Portable, Mostly-Concurrent, Mostly-Copying Garbage Collection for Multi-Processors

Antony L Hosking

Department of Computer Science
Purdue University
West Lafayette, IN 47907, USA
hosking@cs.purdue.edu

Abstract

Modern commodity platforms increasingly support thread-level parallelism, which must be exploited by garbage collected applications. We describe the design and implementation of a portable *mostly-concurrent* mostly-copying garbage collector that exhibits scalable performance on multi-processors. We characterize its performance for heap-intensive workloads on two different multi-processor platforms, showing maximum pause times two orders of magnitude shorter than for fully stop-the-world collection at the cost of some total mutator throughput.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors—code generation, compilers, memory management (garbage collection), run-time environments

General Terms Algorithms, measurement, performance, design, experimentation, languages

Keywords Garbage collection, memory management, conservative, ambiguous-roots, incremental, concurrent, mostly-copying, mostly-concurrent, portability

1. Introduction

Copying garbage collectors have several salient advantages. They permit fast allocation by incrementing a pointer in the allocation space, and compaction of live objects for better locality and to minimize fragmentation. *Incremental* collectors permit the interleaving of mutator (i.e., application) and collector activity to improve responsiveness for mutators by reducing pause times for interactive or other (soft) real-time activities. Incrementality also translates into concurrency, since mutators can cooperate on advancing garbage collection (GC) along with dedicated background collector threads. *Mostly-copying* incremental collection combines these benefits for settings where root references are ambiguous, such as when uncooperative compilers neglect to provide precise information on the location of roots in thread stacks and registers, and for situations where certain objects must be pinned for a time at a fixed location in memory. These situations often arise in systems-oriented programming languages such as C, C++, Modula-3, and C#.

ISMM'06 June 10–11, 2006, Ottawa, Ontario, Canada

With thread-level parallelism via now increasingly available in commodity hardware intended for use on the desktop (e.g., simultaneous multi-threading (SMT) or "hyper-threading", multi-core processors, and chip multi-processors (CMP)), it is increasingly important that memory management is neither a concurrency bottleneck nor a hindrance to exploitation of concurrency by other application threads.

Here, we report on steps taken to retool a uniprocessor incremental mostly-copying collector for efficient execution on multi-processors, and quantify the resulting improvements in pause times and impact on throughput, as well as its scalability for multi-processor execution. We believe that our prototype is the first portable collector in the class of mostly-copying collectors also to be *mostly-concurrent*: having only a brief stop-the-world phase, after which mutator threads can run concurrently. Our implementation is portable, relying only on existing system-level POSIX thread primitives.

2. Mostly-copying GC

Copying garbage collectors operate by *evacuating* live objects as they are discovered, to a new heap space; dead objects remain behind in the old space, which can then be freed wholesale. Live objects are those reachable by tracing the transitive closure from program *roots* (e.g., registers, global variables, and thread stacks).

Mostly-copying garbage collection [8, 9, 17, 41] is a hybrid of conservative [13] and copying [25, 15] collection. It is suitable for use in environments lacking accurate information on the location of references from the register, static or stack areas; objects that *appear* to be referenced from these areas are treated conservatively and not moved. Such references are called *ambiguous roots*, since they have a bit pattern that coincides with the range of valid heap references. In addition to the usual *tidy* language-level object references, which always refer to object headers, ambiguous roots may also include *derived* references that arise out of pointer arithmetic introduced by compiler optimizations or explicitly by the programmer in languages that permit such expression. Ambiguous roots also serve as a useful mechanism for *pinning* objects at fixed locations in memory for the lifetime of the ambiguous root. Mostly-copying collectors thus permit heap references to be passed to non-GC-aware code, so long as the reference is held in a register or on the stack for the duration of the call, causing it to be treated as an ambiguous root. This simplifies composition of programs from both GC-aware and GC-oblivious code.

Mostly-copying garbage collectors do require that all pointers stored in heap-allocated objects be tidy and can be found accurately; objects accessible only from other heap objects can thus be moved during GC. Accurately finding the source locations of heap

pointers requires information describing the layout of heap objects. The compiler may generate such information directly or it may be supplied manually by the programmer [9], though this approach may be error-prone.

For mostly-copying collection the heap is divided into a number of fixed-size *pages*, which are usually some fixed multiple of virtual memory pages. Aligning the heap pages appropriately gives each page a unique page number formed from the common high-order bits of the virtual addresses covered by the page. This permits efficient mapping of heap references to per-page information. Objects larger than a single heap page are allocated as a sequence of consecutive heap pages.

We find it convenient to use Dijkstra's *tri-color* abstraction [19] in describing mostly-copying collection. Pages in the heap (and by implication the objects in them) are "shaded" with one of three colors:

Black pages contain only live objects, and the references contained in those objects do not refer to objects in white pages.

Grey pages contain only live objects, but the references in those objects may still refer to objects in white pages.

White pages contain objects that have not yet been *evacuated* to a non-white page; at the end of collection these pages are garbage and can be reclaimed.

New objects are always allocated in black pages. When the heap is "full" (e.g., some allocation threshold is reached) all the black pages are shaded white. The collector then proceeds to evacuate all reachable objects from white pages to grey pages. The pages of each color are not necessarily contiguous and pages of each color may be interleaved. This arrangement allows an object to be moved by the collector from white to grey either by physically copying it to a grey page, or simply by shading its current page grey. The latter mechanism is called page *promotion*. Objects that appear to be referenced by ambiguous roots can thus logically be "moved" by promoting their page; retaining the same virtual address preserves the integrity of the ambiguous reference. Large objects are also logically "moved" via page promotion, to reduce the copying overhead of the collector.

The mostly-copying collector, sketched in Figure 1, operates in three phases. We assume that the color spaces are abstracted as sets of pages. The variables p, l and r range over heap pages, heap pointer locations and heap pointers (references), respectively. The heap pointer stored at a given heap pointer location l is denoted $l\uparrow$. $\mathcal{AR}$ denotes the set of ambiguous roots; for simplicity we assume these are the only roots, without loss of generality. The auxiliary procedure *promote* recolors a white page to grey. The procedure *trace* performs iterative Cheney-style [15] copying and scanning of the transitive closure of white objects reachable from grey pages. We assume several additional auxiliary procedures:

page(r)**:** returns the the heap page to which heap pointer r refers

pointer_locations(p)**:** returns an *accurate* set of all locations in page p that contain non-nil heap pointers

copy(r)**:** allocates a copy of the object referred to by r; the address of the copy is termed r's *forwarding address*, and denoted by r'

The garbage collector (*gc*) begins by condemning all the current pages of the heap, *flipping* their color from black to white (line 25). All white pages will be reclaimed at the end of collection, unless promoted in the interim. Thus, the collector's job is to evacuate all reachable objects, copying them from the condemned white pages into grey pages. We assume a finite set of ambiguous roots from the registers, stack, and static areas. To preserve these ambiguous roots, the collector must first promote the pages to which they refer (lines 26-28). Note that promotion may retain (i.e., leak) unreferenced garbage objects that just happen to lie in

```
1  proc promote(p, color) ≡
2     white := white\{p};
3     color := color ∪ {p}.
4
5  proc trace() ≡
6     foreach p ∈ grey do
7        grey := grey\{p};
8        foreach l ∈ pointer_locations(p) do
9           move(l)
10       end;
11       black := black ∪ {p}
12    end.
13
14 proc move(l) ≡
15    r := l↑;
16    if page(r) ∈ white then
17       if r' = nil then
18          r' := copy(r);
19          grey := grey ∪ {page(r')}
20       end;
21       l↑ := r'
22    end.
23
24 proc gc() ≡
25    white := black; grey := {}; black := {};
26    foreach r ∈ AR where page(r) ∈ white do
27       promote(page(r), grey);
28    end;
29    trace();
30    foreach p ∈ white do free(p) end.
```

Figure 1. Mostly-copying GC

those pages. Promoting a page means recoloring the page from white to grey for later processing, in which the objects in the promoted page are scanned in the second phase of collection. Scanning of the contents of unreferenced garbage objects in promoted pages is another potential source of leakage, since their contents may contain references that result in retention of other garbage objects. There are techniques for ameliorating such leaks that require mapping ambiguous roots to their corresponding tidy reference or by tagging the sub-regions of each promoted page that are ambiguously-referenced, but they impose additional overheads. We do not consider such techniques here, observing simply that thread stacks are usually sufficiently volatile to mix up the ambiguous root sets enough to avoid significant leakage from one round of GC to the next. There are typically only a couple of ambiguously-referenced pages per thread stack in any round of GC.

