Fast Multiprocessor Memory Allocation and Garbage Collection

Hans-J. Boehm Hewlett-Packard Laboratories
1501 Page Mill Rd.
Palo Alto, CA 94304
Hans_Boehm@hp.com

ABSTRACT

We extended our garbage collecting memory allocator[1] to provide good performance for multi-threaded applications on multiprocessors. The basic design is similar to the approach previously pursued in [12]. However, we concentrate on issues important to more common small-scale multiprocessors, and on specific issues not reported elsewhere. We argue that a reasonable level of garbage collector scalability can be achieved with relatively minor additions to the underlying collector code. Furthermore the scalable collector does not need to be appreciably slower on a uniprocessor.

Since our collector can serve as a plug-in replacement for malloc/free, we have the opportunity to compare it to scalable malloc-free implementations, notably Hoard [3]. Somewhat surprisingly, our collector significantly outperforms Hoard in some tests, a property that is mostly shared by the garbage collecting allocator in [ETY97]. We argue that garbage collectors currently require significantly less synchronization than explicit allocators, but that it may be possible to derive significantly faster explicit allocators from this observation.

Speedy access to thread-local storage is a significant issue in the design of allocators that must conform to standard calling conventions. We present empirical evidence that at least in the presence of a garbage collector, this can often be accomplished faster in a thread-independent way than through the standard thread library facilities, casting some doubt on the utility of the latter.

1. INTRODUCTION

Recently, there has been increasing interest in the performance of memory allocators and garbage collectors on multiprocessors, motivated by several factors:

- Authors of large scale numerical programs for multiprocessors are at least considering programming languages and styles that encourage, support, and sometimes require dynamic memory allocation.
- Large scale commercial Java applications are becoming more common, as is illustrated by the adoption of the SPEC JBB2000 benchmark[2]. These are often naturally multi-threaded, and can, at least in principal, benefit from multiprocessors. Although many Java virtual machines have provided for concurrent allocation for a number of years, the garbage collector itself is still often single-threaded, and can become the bottleneck.
- Small scale multiprocessors are becoming more economical, and far more common. It appears quite possible that in the future most desktop computers will contain multiple processors, perhaps manufactured on a single chip. Thus it becomes interesting to parallelize ordinary desktop applications, which may rely heavily on dynamic memory allocation.

We previously developed a garbage collector that can be used as a replacement for a `malloc/free` style explicit memory allocator. It is used in a number of research programming language implementations and a variety of systems written in C or C++, including some substantial commercial systems. It is also used in a few Java implementations, notably in the runtime for gcj, the runtime for the GNU static Java compiler[3], and it served as the basis for the Geodesic Systems Great Circle[4] garbage collector product.

The original design goals of this collector included:

- Thread-safety on as many platforms as possible.
- Allocator throughput should not degrade too much as more client threads and processors are added.

Note that although the second goal might appear to be trivial, it in fact is not. A number of standard allocator implementations fail to meet the goal [15]. In particular, we found that on many systems, the standard lock implementations would often result in excessive context switching, because they yielded prematurely on a multiprocessor, and possibly resulted in convoying (cf. [9, 4]). Older versions of our collector were reasonably successful at meeting the goal, but this required a custom lock implementation (using a flavor of adaptive locks [14]) on most platforms.

[1] See **http://www.hpl.hp.com/personal/Hans_Boehm/gc**

[2] See **http://www.specbench.org/osg/jbb2000/**

[3] See **http://sources.redhat.com/java**

[4] See **http://www.geodesic.com**

The original set of goals did not include increased throughput on multiple processors. The vast majority of the garbage collection and allocation code was protected by a single lock, thus ensuring that allocation and garbage collection would eventually become a bottleneck if we tried to scale applications to larger numbers of processors.

In this paper we thus address two additional goals:

- Allocation/garbage collection should not become the bottleneck as we scale applications to larger numbers of threads and processors. This requires that allocation and garbage collection throughput scale close to linearly with the number of processors.
- Even single-threaded applications should benefit somewhat from multiprocessors, in that the garbage collector itself should utilize all available processors.

Due to their easy availability, to us, as well as the rest of the world, we concentrated on the low end of the multiprocessor spectrum, i.e. bus-connected systems with 2 to 4 Intel processors. This meant that extreme scalability was less of a goal than in some other projects. However, this also implied that we could not afford to sacrifice uniprocessor performance for scalability; there were unlikely to be enough processors to make up for the initial performance loss.

