP-16 is able to absorb the cost of these extra instructions and still improve performance.

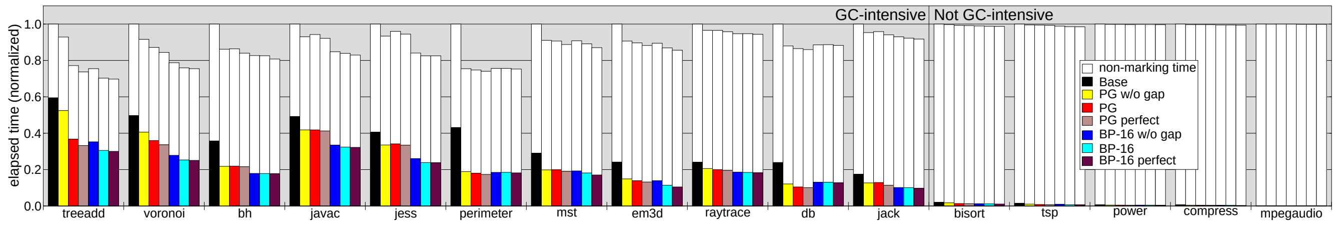
Looking at maximum heap sizes we note that the base case and PG always have the same traversal order so their heap footprints are always the same. Similarly, STW collectors always process the whole heap without interleaving mutator activity, so the base case, PG and BP-16 all have the same footprint. Differences exist only



(a) STW



(b) GEN



(c) INCGEN

Figure 6: Normalized execution time

Table 3: Prefetch timing

	STW						GEN						INCGEN					
	PG			BP			PG			BP			PG			BP		
	E	*T*	*L*	*E*	*T*	*L*	*E*	*T*	*L*	*E*	*T*	*L*	*E*	*T*	*L*	*E*	*T*	*L*
treeadd	1	91	8	3	96	1	1	91	8	3	97	1	1	91	8	4	96	1
voronoi	6	74	20	1	97	2	8	76	16	2	96	1	10	76	14	3	96	1
bh	12	73	16	5	90	5	11	76	13	6	92	2	12	75	13	8	90	2
javac	7	82	11	5	93	2	24	68	8	6	92	1	27	66	7	7	92	1
jess	8	76	16	2	88	10	24	71	5	3	94	3	24	72	5	3	94	4
perimeter	11	84	5	11	86	3	11	84	4	12	86	2	10	86	4	12	87	2
mst	9	87	5	8	83	8	6	90	4	6	89	5	8	88	4	9	84	6
em3d	18	69	13	10	65	25	16	70	14	9	70	21	18	73	9	11	73	17
raytrace	8	86	6	5	92	3	27	69	4	10	86	4	26	70	4	9	86	4
db	7	89	4	9	86	6	11	88	1	10	87	3	8	88	4	11	86	3
jack	8	82	10	1	76	23	10	80	10	2	92	6	10	80	10	2	91	7
bisort	3	89	8	4	90	7	2	92	6	3	93	4	2	92	6	5	92	4
tsp	7	79	13	2	65	33	4	86	10	2	81	17	4	86	10	2	79	20
power	14	81	4	10	78	12	19	73	8	9	79	12	16	76	7	7	82	11
compress	15	70	15	0	43	56	10	75	15	1	65	33	9	76	15	2	67	32
mpegaudio	14	68	19	0	36	64	3	79	18	1	61	38	3	82	15	2	66	32

$E = \%\text{early}, T = \%\text{timely}, L = \%\text{late}$
$(E + L + T = 100\%)$

for BP-16 when mutator activity is interleaved with partial GCs (both GEN and INCGEN); heap footprints can differ somewhat because BP-16 traverses the generations/increments in a different order than base/PG.

The non-mark column shows that times spent in the application (mutator) and for sweeping (*ie*, not marking), do not vary much from the base case.