The second phase of collection (line 29) copies the transitive closure of reachable white objects into grey pages. It proceeds by scanning the pointer locations of grey pages to discover any that refer to white objects, copying each reachable white object to a grey page and leaving behind a forwarding address, and updating the pointer locations to refer to the grey copies. This is an iterative process that completes only when the grey set is empty (i.e., there are no more objects whose locations need to be scanned for references to uncopied objects). Termination is guaranteed because the closure of reachable objects is finite: each iteration of the trace loop recolors a grey page black, and pages are added to the grey set only when objects are copied to them, so eventually the grey set becomes empty. At the end of this second phase there are no pointers from live (i.e., black) objects to white objects, and all white pages can be freed (line 30).

Note that decreasing the page size to the point where each page holds a single object, and then simply promoting all pages/objects

rather than copying them, causes this algorithm to degenerate to mark-sweep collection.

Mostly-copying collectors have both generational [9] and incremental [18] variations, which we now briefly describe.

2.1 Generational mostly-copying GC

Adding generational collection [46, 35, 36] is straightforward, and relies simply on segregating older pages from newly-allocated younger pages. Older pages can be promoted wholesale at the start of each generational collection, so avoiding copying of their objects in that round of GC. The promoted older pages are shaded grey if a reference has been stored to them since the last collection, on the assumption that the reference may be to a newly-allocated (i.e., young) white object. Otherwise, the page can be shaded black since the page can hold only references to other older (now grey or black) pages. Keeping track of updated older pages requires some sort of *write barrier* to track pointer stores to older pages. One approach is to use virtual memory page protection primitives to record page modifications by making older pages read-only and catching the resulting protection traps with signal handlers that record the update [1, 2]. This approach can be expensive given the overhead of fielding protection traps from the operating system via user-level signal handlers to record updates, and imposes the coarse-grained granularity of virtual memory pages as the unit of discourse [28]. Alternatively, the compiler can insert an explicit barrier at each pointer store, for potentially finer control over granularity.

2.2 Incremental mostly-copying GC

To reduce intrusiveness, *incremental* collectors 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 round of GC. More problematic is the creation of new references *from* already-traced (black) objects to white objects 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.

As described earlier, tracing is the process of partitioning objects into black and white. 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, or from destroying paths from grey objects to such white objects, without the collector knowing about it. There are several possible approaches to ensuring correctness for incremental tracing [38], depending on the style of garbage collector (i.e., copying, mark-sweep, etc.) being used, but for mostly-copying collectors, where stacks may contain ambiguous untidy references, there is essentially one feasible approach:

- Prevent the mutator from acquiring references to white objects (so it can never store one in a black object) [6]

This approach imposes a *read barrier* on heap accesses by the mutator. Accesses to grey pages require action to prevent acquisition by the mutator of a reference to a white object. If the mutator *knows* (i.e., via compile-time type information) that it is loading a heap reference from a grey page then it can check to make sure that the reference does not refer to an object on a white page, and trigger relocation of any such white object to a grey page. If the mutator cannot distinguish accesses that load references from those that do not, then it must treat all accesses to grey pages as needing action to shade the page from grey to black by scanning its references. This latter strategy permits barrier approaches using virtual memory page protection primitives: all grey pages are protected no-access, so that *any* access to a grey page triggers the barrier [1]. To summarize, in either case the read barrier may trigger a small amount of incremental collection before the mutator can proceed.

So long as barriers are used to ensure correctness, incremental tracing can proceed independently of the mutator. One approach piggy-backs a small amount of tracing onto each allocation request, to ensure tracing progress is tied to mutator progress (as measured by allocation) [6]. Alternatively, tracing can run concurrently with the mutator in a separate thread or in parallel on a separate processor [12, 40, 23, 34, 7]. Incremental tracing minimizes intrusiveness by reducing collector pause times, at the expense of synchronization overheads to mediate accesses to heap meta-data (e.g., during incremental GC and when acquiring pages for allocating new objects). As a result, while responsiveness for interactive or other (soft) real-time activities may improve, overall throughput will suffer.

3. Implementation

Our prototype mostly-concurrent mostly-copying collector is based on the incremental collector originally implemented for the Digital Systems Research Center's implementation of Modula-3 [14, 18]. We have extended the incremental collector to function with Modula-3 threads mapped to system-level threads (rather than as threads implemented in user-mode) that are preemptively scheduled on multi-processor platforms, and to make the collector mostly-concurrent.

3.1 Modula-3

Modula3's support for GC recognizes the high degree of safety afforded by automatic storage reclamation, which is achievable even in open runtime environments that allow interaction with non-Modula-3 code. The original SRC implementation has evolved over the years, including the derivative, briefly-commercial, Critical Mass version of Modula-3 called CM3. CM3 is now in the public domain, with ports targeting most popular POSIX (i.e., Unix) and Windows (i.e., Win32) platforms.

There are several language features of Modula-3 that complicate GC. One complication arises from the language definition itself, which permits limited explicit creation of untidy derived references from thread stacks. Both `VAR` and `READONLY` parameter passing modes allow by-reference passing of arguments that designate internal fields of heap objects. Compiler optimizations can also create such references. Diwan et al.[20] describe techniques for *accurate* location and processing of such untidy references from the stack, even for copying GC, but their techniques require intrusive compiler support to propagate derived pointer maps through the compiler optimizer and back-end.

However, CM3 is implemented as a front-end to the gcc compiler, building standard gcc intermediate code trees that are then processed by the *unchanged* optimizers and back-end of gcc. This strategy yields a high degree of portability for CM3 but makes the techniques of Diwan et al.much less feasible, at least from a portability and maintainability perspective since gcc is itself a complicated and evolving compiler. Moreover, decoupling the compiler from the run-time system gives flexibility in back-end implementation for CM3 – this fact has been exploited to build experimental back-ends independent of gcc (e.g., there are both BURS [27] and

"quick" compiler back-ends for CM3). As a result CM3 relies on mostly-accurate GC.

3.2 The CM3 collector

The original mostly-copying SRC garbage collector has remained essentially the same through its evolution into the CM3 implementation. On many platforms it functions solely as a stop-the-world, non-generational collector. However, on some platforms, generational and incremental collection are supported by barriers using virtual memory protection primitives. This support is complicated by the need to inter-operate with native system calls, in which virtual memory protection traps cause an error return value, rather than delivery of a signal to the application that can be handled. This means that system calls cannot be passed references to *protected* pages in the heap. However, the presence of untidy derived references enable just such expression. To get around this problem, every port that uses virtual memory primitives to implement barriers for GC must also provide system call wrappers that *validate* any reference being passed to the system call by processing the reference to ensure it refers to neither older nor grey pages. Implementing these wrappers is a burden for portability of the incremental and generational collector.

A second complication for incremental and generational GC arises from preemptive thread scheduling. Since a thread can be preempted at *any* point in its execution, even inside wrappers to system calls, the collector must ensure that all heap references held in thread stacks remain valid for use across such calls. There are essentially two ways to achieve this. For user-level threading, it is straightforward to make system calls atomic by disabling thread switching on entry to their wrappers and re-enabling switching on exit. However, system-level threading requires an expensive lock, so instead it is better simply to *re-validate* stack-held references in the stop-the-world phase of the incremental collector, to make sure that no thread refers to an older or grey page when it resumes.

A more difficult problem arises for incremental GC that relies on virtual memory read barriers. When a mutator thread accesses a grey page, the resulting access violation triggers a signal handler that must do the GC work necessary to shade the page black before returning. In order to do that, the page must be *unprotected* so that its contents can be scanned. Yet the scanning process must execute atomically with respect to other threads, or else they may see inconsistent state. One way to do that is to stop the other threads while the page is scanned. Again, this is straightforward for user-level threading since turning off thread switching is a simple way to ensure atomicity of a thread's operations. However, system-level threading requires that the threads actually be stopped. This destroys any notion that GC is mostly-concurrent, and can have a disastrous performance effect on multi-processors, destroying parallel throughput.