We set ourselves the specific goal of making single threaded applications run as fast in a 4 processor thread-safe environment as in a single-processor thread-unsafe environment. The concurrency in the collector should make up for any synchronization in the allocator.

2. RELATED WORK

There have been several efforts to create processor-scalable implementations of malloc-free style allocators (c.f. [15, 3].) As we will see later, this involves a significantly different set of problems. Nonetheless, there is some hope that the measurements in this paper might inspire further improvement for allocators requiring explicit deallocation.

Traditionally, work on concurrency in garbage collection has concentrated on allowing the collector to run concurrently with the mutator(s) or client(s), rather than supporting multiple collector threads (cf. [8, 2, 6, 16, 11]). This allows visible garbage collection pauses to be reduced, though we now believe that explicit incremental collection during allocations is usually a better approach, since it avoids the scheduling issues in [6, 16]. In any case, this is largely an orthogonal issue, since mutator/collector concurrency does not prevent the collector from becoming a bottleneck with a large number of mutator threads, and collector/collector parallelism only mildly reduces pause times.

There has also been a small amount of work on allowing individual threads to collect local heaps concurrently (cf. [10, 17]). So far this has exhibited a limited benefit for overall scalability of the collector, though it probably has a locality benefit.

Endo Tauro, and Yonezawa previously parallelized an earlier version of our collector, and have made it publicly available.[5] Although we have made heavy use of their techniques, their work differs from ours in a number of ways:

- Their primary goal is performance on large (supercomputer scale) systems. As a result, their emphasis is what we would describe as extreme scalability. The base performance on small systems appears to have been less of an emphasis. In contrast, we wanted to make our library competitive enough on uniprocessors that we could use a single library for both uniprocessors and multiprocessors. As we will see below, this appears to be reflected in the final performance profiles of the two collectors.

- We wanted to integrate our work into our standard collector distribution, which must continue to be buildable for single-threaded platforms, and must continue to provide the original feature set for sequential collector builds, and as many of the original features as possible for parallel collector builds. We also wanted to avoid breaking ports of the sequential collector to platforms (e.g. Windows CE) which would be unlikely to benefit from parallel collection. All of these argue for approaches that minimally disturb the sequential collector code.

  In contrast, the University of Tokyo work provides two separate source code distributions, for shared and distribute memory multiprocessors respectively. Both diverge substantially from the sequential collector.

- We wanted to preserve the ability to link the collector against unmodified C programs as a malloc replacement. Our current parallel collector of our collector has been linked against programs designed for a traditional malloc/free implementation, though this facility tends to require platform specific tuning to deal with initialization-ordering issues.

  This forced us to pay more attention to the thread-specific-data issues discussed below. In the default configuration of the University of Tokyo collector those are finessed by dedicating a register to the allocator.

- Our measurements were performed in a different environment, and compare against different systems, leading to somewhat different conclusions and insights.

Some newer Java Virtual Machines also employ processor-scalable collectors. It is common to address at least allocator synchronization issues[9].

The parallel collector in the Jalapeno virtual machine appears to be one of the more ambitious efforts, but has only been somewhat superficially described in the literature [1]. Since the collector is confined to a Java Virtual Machine, and these normally dedicate a general register to a thread-context pointer (at least on a non-X86 architecture), we presume that the thread-local storage issues discussed below did not arise. Although it also uses a shared work list to perform parallel marking or copying, the more detailed characteristics of its implementation appear to be different. In particular, the work list data structure appears to be closer to that of [12], in large part because there was no issue of sharing code with a more sequential collector.

[5] Their collector, and some of the accompanying papers can be retrieved from **http://www.yl.is.s.u-tokyo.ac.jp/gc/**

3. CONTEXT

Our allocator/collector organizes the heap as a "big bag of pages": Each page[6] in the heap is dedicated to objects of a single size. Each page has an associated descriptor, which contains both mark bits for objects on that page, as well as descriptive information for the page as a whole, *e.g.* the size of individual objects.

Our collector is based on a Mark/Lazy Sweep algorithm.[5] Objects pointer-reachable from the roots (program variables) are marked. Unmarked objects are reclaimed incrementally during allocation calls. The collector may take advantage of compiler or programmer supplied type information to locate pointers, or it may operate in fully conservative mode, and treat everything as a potential pointer. The collector never moves objects.