Table 5 shows L1 and L2 demand-load miss rates in the GC marking code for STW, GEN and INCGEN. To put things in perspective, because the number of L2 accesses is the number of L1 misses, the reduction in both L1 and L2 miss rates actually reflects a larger reduction in L2 misses. From this table, we can see that BP-16 is more effective than PG in reducing cache misses. This data confirms our expectation that BP-16 reduces L2 misses to improve execution times.

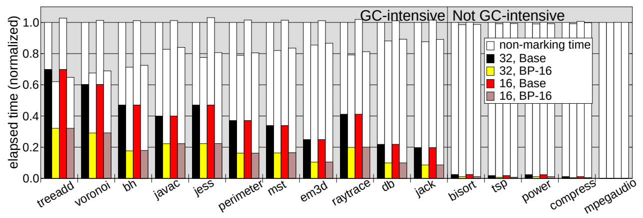
5.3 Effect of number of MSHRs

In modern processors that support non-blocking loads, the number of MSHRs limits the number of outstanding memory requests. Because BP-16 increases the number of memory requests in-flight by issuing prefetches, fewer MSHRs may result in decreased performance. The question is whether BP-16 is still worthwhile with

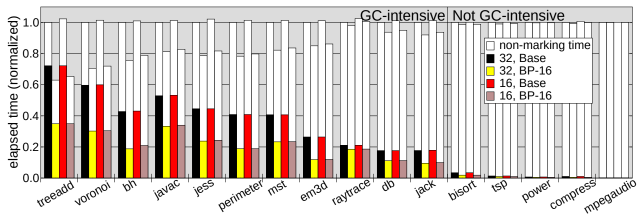
Table 5: Miss rates in the marking code (%)

	STW						GEN						INCGEN					
	L1			L2			L1			L2			L1			L2		
	Base	PG	BP-16	Base	PG	BP-16	Base	PG	BP-16	Base	PG	BP-16	Base	PG	BP-16	Base	PG	BP-16
treeadd	4	1	1	28	7	2	4	1	1	27	7	2	5	1	2	16	5	1
voronoi	5	4	2	26	10	4	5	4	2	24	9	6	6	5	3	18	8	5
bh	7	5	4	28	9	3	9	6	3	30	12	11	10	7	4	29	13	10
javac	6	5	4	25	7	5	8	7	4	21	14	9	8	7	5	19	14	9
jess	7	4	3	27	8	4	7	6	4	21	13	5	8	7	4	17	13	6
perimeter	5	1	2	25	9	2	5	1	2	23	7	2	5	1	2	20	5	1
mst	4	2	1	25	10	2	4	2	2	18	6	2	5	2	2	11	5	2
em3d	5	3	2	32	15	3	6	2	2	26	13	2	6	3	2	19	11	2
raytrace	6	2	2	24	4	2	6	4	3	21	8	5	7	4	3	17	8	6
db	6	3	2	21	7	3	6	3	2	18	6	3	7	3	3	15	4	3
jack	9	4	3	29	14	4	9	6	4	17	9	4	9	6	4	15	9	4
bisort	5	1	1	29	15	2	5	1	1	20	10	2	5	2	2	11	6	2
tsp	9	2	1	31	25	5	8	2	2	16	12	2	8	2	3	12	9	2
power	7	3	2	27	13	2	8	4	3	15	7	3	9	5	3	10	4	3
compress	10	4	1	37	29	14	10	3	3	33	27	7	10	3	3	27	22	7
mpegaudio	11	4	1	39	32	14	11	3	3	28	23	5	10	4	3	18	15	5

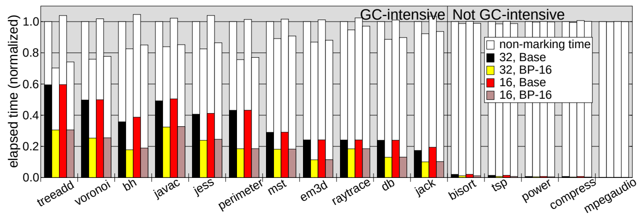
Base = non-prefetching, PG = prefetch-on-grey, BP-16 = 16-entry buffered-prefetch



(a) STW



(b) GEN



(c) INCGEN

Figure 7: Varying the number of MSHRs

fewer MSHRs. In this section, we vary MSHR count, considering 32 and 16 entries for both L1 and L2 cache.