An alternative approach is to map the same physical data from the grey virtual page into a different virtual page with access enabled, leaving the original virtual page protected. The collector thread can then scan the physical contents of the page to shade it black, before unprotecting the original virtual page. We considered this option, but saw the burden of supporting it on multiple target platforms as compromising portability. Moreover, it suffers from the forced granularity of the virtual memory page, as described above.

Finally, the semantics of signals in multi-threaded environments varies so much across platforms (on some systems, the trapping thread may not be the thread that receives the resulting signal) that we consider their use to be effectively, if not inherently, non-portable.

3.3 Mostly-concurrent, mostly-copying GC

To address these problems of heap synchronization we instead rely on software read and write barriers. These have the advantage of being totally portable to all CM3 targets, since they are injected by the CM3 compiler front end, and require no support for system call wrappers. They also mean that mutators explicitly synchronize if they access a grey page that is in the process of being incrementally scanned by another thread, to ensure atomicity of the scanning. This promotes scalability, since threads that are operating independently of that page can run freely. To promote scalability, we also rely on thread-local heap allocation in the fast case, so threads allocate from a private allocation page, only negotiating with the shared heap for a new allocation page when their current one is exhausted [26]. We now present these and other details of our collector. Modula-3 provides accurate information about references in global variables and within heap objects. Ambiguous and untidy references occur only within the thread stacks.

3.3.1 Modula-3 threads

We have ported the CM3 thread libraries to use system-level threading based on portable POSIX threads (a Win32-based thread implementation is also supported). Every Modula-3 thread maps directly to a native thread. These are scheduled preemptively and mapped to processors by the operating system.

3.3.2 Thread-local allocation

We use thread-local allocation to void synchronization among mutator threads on every allocation [26]. Each mutator thread allocates from its own private black allocation page, by incrementing an allocation pointer and comparing against the page limit to check that the page is not full. The allocation and limit pointers are held in thread-local storage, so their manipulation requires no synchronization. Thus, fast-path allocation is simply an increment and a comparison plus object initialization. This fast-path allocation sequence must be atomic with respect to the stop-the-world phase of collection, to prevent the collector from seeing a partially-allocated/initialized object that it cannot decode safely. Although allocation pages are zeroed wholesale, allocating certain objects in Modula-3 (e.g., open multi-dimensional arrays) requires careful initialization of the object meta-data that must be performed atomically for the object to be decodable by other threads. Thus, any thread executing in the fast path is considered to be *uninterruptible* and cannot be stopped until it exits that fast path. Busy threads are guaranteed eventually to transition to interruptible status either because they exit the allocation fast path or they exhaust their private allocation page and are forced to slow-path allocation. Slow-path allocation means requesting a new page from the heap. Thus, if the allocation page is full, the mutator calls out to routines that lock the heap meta-data while obtaining a free page in the heap or mapping a new page from the operating system. All GC work is also performed while the heap meta-data is locked.

3.3.3 Triggering GC

Every slow-path page allocation by a mutator thread also checks to see if the collector is falling behind the mutator. If a round of incremental collection is already in progress, then an increment of GC work is performed sufficient for collection work to catch up [6]. Heuristics for determining whether GC is falling behind allocation by mutators rely on a single tune-able parameter called the *GC ratio*. This parameter specifies how many grey pages should be scanned for each new page allocated by the mutators. A GC ratio of 1.0 implies that one grey page is scanned for every page allocated. When a round of incremental collection completes, new heuristic parameters are computed based on the GC ratio and the number of pages surviving that round of GC, to decide when to trigger the next

round of GC. All variants of our collector use the same heuristics to start a new round of GC and to trigger an increment of work by the mutators.

3.3.4 Stopping the world

The stop-the-world phase occurs only at the beginning of a new round of GC. Our new collector stops all mutator threads except the one triggering the new round. We call this mutator thread the master; the stopped threads are the slaves. Stopping native threads is more or less difficult depending on the platform. Some platforms, such as Darwin (Mac OS X) and Solaris provide explicit primitives to suspend and resume target threads and obtain their thread stack bounds. Others, such as the new POSIX threads library for Linux (NPTL) require a more complicated handshake to stop threads and acquire their thread stack bounds using signals and semaphores.

Interruptible threads (see above) can be stopped immediately, whether via primitives or signals. The master thread must explicitly wait for any uninterruptible threads to become interruptible. This wait is bounded, because uninterruptible threads either exit the fast path uninterruptible allocation code or because they are forced to slow path allocation when their private allocation page is exhausted. Because the master thread holds the heap meta-data lock, threads entering the slow path will try to acquire this lock and so can be stopped.

Having stopped all the other mutator threads, the master thread proceeds by processing all mutator thread stacks (including its own) for ambiguous roots. All referenced pages are promoted to grey, as described earlier. However, because of the need to preserve *validated* references on the stack that are parameters to system calls or derived untidy references, the master thread must also scan these grey pages and shade them black, because threads can use their stack references without traversing a barrier, as discussed below. Thus, the stop-the-world phase scans both the thread stacks and the pages directly referred to from those stacks to make them black. We now discuss how barriers enforce this invariant throughout incremental collection.

3.3.5 Barriers

In Pirinen's [38] terminology, not only do we preserve a black mutator invariant, but we also make sure that mutators refer only to black pages. The need for this invariant is because of the difficulty of placing explicit barriers on accesses via untidy derived references. In particular, an untidy reference can be created well before it is ever used. Moreover, preemption means the stop-the-world phase can occur at any time (other than in the uninterruptible fast path of the allocator), including after the point at which any barrier is executed but before the point at which the reference guarded by that barrier is used. Hence the need to preserve thread-referenced pages as black during the stop-the-world phase.

Write barriers We place write barriers for generational collection on every operation that explicitly stores a reference to a field of an object in the heap, and whenever any traced substructure (i.e., containing some reference field) of a heap object is passed as a `VAR` parameter (on the assumption that the callee will update the field, and while we have the tidy reference to the heap object). Every heap page has an associated *dirty* bit that records if that page might contain a reference to a younger object [42]. For generational collection older clean pages need never be scanned for references to young objects. The write barrier dirties any older page to which a reference is stored, in case that reference might be to a younger object. During the stop-the-world phase, older pages are promoted wholesale, and dirty older pages are treated as containing roots for the GC.

Note that our software write barriers inject minimal code at each store site, dirtying *any* page to which a pointer is stored (or might be stored, for `VAR` parameters that contain traced references). We could do more complicated filtering of writes to avoid dirtying pages when the reference is not from an older to a younger page but wanted to minimize the invasiveness of the barrier [29, 10, 11].

In addition to the page-level dirty bit we also maintain per-object dirty bits in the object header. Every dirty page is guaranteed to contain at least one dirty object. Conversely, no clean page contains a dirty object. When older dirty pages are scanned, the dirty bits in their objects are cleared. Managing separate object-level dirty bits allows us to separate the barrier into a fast path and a slow path. At each pointer store, the fast path simply checks the dirty bit of the object to which the store is being made. If the object is already dirty, then nothing further need be done. Otherwise, the slow path of the barrier calls out to a subroutine that locks the heap (i.e., page) meta-data and proceeds to dirty the object's page, and sets the object's dirty bit. Thus, where a thread is repeatedly storing to the same object (e.g., when iterating through an array), it need only execute the slow path of the barrier for the first store to the initially clean object. We recognize that locking the entire heap just to dirty a page seems overkill. The reason we must do so is that the dirty bits are held in page meta-data kept on the side apart from the pages themselves. This meta-data is resized as the heap itself grows, so it must be locked in case another thread is growing the heap. Moreover, we need to ensure that dirtying occurs atomically with respect to GC initiated by another thread, or else the dirtying might be lost.

Read barriers Read barriers for incremental collection are placed on every load of a reference from the heap or a global variable. If the page that is the target of the *loaded* reference is grey then the page is scanned to make it black. Thus the mutator can never obtain a reference to a grey page, and so never even see references to white pages. This approach is more *eager* [5] than simply making sure that the page from which the reference is being loaded is not grey (i.e., so that the loaded reference is not white), but permits us to avoid the complication of placing a read barrier on loads of references via untidy pointers (e.g., accessing a pointer field passed as a `VAR` or `READONLY` parameter).