The collector only supports a limited form of generational garbage collection, and even that is not used in our measurements. It is nonetheless performance competitive for many, but not all, applications. The techniques described here could also be used for the old generation in a collector with multiple generations.

The marker uses an explicit stack to store objects which are known to be reachable, but whose contents have not yet been examined (i.e. the "grey" objects in [8]) Each entry in the stack contains a base address and a mark descriptor, indicating the location of possible pointers relative to that starting address. Mark descriptors typically take the form of either a simple length specification, or a bit vector describing pointer locations.

Mark stack entries are used to describe both roots and heap objects.

The mark process is started by pushing starting addresses and mark descriptors for all root segments onto the mark stack. Nearly all of the marking time is then spent in a relatively small loop, which repeatedly removes an object from the top of the mark stack and, if it finds references to previously unmarked objects, marks those, and pushes them onto the stack.

At the end of the mark phase the page descriptors for all pages in the heap are scanned. Completely empty pages are recycled in their entirety without being touched by the collector. Nearly full pages are eliminated from further consideration. The remaining pages are enqueued by object size for later sweeping.

The allocator satisfies large object requests by going directly to a page level allocator. It maintains separate free-lists for various small object sizes. If the free-list for a requested small object size is empty, it is refilled first by sweeping an enqueued page containing objects of the right size or, if there are no more such pages, by obtaining a new page from the large object allocator, and dividing that into appropriately sized objects.

4. PARALLEL ALLOCATION

Allocators in many Java virtual machines provide per-thread allocation arenas, to avoid synchronization on every small object allocation. Since these collectors often compact memory, these arenas are typically contiguous regions of memory[9]. The size of these arenas is usually limited, since the arena dedicated to a thread will go unused if that thread allocates few or no small objects.

We use a similar scheme. However, since we cannot move objects, our scheme relies on thread-local free-lists instead of contiguous arenas.

We introduce a new thread-local allocation procedure. The old global free-list allocator remains available, and is used internally, as we see below.

Each thread has an associated array of 64 or 48 free-list headers. Each header corresponds to a different object size.[7] Requests for objects larger than that covered by these free-lists are handled through the old global free-list mechanism and require a lock acquisition for allocation. Since such allocations are more expensive anyway, in part because the object must be initialized, the locking cost is amortized over a larger amount of other work. Large object allocations also tend to be far less frequent.

We would like to avoid filling one of the thread-local free-lists, only to discover that that particular thread only allocates one object of the requested size. Thus each free-list header may contain either a (small) count of allocated memory for that size, *or* a pointer to a suitable free-list. It is initialized to a zero count.

Initially, allocations for each object size are satisfied from the corresponding global free-list, and the count is incremented. Once the count exceeds a threshold of about a page size, we start using a local free-list for that thread and that object size. Since we add at most a page at a time to a thread-local free-list, this ensures that we never reserve more memory for a thread than it has already allocated, and thus overallocate by at most a factor of two.

A thread-local free-list is refilled using the same mechanism that was already in place to refill the global free-lists.[8] It was relatively easy to no longer acquire the main allocator lock during free-list construction, thus allowing several threads to build free-lists concurrently. Mutual exclusion is still needed during large block allocation or to remove a page from the "waiting to be swept" queue. But these are relatively fast operations, and should not limit scalability for moderate numbers of processors.

5. PARALLEL MARKING

It is more difficult to parallelize the mark phase of the garbage collector. But even that turned out to require surprisingly little additional code.

At process startup, we arrange to create $N-1$ specialized marker threads, where N is the available number of processors. A garbage collection is initiated by an allocating client thread. The client thread then stops all other client threads, as it did in the original collector. It carries out the mark phase jointly with the $N-1$ marker threads, thus utilizing all available processors.

The same data structure that was originally used as the mark stack now serves as a global list of waiting mark tasks. Although we retain its old representation as an array together with a "stack" pointer, it in fact now serves as a queue. It is initialized with descriptors of the root set as before. However, objects are removed by atomically replacing

[6]By "page" we mean a heap section which is usually either 4K or 8K in length, not necessarily a physical page. We also sometimes refer to it as a "chunk" or "heap block".

[7]On 32-bit machines we allocate multiples of 8 bytes, so these free-lists cover requests up to 512 byte objects. On 64-bit machines the allocation granularity is 16 bytes, and we use 48 headers, so the limit is 768 bytes.

[8]We did have to clean up the code somewhat to enable this.