Similar to the previous section, Figure 7 shows normalized execution time. Again, GC marking overhead appears as a fraction of total time. The first and second bars show execution time for the base case and BP-16 with 32 MSHRs. The third and fourth bars show execution time for the base case and BP-16 with 16 MSHRs. All the bars are normalized to the same base case with 32 MSHRs.

We see that both the base case and BP-16 have longer execution time with only 16-entry MSHRs. This trend is not surprising because the machines are more exposed to cache misses when MSHRs are exhausted, which happens in both marking and non-marking portions of execution. However, even on a machine with only 16 MSHRs, BP-16 is able to achieve speedups that are comparable to those achieved on a machine with 32 MSHRs. This shows that BP-16 does not require a large number of MSHRs to achieve its performance benefit. Because the BDW GC uses many instructions to mark an object, BP-16 allows the prefetch to sufficiently overlap memory latency with the processing of 16 objects, even if the loads to these objects hit in the cache. As a result, BP-16 does not rely on overlapping many prefetches to reduce cache misses.

5.4 Effect of delaying processing

As mentioned in Section 3.4, BP may cause extra overhead by delaying processing of blocks that hit in the cache. We now compare the performance of BP to a conditional BP algorithm that uses c-loads, to see if deferring tracing via the prefetch buffer hurts performance. As a reminder, a c-load notifies the CPU about the hit/miss status of a block. The program uses this status information to choose between processing the block if the load hits or using BP if the load misses.

Figure 8 shows execution time normalized to the non-prefetching base case similarly to the earlier graphs. The first bar shows execution time for the base case; the second and third bars show execution time for the BP-16 and conditional BP-16 using c-loads, respectively. We see that c-loads performs similarly to BP. This shows that deferral of tracing through the prefetch buffer does not hurt performance. In some cases, c-loads perform worse than BP. We attribute that to the overheads of additional instructions and branch mispredictions.

5.5 Effect of tracing pollution

As mentioned in Section 2.1, GC tracing may pollute the caches with data not needed by the mutator. To expose the overhead of tracing pollution, we simulated an idealized machine that has two L2 caches, each with the same size/associativity as for the base case. The simulator accesses both L2 caches for each L1 miss, but pollutes only one of the L2 caches with tracing accesses. The simulator treats an access that hits in either L2 cache as a hit, so the run benefits from tracing locality but does not suffer from tracing pollution. Table 6 shows the resulting speedups for the base case when tracing pollution is avoided in this way. We see only 2-3% overall improvement for GC-intensive benchmarks. This shows that tracing pollution is not a problem for a machine with a large L2 cache.