Since the loaded reference is guaranteed to be tidy, we can again separate the barrier into fast and slow paths, using a *grey* bit in the header of the target object to decide if the slow path is needed. Here, the invariant is that all objects in a grey page have their grey bit set (the bit is set in the header of an object when it is relocated from a white page to a grey page). At each pointer load, the fast path simply checks the grey bit of the object to which the loaded pointer refers. If the object is not grey then nothing further need be done. Otherwise, the slow path of the barrier calls out to a subroutine that locks the heap (i.e., page) meta-data and proceeds to scan the target page, shading it black. Again, because the loaded reference is tidy, using the object-level grey bits avoids the slow path unless absolutely necessary.

A note on multi-processor coherence Since mutator threads are all stopped while the roots are processed, they are guaranteed to have synchronized with the collector at the start of a round of GC, at which point the dirty bits in older pages have all been cleared. Thus, they are guaranteed not to see a set dirty bit unless the page really has been dirtied. If weak memory ordering occurs such that a given mutator thread accidentally sees a cleared dirty bit when some other thread has already set it there is no harm done since the mutator will take the slow path and synchronize on the heap. Similarly, since grey bits are set only in newly copied objects (i.e., new grey pages) that have been written to since the stop-the-world phase, mutators can never see a cleared grey bit unless the corresponding page really has been turned from grey to black. If a mutator accidentally sees a set grey bit in a black page due to weak ordering there is

no harm done since the mutator will then take the slow path and synchronize.

3.3.6 Background GC

There is also an optional background mode for concurrent GC, which extends incremental mode (where mutators perform increments of collector work on slow path allocation as necessary) with a low-priority concurrent background GC thread to move collection ahead in the absence of mutator activity [12, 40, 23, 34, 7]. The background thread is tuned to cause insignificant interruption of mutator activities, but may therefore move the collection forward quite slowly.

4. Experiments

Our experiments explore the space of several of our implementation decisions. First, we wish to quantify the effect of switching from hardware barriers (i.e., using virtual memory protection primitives) to software barriers. We also explore how the addition of incremental collection impacts mutator pause times and mutator throughput compared with stop-the-world GC. Finally, we wish to demonstrate scalability of multi-mutator workloads on multi-processor platforms.

4.1 Platforms

Our experiments were performed on two platforms. The first is a Macintosh Xserve G5 running Mac OS X Server 10.4.4, with dual PowerPC 970 processors running at 2.3GHz, and 4 GB of RAM. Experiments using this platform were run while the system was very lightly loaded. The second platform is a Dell multi-processor running Linux kernel 2.6.14.4 (NPTL) with 8 Intel Pentium III processors running at 700 MHz, and 2 GB of RAM.

4.2 Benchmark

In the absence of useful real multi-mutator benchmarks, we use the synthetic GCOld benchmark devised by Printezis and Detlefs [40] which we have faithfully translated from Java into Modula-3. This benchmark measures the steady-state elapsed time for a mutator that performs allocation, some amount of object reference mutation, and some amount of non-allocation "work". The GCOld application allocates an array, each element of which points to the root of a binary tree about a megabyte in size. These data structures are allocated during an initialization phase, after which the program executes some number of *steps*, maintaining a steady-state heap size. Each step allocates some number of bytes of short-lived data that will die in a young-generation collection, and some number of bytes of long-lived subtrees that replace some previously existing subtrees, making them garbage. Each step further simulates some amount of mutator computation by several iterations of a busy-work loop. Finally, since the rate of pointer mutation and accesses are an important factor in the performance of both generational and mostly-concurrent collection, each step modifies some number of pointers (in a way that preserves the amount of reachable data). Command-line parameters control the amount of *live* data in the steady state, the number of units of mutator non-allocation work per byte allocated, the ratio of short-lived bytes allocated to long-lived bytes allocated, the number of pointer mutations per step, and the number of steps. In our experiments we use the following parameters: live storage of 8 megabytes; work varying as 1, 10, 100, 1000; 1:32 short-lived to long-lived allocation ratio; 2 pointer mutations per step; 100 steps. The fixed parameters were chosen not so much because they are truly representative in any way, but because they result in a moderately low survival rate, and reasonably low mutation rate, both of which are general characteristics of many applications. These parameters have also been fixed on by other authors [43].

We also modify the GCOld benchmark to capture a multi-mutator workload, by measuring the total elapsed time for some number of concurrent mutator threads each to perform the same amount of single-mutator work. Thus, while the threads allocate and manipulate independent tree structures, they compete for access to the heap to obtain private allocation pages, and cooperate by performing concurrent incremental collection as necessary on each slow-path allocation.

4.3 Results: hardware vs. software barriers

The first set of results reveal the relative costs of hardware and software barriers for the *user-level* threads implementation running on the XServe. In this configuration, STW collectors with hardware read barriers will never trigger a barrier, so the difference between the software and hardware STW collectors is solely due to the overhead of software barriers versus the overhead of manipulating virtual memory page protections at each STW GC. Heap locking overheads are negligible for user-level threading since locking is implemented as a simple increment/decrement of the variable that disables thread switching.

Figure 2 plots single-mutator elapsed time as the GC ratio varies for several work values. In these graphs, the hardware barrier results are presented as thicker grey lines, while the thin black lines represent software barrier results. The graphs also record 90% confidence intervals for 10 different runs for each configuration of the benchmark. From the graphs, one sees that stop-the-world generational collection using a software write barrier (STW) has best performance across all GC ratios and work values. However, the hardware-synchronized write barrier (STW-vm) suffers only slight performance degradation. At low non-allocation mutator work rates (1/10/100 in Figures2(a-c)) the hardware-synchronized incremental collector (INC-vm) is next best, whereas for work 1000 software-synchronized incremental collection (INC) is marginally better than INC-vm at low GC ratios, but approximates STW at higher GC ratios.

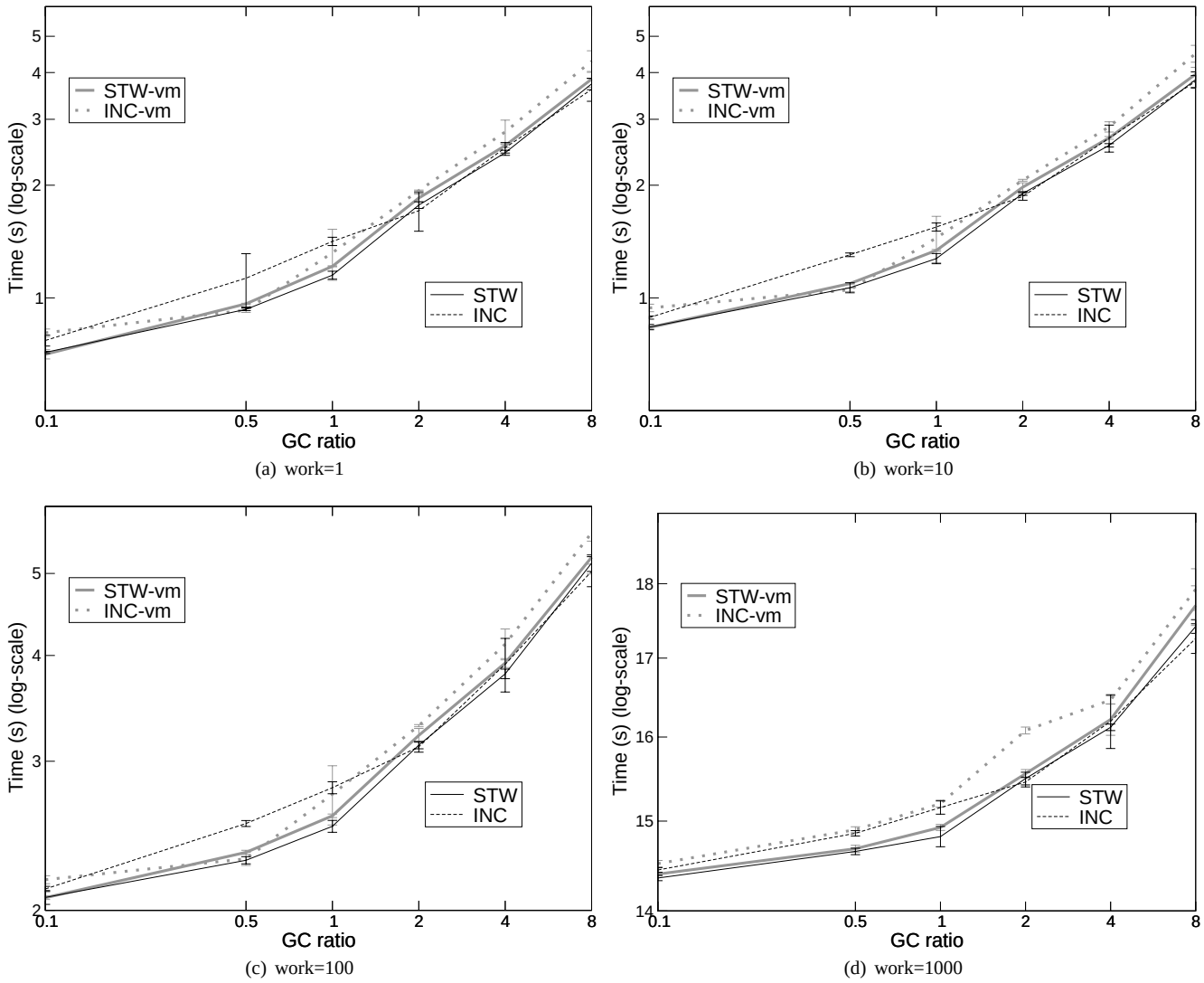
In any case, the use of software read barriers, which enable multi-processor concurrent collection, has noticeable, expected, but acceptable degradation of performance at GC ratio 0.5, but better or comparable performance at less passive (i.e., more realistic) GC ratios.