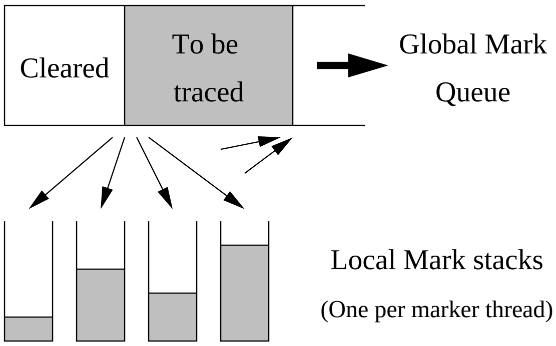

Figure 1: Data structure for grey objects

the mark descriptor with a zero descriptor indicating that no pointers remain to be followed. The stack pointer is never decremented until it is reset at the end of the collection.

Each of the marker threads repeatedly removes a small number of entries from the shared work queue (currently between 1 and 5 depending on the number of remaining entries), copies them to the bottom of a local mark stack, and proceeds to mark from there as in the sequential case, pushing newly found objects onto the *local* mark stack. When the local mark stack is emptied, more object descriptors are remove from the global work queue.

A picture of the overall data structure for 4 marker threads is given in figure 1. The shaded sections represent grey objects (i.e. objects yet to be traced). The thin arrows represent possible movement of object descriptors.

Note that since the marker threads operate primarily on their local mark stacks, the data structure is still traversed in a mostly depth-first fashion, which is probably more likely to resemble the allocation order, and thus exhibit better spatial locality.

The removal of an object from the global mark queue requires no synchronization whatsoever. The descriptor is simply overwritten. We assume that aligned word writes are atomic.[9]

There is no guarantee that an object will be traced by exactly one marker thread. But repeated tracing is unlikely to occur, unlikely to have more than a minor performance impact, and is guaranteed to .remain correcerve correctness.

To speed up the search for a work queue entry, we maintain a shared pointer to the first entry on the mark queue with a possibly nonzero descriptor. This is updated with an atomic compare-and-swap whenever a thread discovers a larger safe value. Thus it is guaranteed to increase monotonically during a collection.

Objects may be added back to the global work queue. This currently requires locking and is expected to be rare. In addition to the initial addition of the root set, this happens when:

1. A mark thread discovers that the global work queue is empty, but it still has multiple entries on its local mark stack. This condition is checked only relatively rarely (after about a page of tracing). It is necessary for load balancing, since a single thread may otherwise end up tracing most of the data structure.
2. A local mark stack is in danger of overflowing. This should be very rare, though it may happen with very long linked lists of certain kinds.

As in [12], we also found it necessary to split large objects before marking them, so that the tracing duties for such objects could be shared between threads.

Overflow of the work queue is handled by the same code that handled mark-stack overflow in the sequential case.

6. ISSUES AFFECTING ABSOLUTE PERFORMANCE

Since we focussed heavily on absolute performance, we encountered several issues that have apparently not been pointed out by prior work:

6.1 Mark bit representation

As we mentioned above, in our collector, each page has an associated array of mark bits. In the sequential collector, we reserved one bit per word.[10] Setting a mark bit is implemented, as it must be on most architectures, by reading a containing addressable unit (we use a word for historical reasons), *or*ing in the appropriate bit, and writing the result back.

In the parallel collector, adjacent mark bits may be set concurrently by multiple threads. This means that we must find a way to make the setting of mark bits appear atomic, where the naive implementation of the above scheme may lose bits written by another thread between the read and write operations.

We explored two ways of resolving this issue:

1. Update mark bits using an atomic compare-and-swap instruction. We compute the word containing the new mark bit as above. However, we write it back using an atomic compare-and-swap instruction to ensure that that word of the mark bit array was not concurrently modified.[11] If we discover that there was a concurrent modification, we retry the process, starting with the read operation.

   This has the advantage that the heap and the mark bits consume the same amount of memory as in the sequential case. It has the disadvantage that we have added overhead to the mark phase, since an atomic compare-and-swap typically is significantly more expensive than a simple store instruction.

2. We expand each mark bit to a byte. To partially compensate for the space overhead, we restrict object sizes to be a multiple of two words, and thus only allocate one mark byte for every two words in the heap. (Except in the case of one word objects, this is usually required for alignment reasons anyway.)

[9] As far as we know, all modern multiprocessors satisfy this constraint.