Table 4: GC statistics

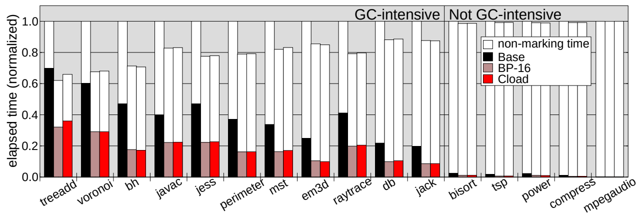
		Cycles ($\times 10^9$)	Insn. overhead (%)		Max. heap (MB)		non-mark time overhead (%)		GCs (count)	
		Base	PG	BP-16	PG	BP-16	PG	BP-16	PG	BP-16
STW	treeadd	0.5	2	15	44	44	0	-1	10	10
	voronoi	1.7	2	7	33	33	-1	-3	15	15
	bh	10.8	2	5	5	4	1	1	335	338
	javac	6.3	1	4	31	31	1	1	37	37
	jess	5.0	2	3	7	7	2	4	233	233
	perimeter	0.7	4	6	28	28	0	0	9	9
	mst	0.1	3	6	5	5	0	-1	5	5
	em3d	0.4	1	1	12	12	0	0	7	7
	raytrace	3.8	2	5	11	11	1	1	64	64
	db	0.5	2	5	15	15	0	0	6	6
	jack	6.1	1	1	3	3	0	-2	285	285
	bisort	1.5	0	0	7	7	0	0	5	5
	tsp	0.9	0	0	2	2	0	0	5	5
	power	7.1	0	0	3	3	0	0	30	30
	compress	1.2	0	0	16	16	0	0	6	6
	mpegaudio	3.1	0	0	4	4	0	0	2	2
GEN	treeadd	0.6	2	16	44	44	2	1	28	28
	voronoi	1.7	2	5	33	33	0	0	43	43
	bh	10.9	2	5	5	5	0	-1	658	664
	javac	8.3	2	5	48	44	0	2	72	72
	jess	5.6	2	3	5	5	1	-1	663	665
	perimeter	0.7	4	7	28	28	0	1	26	26
	mst	0.1	4	9	5	5	-1	-1	16	16
	em3d	0.4	2	2	13	13	0	-1	21	21
	raytrace	2.8	1	2	8	8	1	1	163	163
	db	0.5	2	12	15	15	0	0	12	12
	jack	6.1	0	1	3	3	0	0	588	601
	bisort	1.5	0	0	7	7	0	0	17	17
	tsp	0.9	0	1	2	2	0	-1	16	17
	power	7.0	0	0	2	2	0	0	88	88
	compress	1.2	0	0	16	16	0	0	10	10
	mpegaudio	3.1	0	0	4	3	0	0	3	3
INCGEN	treeadd	0.4	2	15	42	41	0	-2	11	11
	voronoi	1.3	2	3	39	37	2	1	18	17
	bh	9.4	1	6	8	8	0	1	388	406
	javac	8.0	2	4	52	51	3	1	38	38
	jess	5.1	2	3	6	6	4	-1	410	404
	perimeter	0.7	4	4	29	29	0	1	19	19
	mst	0.1	3	6	5	5	0	0	12	12
	em3d	0.4	1	1	12	12	0	0	16	16
	raytrace	3.0	1	1	11	11	1	0	127	127
	db	0.5	2	8	15	15	0	-1	8	8
	jack	5.9	0	2	5	4	0	-1	403	459
	bisort	1.4	0	0	6	7	0	0	6	8
	tsp	0.9	0	0	2	2	0	-1	11	11
	power	7.0	0	0	2	2	0	0	77	75
	compress	1.2	0	0	16	16	0	0	7	7
	mpegaudio	3.1	0	0	4	4	0	0	3	3

Base = non-prefetching, PG = prefetch-on-grey
BP-16 = 16-entry buffered-prefetch

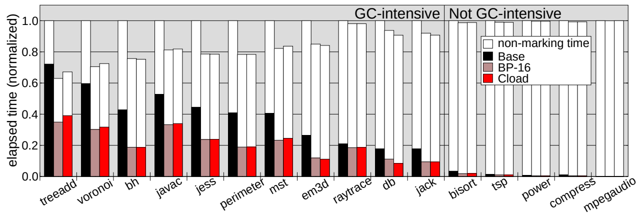
5.6 Live runs

We have validated our simulation-based results by running the benchmarks on a real machine. The platform used is an Apple G5 PowerPC 970 (see hardware details in Table 7), running 32-bit PowerPC Linux 2.6.3 (with benh patches) in single-user mode. We compiled the benchmarks using gcj 3.2.3, and ran the BDW collector in its out-of-the-box STW configuration, varying only the free-space-divisor (FSD), and adding prefetching for PG and BP using the PowerPC dcbt "data-cache-block-touch" instruction to prefetch into L1. Because memory latency for current machines is less than what we simulated, we found that tuning the prefetch buffer for size 4 (BP-4) produced best results.