4.4 Results: elapsed time and maximum heap

The second set of results compare performance of stop-the-world (STW) and mostly-concurrent incremental collection (INC), as well as the impact of a low-priority concurrent collector thread (BG), when run on the dual-processor Xserve. Again, we report results with 90% confidence intervals for 10 runs of each configuration of the collector and benchmark.

The graphs of Figure 3 plot single-mutator elapsed time (left scale, thin black lines), as well as maximum heap size (right scale, thicker grey lines), as the GC ratio varies for several work values. Even as non-allocation work varies, maximum heap sizes remain essentially the same for each collector variant and GC ratio. This is no surprise, since non-allocation work has no impact on the heap.

Maximum heap sizes converge for both incremental variants (INC, BG) and stop-the-world STW at GC ratio 8.0, since that effectively forces the collector to use the smallest possible heap for the given allocation workload. The incremental variants exhibit essentially identical heap size footprints across the work range, and significantly higher than for stop-the-world collection. This is not unexpected, since incremental collection interleaves collector work with mutator allocation, so allowing the mutator to add more pages to the heap before the GC cycle completes and white pages can be freed. Naturally, elapsed time increases and heap size decreases as GC ratio increases.



(a) work=1 (b) work=10

(c) work=100 (d) work=1000

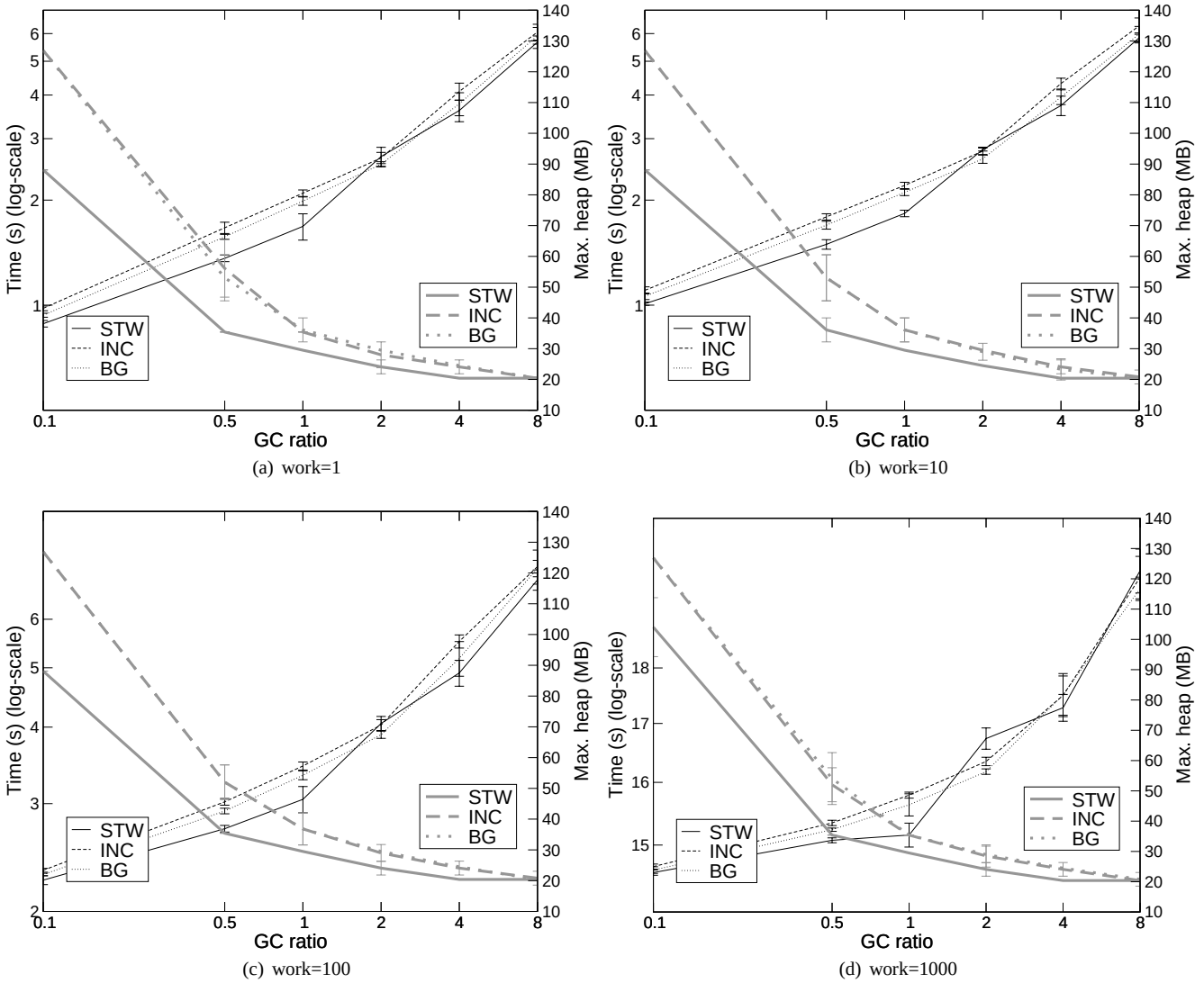
Figure 2. Hardware barriers versus software barriers

As expected, elapsed time is worse for the incremental variants except at high GC ratios, since they introduce synchronization overheads. Note, however, that for all work levels, having a background collector thread in addition to incremental collection by the mutator on slow path allocation (BG) results in slightly improved performance over pure incremental collection. This is because the background thread is able to advance collection even for mutators like GCOld that are voracious allocators.

4.5 Results: responsiveness and throughput

To evaluate responsiveness and throughput, we adopt the minimum mutator utilization (MMU) methodology of Cheng and Blelloch which measures the fraction of time in which the mutator does useful work in a given interval of time, but modified to obtain *bounded* mutator utilization (BMU). BMU plots a point (w, m) on a BMU curve if, for all intervals (windows) of length w or more that lie entirely within the program's execution, the mutator utilization is at least m. Thus, BMU curves are monotonically increasing, where the x-intercept is the maximum GC-related pause experienced by the program, and the asymptotic y-value is the overall throughput (fraction of execution time spent in the mutator). The BMU curve plots the maximum interval for which the mutator's fraction requirement is not satisfied. When comparing collectors, BMU curves that lie to the left (smaller maximum pause) and above (higher throughput) other curves represent more desirable behavior.