[10] It is unclear whether this is optimal. It allows the mark bit to be retrieved quickly from the object address, and allows single-word objects (*e.g.* for short character strings) to be allocated efficiently. But it adds more space overhead than necessary.

[11] On an architecture such as MIPS or Compaq Alpha, we would use a "load locked" instruction to read the mark word, and a "store conditional" instruction to write it back. The issues are otherwise the same.

This quadruples the space overhead for mark bits. On 32-bit machines, it becomes one eighth of the heap size. This is likely to affect the number of cache misses encountered by the marker threads. It also forces single-word objects to be allocated as double-word objects.

On the positive side, it reduces the number of instructions executed and memory operations below that of the sequential case. Setting a mark bit now requires just a byte store operation.

Unfortunately neither alternative is a clear winner in all cases. Mark bytes are infeasible on architectures without atomic byte stores, *e.g.* old Compaq Alpha machines. The mark-bit-based approach is infeasible on machines without something like an atomic compare-and-swap instruction. Even on machines that support both, we found that either could substantially outperform the other on different platforms. In particular, on the X86 machines we tried, the cache overhead of the mark bytes appeared to outweigh the cheaper instructions. However, on Itanium we decided to use the mark-byte-based implementation after similar experiments.

6.2 Thread-specific-data

Since we use thread-specific free-lists, we need a way to quickly generate a pointer to the thread-specific data structure containing the free-lists. In most (all?) Java virtual machines this is handled by maintaining a "thread context" pointer in a register at all times. The "thread context" would contain the free-list headers.

Unfortunately, this approach is not viable if we must obey standard calling conventions. Such conventions often do include a register reserved for a thread identifier, which may in fact point to a data structure describing the thread.[12] But the contents of this data structure are typically determined by the implementor of the thread library, opaque to the client, and not directly extensible.

Thread interfaces such as the Posix one[13] address this issue by providing a way to store and retrieve per-thread data. In the case of Posix threads, this is accomplished primarily through three functions:

`pthread_key_create` Creates a key value that can be used to refer to thread-local storage. (The approximate win32 equivalent is `TlsAlloc`.)

`pthread_setspecific` Sets the value associated with a given key and the current thread. (The approximate win32 equivalent is `TlsSetValue`.)

`pthread_getspecific` Retrieves the thread-local value associated to the given key and the current thread. (The approximate win32 equivalent is `TlsGetValue`.)

In most cases, including ours, only `pthread_getspecific` is performance critical, since the per-thread data is actually a pointer to a data structure containing the free-list headers, not the header itself.

[12] On register poor architectures such as the X86, it may be too expensive to dedicate a general register for this purpose, possibly even in a JVM. The alternatives are to dedicate a less useful register (*e.g.* a segment register on the X86) or to use the techniques we discuss below.

[13] See, for example, the description of the `pthread_` functions at **http://www.unix-systems.org/online.html**

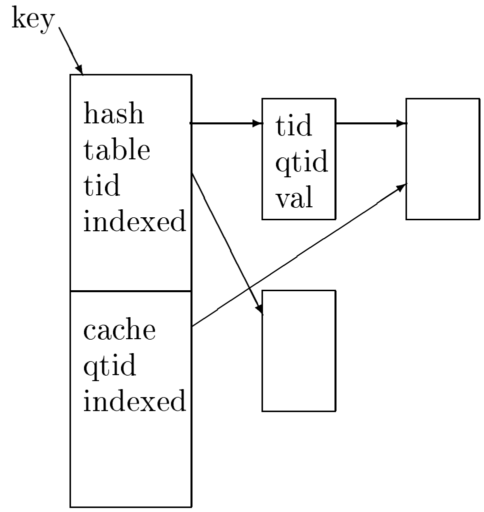

Figure 2: Thread specific value lookup

We started out using `pthread_getspecific` in this way. This requires that it be called once per allocation. Unfortunately, its performance turned out to be inadequate, in spite of individually reasonable design decisions in its implementation. We believe this is typical of many implementations of thread-local storage.

Typical implementations of `pthread_getspecific` use small integers as keys. `pthread_getspecific` typically involves a call to a dynamic library routine, with its associated overhead. It proceeds by first obtaining a pointer to the thread data structure maintained by the thread library. This data structure could contain an array of thread-specific values indexed by the key. In our environment (recent linuxthreads versions), it is actually a multilevel data structure, which is used to avoid small upper bound on the number of keys. The cost of `pthread_getspecific` is increased slightly more by the fact that it needs to error-check the key argument.