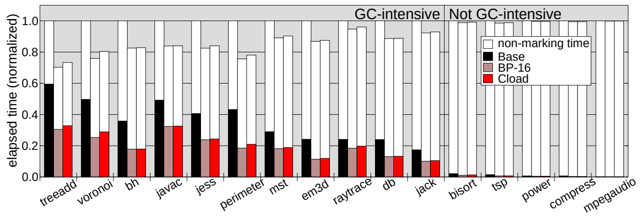
Speedups with FSD 3 (the BDW default used in our simulations) for PG and BP-4, and with FSD 10 for PG and BP-4, are shown in Figure 9. The FSD parameter to the BDW collector controls the rate of GC by stating the ideal free-space that a given GC should



(a) STW



(b) GEN



(c) INCGEN

Figure 8: Impact of capitulating loads

Table 6: Speedup without tracing pollution (%)

	STW	GEN	INCGEN
treeadd	2	1	2
voronoi	4	3	3
bh	7	1	4
javac	5	3	4
jess	7	5	3
perimeter	2	2	2
mst	4	2	4
em3d	1	1	1
raytrace	1	1	2
db	1	1	1
jack	2	2	1
bisort	0	0	0
tsp	0	0	0
power	0	0	0
compress	0	0	0
mpegaudio	0	0	0
GeoMean			
GC-intensive	3.2	2.0	2.4
Overall	2.1	1.4	1.7

Table 7: Live hardware parameters

Processor	PowerPC 970 (Apple G5), 1.8GHz, 8-way issue, 200 maximum instructions in flight
Caches	64KB direct-mapped instruction L1, 32KB 2-way associative data L1, 512KB 8-way unified L2, 8 L1 and 8 L2 MSHRs
Memory	512MB
Prefetch	Prefetch into L1 using dcbt instruction

obtain after each collection cycle as a fraction of total heap size. Thus, FSD 3 (the BDW default) states that GC should free up a third of the heap at each GC, expanding the heap as necessary only

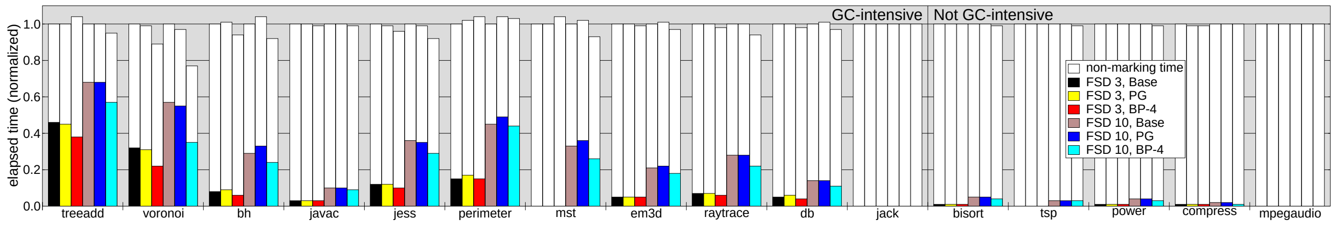


Figure 10: Normalized execution time: live runs

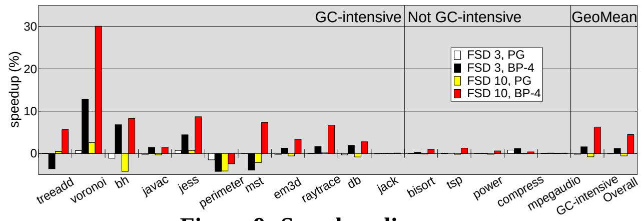


Figure 9: Speedup: live runs

to achieve this goal. The higher the FSD, the more frequently the collector will run. In effect, the FSD parameter controls the trade-off of space (smaller heap sizes) for time (more frequent collections). Any benefits obtained from prefetching while marking will be more pronounced with more frequent invocations of the collector (*ie*, higher FSD).

The superior performance of BP over PG is evident in Figure 9. Indeed, slowdown predominates for PG at both FSD 3 and FSD 10 (first and third bars), showing marginal mean slowdown at FSD 3 (-0.19% for GC-intensive benchmarks and -0.08% overall) and noticeable slowdowns for FSD 10 (-0.79% GC-intensive and -0.58% overall). In stark contrast, BP-4 (second and fourth bars) shows noticeable mean speedup for both GC-intensive benchmarks and overall, with marginal speedup for FSD 3 (1.57% GC-intensive and 1.17% overall) but significant speedup at FSD 10 (6.25% GC-intensive and 4.47% overall).