Figures 4 and 5 plot single-mutator BMU curves at several work levels for GC ratios of 0.5 and 1.0 respectively, both of which trade off reasonable GC overhead for space consumed (as illustrated earlier in Figure 3). These plots aggregate the mutator and collector activity from all 10 runs of the benchmark for each collector configuration. These results expose the allocation-intensiveness of the GCOld benchmark, at least for work values less than 1000. As revealed by these graphs, maximum pause times are reduced by 2 orders of magnitude from around 350 ms for STW to less than 10 ms for INC and BG. Mutator throughput for STW is always better than for the incremental variants, though throughput converges as non-allocation mutator work increases. Across all work values, the BMU curve for BG lies mostly outside of that for STW, once more showing that the low-priority background thread advances collection more quickly.



(a) work=1 (b) work=10

(c) work=100 (d) work=1000

Figure 3. Elapsed time (solid) and maximum heap size (dashed)

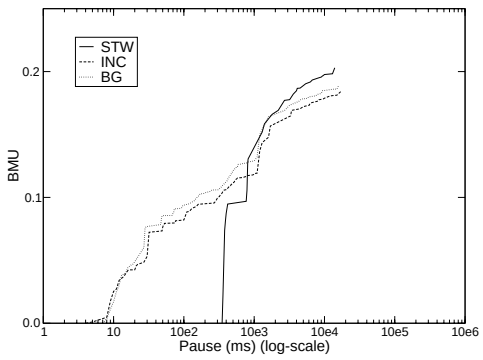
4.6 Results: multi-processor scalability

Our final set of results reveal scalability for multi-mutator runs on the 8-way Intel multi-processor, showing both elapsed time (lines) and heap size (points: circles=STW and crosses=INC) for STW and INC, as the number of concurrent mutator threads varies, and for both 0.5 and 1.0 GC ratios. Both elapsed time and maximum heap size are reported per thread, so as to provide a means of comparison as the number of threads grows. Per-thread throughput is essentially flat at work values of 1 and 10. Contention for heap meta-data and incremental GC is so high at these work rates that concurrency is severely curtailed. At work value 100, we begin to see improvement up to 8 threads, after which throughput declines (note that the log-scale masks the true improvement). However, for work of 1000, throughput scales nicely up to 8 threads before flattening off.

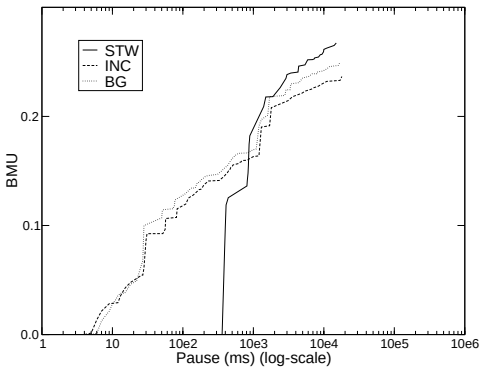
5. Related work

Incremental and concurrent garbage collection has been the subject of many prior works. Early concurrent mark-sweep (non-copying) collectors include those of Steele [44, 45] and Dijkstra et al.[19]. Baker's algorithm is the first incremental copying collector to use a read barrier to enforce an incremental invariant similar to ours [6]. Replicating collectors [37, 16, 31] use write barriers to record object mutations so that mutators can manipulate white objects even while they have been forwarded, with the mutation records used to update the copies. The "train" incremental algorithm [30] uses a write barrier to track references among different heap areas that are collected independently. Fully-concurrent (i.e., having no stop-the-world phase) on-the-fly mark-sweep collectors are unable to move objects [22, 21, 24, 23, 3]. Similarly for concurrent [18] and on-the-fly [4, 34] reference counting collectors, which are also unable to collect cycles.

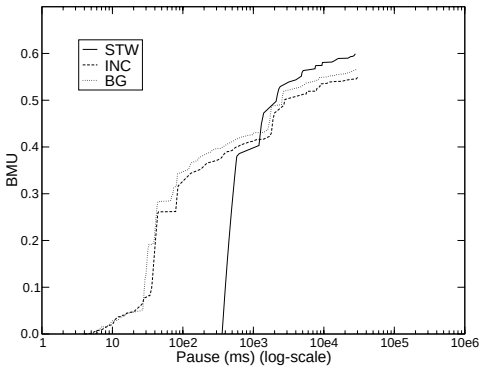
Several works focus on mostly-concurrent collection. Detlefs [17] described an early effort to implement a concurrent mostly-copying collector for C++ based on the techniques of Appel et al.[1] and Bartlett [8, 9], but his collector was neither generational nor portable, relying on Mach's virtual memory primitives for thread synchronization. Detlefs did not precisely characterize the impact on mutator utilization of his collector.