Fortunately, it is possible to implement nearly the same interface entirely in our own code with better performance.

Instead of using small integers as keys, the keys themselves will be pointers to data structures containing two kinds of data:

1. A chained hash table mapping thread ids to the associated thread-specific data. Insertions and deletions in this table require locking. We do however take care that the fields representing the thread id and the corresponding thread-local value in each chain entry always remains valid, even during hash table manipulations. Thus lookups never need to acquire a lock.

2. A second hash table used as a cache for faster lookups, as described below.

The data structure is pictured in figure 2. Note that in our case the *val* field represents a pointer to the thread-specific free-list headers. The fast path of the lookup procedure works as follows: We first quickly compute a value, the *quick thread id* that uniquely identifies our thread. Unlike a conventional thread id, we do not require that the quick thread id be unique for a thread, but only that two different threads never generate the same quick thread id. We could use the in-register thread-pointer, if available. However, for our experiments we use the top bits of the stack pointer, which we approximate by the address of a local variable.[14]

The quick thread id is used as an index into the cache array. Each entry in the cache array is either a pointer to a recognizably invalid hash chain entry, or points to a hash chain entry corresponding to a quick thread id with the appropriate hash value. To quickly check that we found the right entry, we store the quick thread id that was most recently used to access a particular entry in the entry itself. If we find a non-matching quick thread id, we revert to a lookup through the main hash table.

Thus a successful lookup through the cache requires something like 4 memory references and a test: We load the key value, load the cache entry, load the quick thread id in its target, check that it matches, and load and return the associated value. Further this code can easily be inlined in the allocator.[15]

The resulting execution times for one of our benchmarks is given in table 1. Note that thread-library-independent implementation is consistently faster than the thread-library-provided one.

Unfortunately, it appears to be nontrivial to carry this over to a non-garbage-collected environment. In our environment, hash chain entries are simply dropped on thread exit. If another reader thread happens to be accessing that entry, it will continue to be able to do so. When the last such thread finishes, the collector reclaims the entry. It appears nontrivial to explicitly deallocate hash chain entries without requiring some kind of synchronization for readers of the data structure. This is of course only an issue for applications that start an unbounded number of threads over their lifetime.

7. COLLECTOR MEASUREMENTS

We measured the performance of both the sequential and parallel collectors on a 4 processor Pentium Pro 200 machine running RedHat 7 Linux.[16] This machine has a single 66MHz system bus. As we will see, it is possible for the garbage collector to become memory bandwidth limited.[17]

7.1 Allocators

We compare the following allocators, though not all of them are included in all cases:

RH7 This is the standard glibc `malloc/free` implementation as distributed with the RedHat 7 Linux distribution. This is a somewhat scalable allocator derived from Doug Lea's malloc. We liked against the `pthreads` library to force locking.

RH7-single The above allocator, but with the application not linked against the pthreads library. This is of course not thread-safe, but avoids locking, and is therefore measurably faster in the single-threaded case than RH7.

Hoard The Hoard scalable memory allocator.[3]. Also requires explicit deallocation.

GC-extrapolated Our garbage collector run on one processor. The multiprocessor throughput numbers were computed by multiplying the uniprocessor number by the number of processors. This is included only for reference purposes, since it is not achievable. All measurements of our garbage collector were performed with a version similar to 6.0alpha5, and should be reproducible with 6.0alpha5.

GC-process Multiple copies of the single-threaded, but thread-safe benchmark are run concurrently in separate processes. Scalability here should be limited by memory and kernel issues, but not by garbage collector scalability, since the multiple garbage collector instances do not interact.

GC-thread Our parallel garbage collector run with multiple client threads. Except when stated otherwise, we set the number of marker threads to be equal to the number of client threads.

GC-seq Our collector with parallel collection and thread-local allocation disabled. This is similar to our original collector.

GC-single Our collector in thread-unsafe mode.

SGC The University of Tokyo scalable collector in its default configuration.

7.2 Benchmarks

We give throughput measurements for various benchmarks. In many cases, speedup graphs would hide much of the information, since we are as interested in the base uniprocessor performance as anything else, and that varies tremendously between allocators.

For each test, we used between 1 and 4 processors, with the number of processors appearing along the bottom axis. The number of marker threads, and (when applicable) the number of concurrent client threads, was set to the number of "in-use" processors.