Detailed comparisons between PG and BP-4 for normalized execution times appear in Figure 10. Again, we highlight the fraction of total execution time spent in the marking phase of GC on which prefetching has impact. The first three bars give results with FSD 3 for the base, PG and BP-4 configurations, while the fourth through fifth bars show results for FSD 10. For all benchmarks, BP-4 always improves mark time over the base case, though in some cases we see increased total execution time (*ie*, slowdown) despite the reduction in mark time. We attribute this to pollution effects of marking on sweeping and the mutator, in contrast to our simulation results. Note that this observation is consistent with the simulation results in Section 5.5 which show some benchmarks gaining up to 7% improvement when pollution is avoided.

6. RELATED WORK

We focus discussion of related work on software-based prefetching techniques for linked data structures, noting also the existence of a vast literature on hardware prefetching and array prefetching that is not directly related to our work. Prior work for linked data structures has focused on general techniques for improving accesses by *general* programs. Some apply *jump-pointer* techniques, which place prefetch hints in the form of object pointers within the linked data structures [16, 17, 18, 7]. Thus, for example, a linked list node might contain additional pointers to nodes beyond simply the next node in the list, in addition to its own data payload. When traversing the list, it is then possible to prefetch some set of nodes

Table 8: GC statistics: live runs

		Time (s)	Max. heap (MB)		non-mark time overhead (%)		GCs (count)	
		Base	PG	BP-4	PG	BP-4	PG	BP-4
FSD 3	treeadd	0.4	20	20	1	21	5	5
	voronoi	2.1	21	20	0	-2	8	8
	bh	8.1	11	11	0	-5	58	58
	javac	27.9	24	24	0	-1	19	19
	jess	7.2	8	8	0	-2	63	63
	perimeter	0.5	15	15	0	6	4	4
	mst	0.1	2	2	0	4	1	1
	em3d	0.6	8	8	0	0	3	3
	raytrace	6.8	9	9	0	-1	24	24
	db	0.7	10	10	0	-1	3	3
	jack	184.8	6	6	0	0	40	40
	bisort	1.2	3	3	0	0	2	2
	tsp	0.9	2	2	0	0	1	1
	power	8.8	3	3	0	0	8	8
	compress	1.7	13	13	-1	-1	5	4
	mpegaudio	3.8	4	4	0	0	2	2
FSD 10	treeadd	0.8	18	18	0	18	16	16
	voronoi	3.5	16	16	0	-2	29	29
	bh	11.0	3	3	0	-5	314	270
	javac	30.2	15	14	0	0	59	67
	jess	10.1	3	3	0	-1	300	301
	perimeter	0.8	15	15	0	6	15	15
	mst	0.1	2	2	0	0	5	5
	em3d	0.7	8	8	0	0	11	11
	raytrace	8.9	4	5	0	0	125	117
	db	0.7	10	10	0	-1	8	8
	jack	185.4	3	3	0	0	110	115
	bisort	1.2	2	2	0	0	6	6
	tsp	1.0	1	1	0	-1	5	5
	power	9.1	1	1	0	0	40	40
	compress	1.7	14	14	0	0	6	6
	mpegaudio	3.8	4	4	0	0	2	2

ahead of the current node. Automatic derivation of jump-pointers and placement of prefetch instructions by optimizing compilers are important contributions [16, 7]. Other software techniques have not relied on such support [15, 19, 14, 23].

In comparison with jump-pointer prefetching approaches that use a queue to remember application data-access orders and then construct a prefetch order from the queue, with our approach we use a queue to remember the prefetch order and then later construct the data-access order from this queue. The reason is that in a general pointer-based program one cannot change the data-access order, but only the prefetch order. Here, we change both data-access order and prefetch order, giving us more control over timing. Thus, in contrast to prior work that addresses general programs, we obtain detailed analysis of prefetching for a particular algorithm, namely tracing GC, and use that analysis to drive redesign of the algorithm for significant performance improvement.