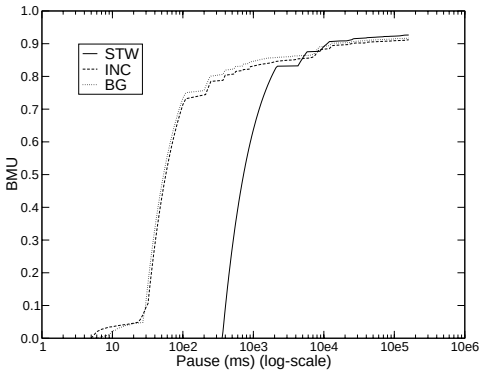
(a) work=1



(b) work=10

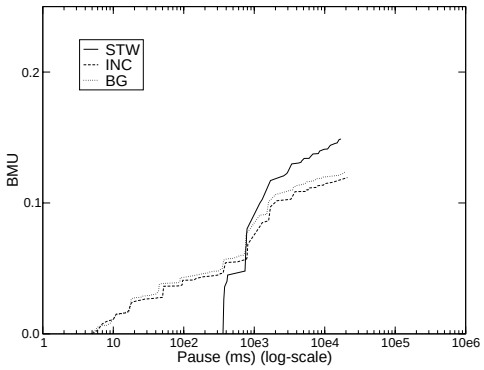


(c) work=100

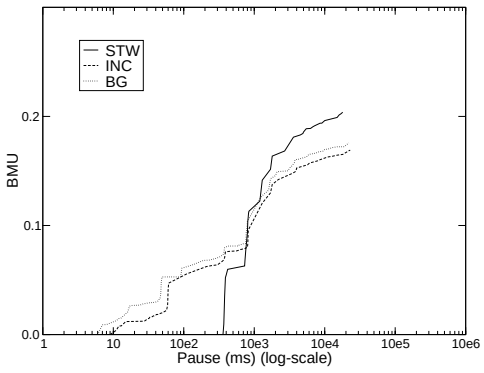


(d) work=1000

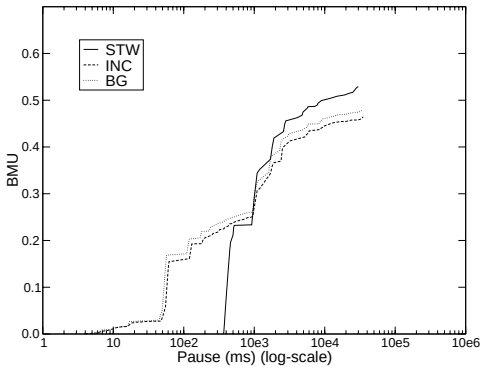
Figure 4. Aggregate BMU; GC ratio=0.5



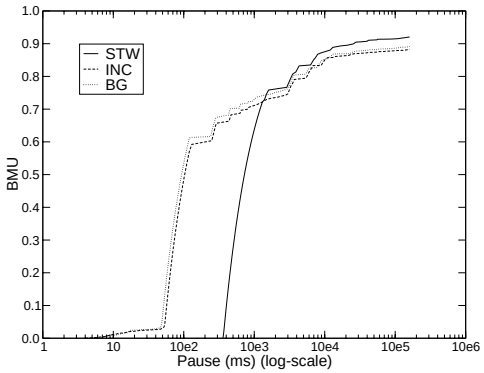
(a) work=1



(b) work=10

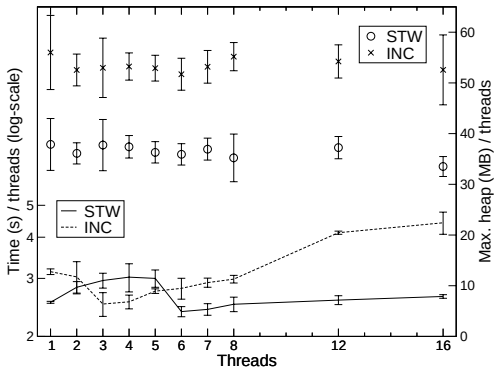


(c) work=100

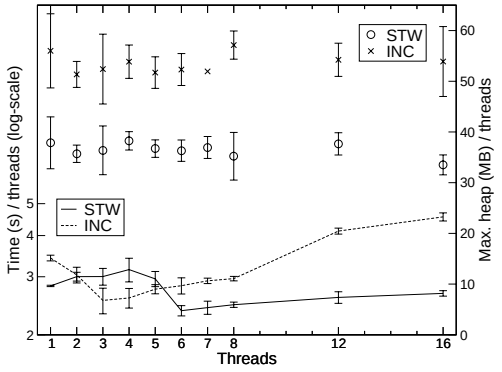


(d) work=1000

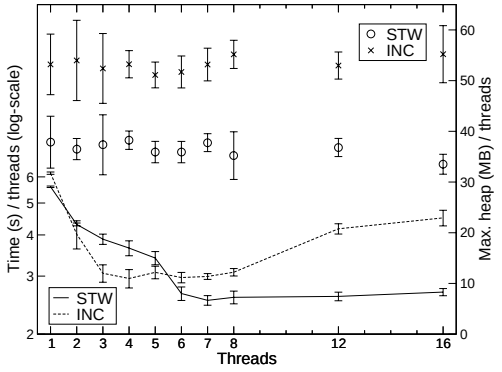
Figure 5. Aggregate BMU; GC ratio=1.0



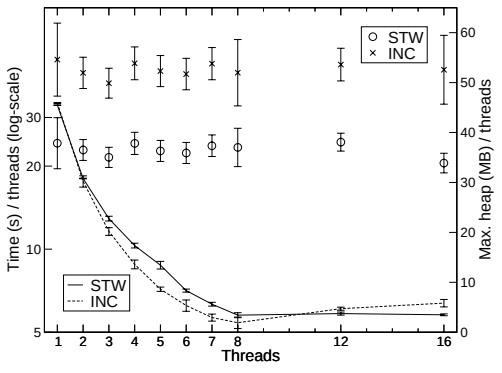
(a) work=1



(b) work=10

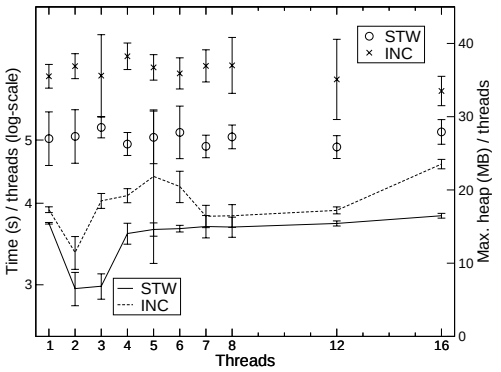


(c) work=100

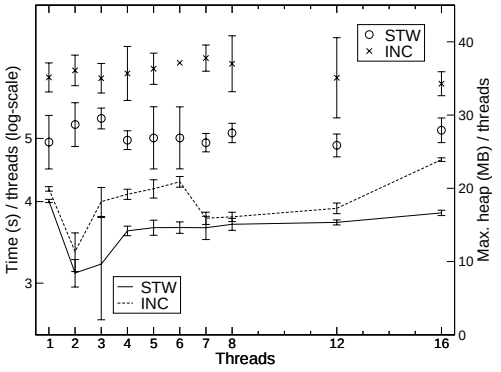


(d) work=1000

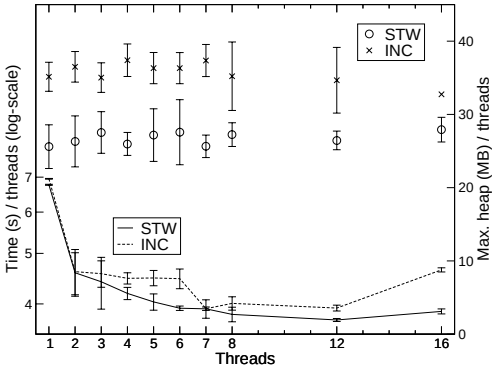
Figure 6. Scalability; GC ratio=0.5



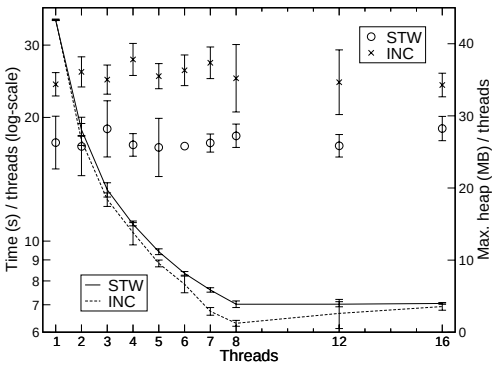
(a) work=1



(b) work=10



(c) work=100



(d) work=1000

Figure 7. Scalability; GC ratio=1.0

The mostly-concurrent conservative mark-sweep collector of Boehm et al.[12] uses a write barrier to keep the collector informed of mutator updates that create fresh grey pages needing to be scanned. Printezis and Detlefs [40] have extended this approach for generational collection. Barabash et al.[7] report experiences extending a similar parallel mark-sweep collector, focusing on techniques for parallelizing collector work. Mark-sweep collectors must rely on some separate, typically expensive, *compaction* phase to obtain better locality, minimize fragmentation, and improve heap utilization. In contrast, our collector is mostly-copying so no compaction phase is necessary.

6. Future work

There are several avenues of future work. First, our experiments have not explored the impact of page granularity on performance. Since we are now free of the constraint of virtual memory page granularity for heap pages, we can expect to benefit from the ability to adjust *heap* page granularities to match the application.

We are also interested in the impact of fast path barrier optimizations. We note that our fast path read and write barriers still need to check an object header bit on each access to a reference field. Testing this bit more than once is redundant, since the black to-space invariant for stack references preserves their validity. Any intervening GC between traversal of a given barrier and its guarded access will preserve the validity of the access. By communicating the invariant nature of barrier tests to the compiler we can expect such redundant checks to be removed by the gcc's existing redundancy elimination optimizations.

7. Conclusions

We have described the design and implementation of what we believe to be the first portable mostly-concurrent mostly-copying garbage collector, and characterized its performance for heap-intensive workloads. Our implementation is portable, relying only on existing system-level thread libraries including both POSIX threads and Win32. Our results report maximum pause times for the mostly-concurrent collector that are 1/100th the maximum pause time for fully stop-the-world collection. We also demonstrate good multi-processor scalability for applications that are not so allocation-intensive as to cause a bottleneck in slow-path allocation.

Acknowledgments

We thank the anonymous referees for their suggestions and improvements to this paper. This work is supported by the National Science Foundation under grants Nos. CCR-0085792, CNS-0509377, CCF-0540866, and CNS-0551658, and by IBM and Microsoft. Any opinions, findings and conclusions expressed herein are the authors and do not necessarily reflect those of the sponsors.

References

[1] APPEL, A. W., ELLIS, J. R., AND LI, K. Realtime concurrent collection on stock multiprocessors. In *Proceedings of the ACM Conference on Programming Language Design and Implementation* (Atlanta, Georgia, June). *ACM SIGPLAN Notices 23*, 7 (July 1988), pp. 11–20.

[2] 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.

[3] AZATCHI, H., LEVANONI, Y., PAZ, H., AND PETRANK, E. An on-the-fly mark and sweep garbage collector based on sliding views. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications* (Anaheim, California, Nov.). *ACM SIGPLAN Notices 38*, 11 (Nov. 2003), pp. 269–281.

[4] BACON, D. F., ATTANASIO, C. R., LEE, H., RAJAN, V. T., AND SMITH, S. Java without the coffee breaks: A nonintrusive multiprocessor garbage collector. In PLDI'01 [39], pp. 92–103.

[5] BACON, D. F., CHENG, P., AND RAJAN, V. T. A real-time garbage collector with low overhead and consistent utilization. In *Conference Record of the ACM Symposium on Principles of Programming Languages* (New Orleans, Lousiana, Jan.). *ACM SIGPLAN Notices 38*, 1 (Jan. 2003), pp. 285–298.

[6] BAKER, H. G. List processing in real time on a serial computer. *Commun. ACM 21*, 4 (Apr. 1978), 280–294.