7. CONCLUSIONS

Memory accesses made during garbage collection (GC) exhibit significantly less locality than typical programs and incur considerable overhead due to cache misses. Using simulations of standard Java benchmarks on a projected hardware platform, we found that as much as 60% of program time may be spent in tracing in the Boehm-Demers-Weiser (BDW) garbage collector. While Boehm's prefetch-on-grey technique for BDW reduces this overhead by 16% on average with incremental/generational GC for GC-intensive benchmarks, our results showed that prefetch-on-grey is less than ideal. Further analysis revealed that prefetch-on-grey suffers from lack of timeliness – many prefetches are too early or too late with respect to the corresponding access. The key reason is that while prefetch-on-grey issues prefetches in FIFO order the accesses occur in LIFO ordering, causing prefetches to be separated from accesses by arbitrary amounts of time. By processing a small FIFO window at the top of the LIFO BDW stack we successfully control the timing between prefetch and access. Our simulation results show that this approach achieves 27% average speedup and up to three times the speedup of prefetch-on-grey for GC-intensive benchmarks, virtually eliminating misses in tracing accesses for some benchmarks. Moreover, buffered prefetching produces significant speedup on current hardware of 6% on average for a GC configuration that tightly controls heap growth, whereas Boehm's prefetch-on-grey yields no noticeable improvement.

Acknowledgments

This work is supported by the National Science Foundation under grants Nos. CCR-9711673, IIS-9988637, CCR-0085792, CCR-9875960 (CAREER), CCR-9986020 (Instrumentation), by the Defense Advanced Research Program Agency, and by gifts from Sun Microsystems and IBM.

8. REFERENCES

[1] APPEL, A. W., AND LI, K. Virtual memory primitives for user programs. In *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Santa Clara, California, Apr.). *ACM SIGPLAN Notices 26*, 4 (Apr. 1991), pp. 96–107.

[2] AUSTIN, T. M., LARSON, E., AND ERNST, D. SimpleScalar: An infrastructure for computer system modeling. *IEEE Computer 35*, 2 (Feb. 2002), 59–67.

[3] BOEHM, H.-J. Space efficient conservative garbage collection. In *Proceedings of the ACM Conference on Programming Language Design and Implementation* (Albuquerque, New Mexico, June). *ACM SIGPLAN Notices 28*, 6 (June 1993), pp. 197–206.

[4] BOEHM, H.-J. Reducing garbage collector cache misses. In *Proceedings of the ACM International Symposium on Memory Management* (Minneapolis, Minnesota, Oct., 2000). *ACM SIGPLAN Notices 36*, 1 (Jan. 2001), pp. 59–64.

[5] BOEHM, H.-J., DEMERS, A. J., AND SHENKER, S. Mostly parallel garbage collection. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications* (Phoenix, Arizona, Oct.). *ACM SIGPLAN Notices 26*, 11 (Nov. 1991), pp. 157–164.

[6] BOEHM, H.-J., AND WEISER, M. Garbage collection in an uncooperative environment. *Software—Practice and Experience 18*, 9 (Sept. 1988), 807–820.

[7] CAHOON, B., AND MCKINLEY, K. S. Data data flow analysis for software prefetching linked data structures in Java. In *Proceedings of IEEE International Conference on Parallel Architectures and Compilation Techniques* (Barcelona, Spain, Sept.). 2001, pp. 280–291.

[8] CAHOON, B. D. *Effective Compile-Time Analysis for Data Prefetching in Java*. PhD thesis, University of Massachusetts at Amherst, Sept. 2002.

[9] DIJKSTRA, E., LAMPORT, L., MARTIN, A., SCHOLTEN, C., AND STEFENS, E. On-the-fly garbage collection: An exercise in cooperation. *Commun. ACM 21*, 11 (Nov. 1978), 966–975.

[10] GOSLING, J., JOY, B., STEELE, JR., G., AND BRACHA, G. *The Java Language Specification*, second ed. Addison-Wesley, 2000.

[11] HOROWITZ, M., MARTONOSI, M., MOWRY, T. C., AND SMITH, M. D. Informing memory operations: Memory performance feedback mechanisms and their applications. *ACM Trans. Comput. Syst. 16*, 2 (May 1998), 170–205.

[12] HUGHES, R. J. M. A semi-incremental garbage collection algorithm. *Software—Practice and Experience 21*, 11 (Nov. 1982), 1081–1084.

[13] JONES, R., AND LINS, R. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. Wiley, May 1996. Chapter on distributed collection by Lins.

[14] KARLSSON, M., DAHLGREN, F., AND STENSTRÖM, P. A prefetching technique for irregular accesses to linked data structures. In *Proceedings of the International Symposium on High Performance Computer Architecture* (Toulouse, France, Jan.). IEEE Computer Society, 2000, pp. 206–217.

[15] LIPASTI, M. H., SCHMIDT, W. J., KUNKEL, S. R., AND ROEDIGER, R. R. SPAID: Software prefetching in pointer- and call-intensive environments. In *Proceedings of the International Symposium on Microarchitecture*. ACM/IEEE, 1995, pp. 231–236.

[16] LUK, C.-K., AND MOWRY, T. C. Compiler-based prefetching for recursive data structures. In *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Cambridge, Massachusetts, Oct.). *ACM SIGPLAN Notices 31*, 9 (Sept. 1996), pp. 222–233.

[17] ROTH, A., MOSHOVOS, A., AND SOHI, G. S. Dependence based prefetching for linked data structures. In *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (San Jose, California, Oct.). *ACM SIGPLAN Notices 33*, 11 (Nov. 1998), pp. 115–126.

[18] ROTH, A., AND SOHI, G. S. Effective jump-pointer prefetching for linked data structures. In *Proceedings of the International Symposium on Computer Architecture* (Atlanta, Georgia, May). *Computer Architecture News 27*, 2 (May 1999), pp. 111–121.

[19] RUBIN, S., BERNSTEIN, D., AND RODEH, M. Virtual cache line: A new technique to improve cache exploitation for recursive data structures. In *Proceedings of the International Conference on Compiler Construction* (Amsterdam, The Netherlands, Mar.), S. Jähnichen, Ed. vol. 1575 of *Lecture Notes in Computer Science*. 1999, pp. 259–273.

[20] SCHKOLNICK, M. A clustering algorithm for hierarchical structures. *ACM Trans. Database Syst. 2*, 1 (Mar. 1977), 27–44.

[21] SPEC. SPECjvm98 benchmarks, 1998. http://www.spec.org/osg/jvm98.

[22] STAMOS, J. W. Static grouping of small objects to enhance performance of a paged virtual memory. *ACM Trans. Comput. Syst. 2*, 2 (May 1984), 155–180.

[23] STOUTCHININ, A., AMARAL, J. N., GAO, G. R., DEHNERT, J. C., JAIN, S., AND DOUILLET, A. Speculative prefetching of induction pointers. In *Proceedings of the International Conference on Compiler Construction* (Genova, Italy, Apr.), R. Wilhelm, Ed. vol. 2027 of *Lecture Notes in Computer Science*. 2001, pp. 289–303.

[24] UNGAR, D. Generation scavenging: A non-disruptive high performance storage reclamation algorithm. In *Proceedings of the ACM Symposium on Practical Software Development Environments* (Pittsburgh, Pennsylvania, Apr.). 1984, pp. 157–167.

[25] WILSON, P. R., LAM, M. S., AND MOHER, T. G. Effective "static-graph" reorganization to improve locality in garbage-collected systems. In *Proceedings of the ACM Conference on Programming Language Design and Implementation* (Toronto, Canada, June). *ACM SIGPLAN Notices 26*, 6 (June 1991), pp. 177–191.

[26] ZILLES, C. B. Benchmark HEALTH considered harmful. *ACM SIGARCH Newsletter 29*, 3 (June 2001), 4–5.