[7] BARABASH, K., BEN-YITZHAK, O., GOFT, I., KOLODNER, E. K., LEIKEHMAN, V., OSSIA, Y., OWSHANKO, A., AND PETRANK, E. A parallel, incremental, mostly concurrent garbage collector for servers. *ACM Trans. Program. Lang. Syst. 27*, 6 (Nov. 2005), 1097–1146.

[8] BARTLETT, J. F. Compacting garbage collection with ambiguous roots. Research Report 88/2, Western Research Laboratory, Digital Equipment Corporation, Feb. 1988.

[9] BARTLETT, J. F. Mostly-copying garbage collection picks up generations and C++. Technical Note TN-12, Western Research Laboratory, Digital Equipment Corporation, Oct. 1989.

[10] BLACKBURN, S., AND MCKINLEY, K. S. In or out?: Putting write barriers in their place. In *Proceedings of the ACM International Symposium on Memory Management* (Berlin, Germany, Jun., 2002), D. Detlefs, Ed. *ACM SIGPLAN Notices 38*, 2 (Feb. 2003), pp. 281–290.

[11] BLACKBURN, S. M., AND HOSKING, A. L. Barriers: Friend or foe? In *Proceedings of the ACM International Symposium on Memory Management* (Vancouver, Canada, Oct., 2004), D. F. Bacon and A. Diwan, Eds. ACM, 2004, pp. 143–151.

[12] 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.

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

[14] CARDELLI, L., DONAHUE, J., GLASSMAN, L., JORDAN, M., KALSOW, B., AND NELSON, G. Modula-3 language definition. In *Systems Programming with Modula-3*, G. Nelson, Ed. Prentice Hall, 1991, ch. 2, pp. 11–66.

[15] CHENEY, C. J. A nonrecursive list compacting algorithm. *Commun. ACM 13*, 11 (Nov. 1970), 677–678.

[16] CHENG, P., AND BLELLOCH, G. A parallel, real-time garbage collector. In PLDI'01 [39], pp. 125–136.

[17] DETLEFS, D. L. Concurrent garbage collection in C++. Tech. Rep. CMU-CS-90-119, Carnegie Mellon University, Mar. 1990.

[18] DETREVILLE, J. Experience with concurrent garbage collectors for Modula-2+. Tech. Rep. 64, Systems Research Center, Digital Equipment Corporation, Palo Alto, CA, Aug. 1990.

[19] 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.

[20] DIWAN, A., MOSS, J. E. B., AND HUDSON, R. L. Compiler support for garbage collection in a statically typed language. In *Proceedings of the ACM Conference on Programming Language Design and Implementation* (San Francisco, California, June). *ACM SIGPLAN Notices 27*, 7 (July 1992), pp. 273–282.

[21] DOLIGEZ, D., AND GONTHIER, G. Portable, unobtrusive garbage

collection for multiprocessor systems. In *Conference Record of the ACM Symposium on Principles of Programming Languages* (Portland, Oregon, Jan.). 1994, pp. 70–83.

[22] DOLIGEZ, D., AND LEROY, X. A concurrent, generational garbage collector for a multithreaded implementation of ML. In *Conference Record of the ACM Symposium on Principles of Programming Languages* (Charleston, South Carolina, Jan.). 1993, pp. 113–123.

[23] DOMANI, T., KOLODNER, E. K., LEWIS, E., SALANT, E. E., BARABASH, K., LAHAN, I., LEVANONI, Y., PETRANK, E., AND YANOVER, I. Implementing an on-the-fly garbage collector for Java. In ISMM'00 [32], pp. 155–166.

[24] DOMANI, T., KOLODNER, E. K., AND PETRANK, E. A generational on-the-fly garbage collector for Java. In *Proceedings of the ACM Conference on Programming Language Design and Implementation* (Vancouver, Canada, June). *ACM SIGPLAN Notices 35*, 6 (June 2000), pp. 274–284.

[25] FENICHEL, R. R., AND YOCHELSON, J. C. A LISP garbage-collector for virtual-memory computer systems. *Commun. ACM 12*, 11 (Nov. 1969), 611–612.

[26] FLOOD, C. H., DETLEFS, D., SHAVIT, N., AND ZHANG, X. Parallel garbage collection for shared memory multiprocessors. In *Proceedings of the Java Virtual Machine Research and Technology Symposium* (Monterey, California, Apr.). USENIX, 2001.

[27] FRASER, C. W., HANSON, D. R., AND PROEBSTING, T. A. Engineering a simple, efficient code generator generator. *ACM Letters on Programming Languages and Systems 1*, 3 (Sept. 1992), 213–226.

[28] HOSKING, A. L., AND MOSS, J. E. B. Protection traps and alternatives for memory management of an object-oriented language. In *Proceedings of the ACM Symposium on Operating Systems Principles* (Asheville, North Carolina, Dec.). *ACM Operating Systems Review 27*, 5 (Dec. 1993), pp. 106–119.

[29] HOSKING, A. L., MOSS, J. E. B., AND STEFANOVIĆ, D. A comparative performance evaluation of write barrier implementations. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications* (Vancouver, Canada, Oct.). *ACM SIGPLAN Notices 27*, 10 (Oct. 1992), pp. 92–109.

[30] HUDSON, R. L., AND MOSS, J. E. B. Incremental collection of mature objects. In *Proceedings of the International Workshop on Memory Management* (Saint-Malo, France, Sept.), Y. Bekkers and J. Cohen, Eds. vol. 637 of *Lecture Notes in Computer Science*. Springer-Verlag, 1992, pp. 388–403.

[31] HUDSON, R. L., AND MOSS, J. E. B. Sapphire: copying garbage collection without stopping the world. *Concurrency and Computation—Practice and Experience 15*, 3 (March 2003), 223–261.

[32] *Proceedings of the ACM International Symposium on Memory Management* (Minneapolis, Minnesota, Oct., 2000). *ACM SIGPLAN Notices 36*, 1 (Jan. 2001).

[33] *Proceedings of the ACM International Symposium on Memory Management* (Vancouver, Canada, Oct., 1998). *ACM SIGPLAN Notices 34*, 3 (Mar. 1999).

[34] LEVANONI, Y., AND PETRANK, E. An on-the-fly reference counting garbage collector for Java. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications* (Tampa, Florida, Oct.). *ACM SIGPLAN Notices 36*, 11 (Nov. 2001), pp. 367–380.

[35] LIEBERMAN, H., AND HEWITT, C. A real-time garbage collector based on the lifetimes of objects. *Commun. ACM 26*, 6 (June 1983), 419–429.

[36] MOON, D. Garbage collection in a large Lisp system. In *Proceedings of the ACM Conference on Lisp and Functional Programming* (Austin, Texas, Aug.). 1984, pp. 235–246.

[37] NETTLES, S., AND O'TOOLE, J. Real-time replication 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. 217–226.

[38] PIRINEN, P. P. Barrier techniques for incremental tracing. In ISMM'98 [33], pp. 20–25.

[39] *Proceedings of the ACM Conference on Programming Language Design and Implementation* (Snowbird, Utah, June). *ACM SIGPLAN Notices 36*, 5 (May 2001).

[40] PRINTEZIS, T., AND DETLEFS, D. A generational mostly-concurrent garbage collector. In ISMM'00 [32], pp. 143–154.

[41] SMITH, F., AND MORRISETT, G. Comparing mostly-copying and mark-sweep conservative collection. In ISMM'98 [33], pp. 68–78.

[42] SOBALVARRO, P. G. A lifetime-based garbage collector for LISP systems on general-purpose computers, 1988. B.S. Thesis, Dept. of EECS, Massachusetts Institute of Technology, Cambridge.

[43] SPOONHOWER, D., BLELLOCH, G., AND HARPER, R. Using page residency to balance tradeoffs in tracing garbage collection. In *Proceedings of the ACM/USENIX Conference on Virtual Execution Environments* (Chicago, Illinois, June). ACM, 2005, pp. 57–67.

[44] STEELE, JR., G. L. Multiprocessing compactifying garbage collection. *Commun. ACM 18*, 9 (Sept. 1975), 495–508.

[45] STEELE, JR., G. L. Corrigendum: Multiprocessing compactifying garbage collection. *Commun. ACM 19*, 6 (June 1976), 354.

[46] 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.