Like most low-end multiprocessors, our machine appears to have memory bandwidth limitations which sometimes limit scalability even for programs which otherwise parallelize perfectly.

Note that the artificial benchmarks do very little other than allocate memory. Thus one would expect them to point out allocation contention issues on 4 processors that would only be exhibited on larger machines with real code.

We discuss each benchmark in turn:

[14]This requires that thread stacks have some spacing between them. This is normally true anyway, since unmapped pages are used to help detect thread stack overflows.

[15]This approach does require careful attention to the underlying memory model, e.g. to make sure that new hash-chain entries are fully initialized before they become visible to other threads.

[16]We unfortunately cannot yet release measurements measurements for our primary target platform. We hope be able to use a more modern machine and additional benchmarks for the final paper.

[17]The prefetching techniques of [5] were not used in the mark phase, since Pentium Pro processors do no implement the Pentium III prefetch instruction.

Number of processors	1	2	3	4
PThread	11.62	14.61	15.57	19.27
GC custom	10.68	12.97	14.99	19.11

Table 1: MT_GCBench2 execution times vs. thread-local storage impl.

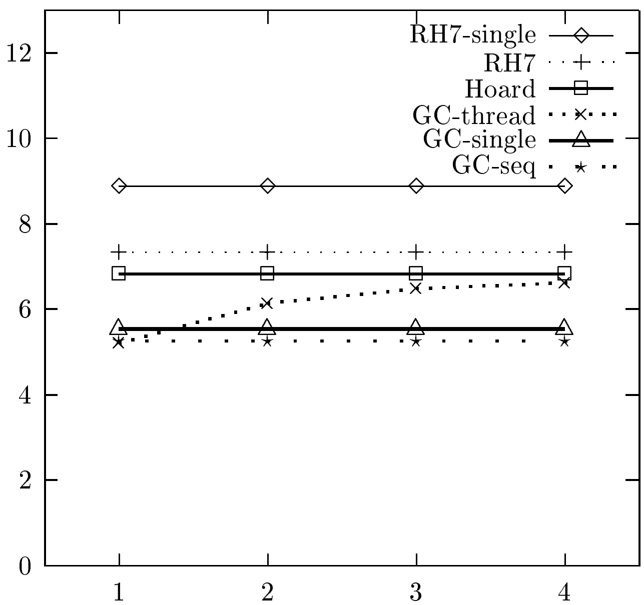

Figure 3: Throughput for Ghostscript Benchmark

7.2.1 Ghostscript

This is one of the Zorn allocation benchmarks[18]. Arguably, it is the most interesting member of the publicly available suite, in that the runtime on the largest input is still long enough to be accurately measurable, and the heap size significantly exceeds the cache size on most commodity machines. It is an older version of Ghostscript, built with the custom memory allocator disabled.

This is the only real program in our collection. Unfortunately, it is hard to it turn into a multi-threaded benchmark, and we did not attempt to do so. Thus only the parallel-garbage-collected case benefits from multiple processors, and only the garbage collector itself is running in multi-threaded mode. This benchmark has not been tailored for garbage collection. Like the two versions of the Larson benchmark below, the deallocation logic remains in the garbage-collected case, but no deallocation is actually performed.

We let the garbage collector use its heuristics for setting the heap size.

Throughput measurements, in benchmark iterations per minute, are given in figure 3.

As in in [7], this benchmark appears relatively unfavorable to garbage collection. It appears that with explicit deallocation, a significant amount of memory can be deallocated and reallocated without leaving the cache, something that is unlikely to happen with a garbage collector. Its average object size in our environment is 97 bytes, which is relatively large, and thus leads to more frequent collections. We present it here, since it exhibits significantly different performance characteristics from the artificial benchmarks that follow, though real programs that repeatedly build and destroy large data structures and allocate smaller objects may in fact behave more like the artificial benchmarks.

7.2.2 MT_GCBench2

MT_GCBench2 essentially runs multiple concurrent copies of GCBench, a commonly used, and sometimes criticized [13], garbage collector benchmark.[19]

For the garbage collected runs, the heap size is set to the number of client threads times 32 MB. The benchmark itself alternately builds and drops complete binary trees of various heights. In the explicit deallocation case, further recursive tree traversals are added to explicitly deallocate the trees.

Throughput measurements, expressed in benchmark iterations per minute, are given in figure 4. Note that, at least in this case, GC-thread performs nearly as well as GC-process, suggesting that our GC algorithm is highly parallel, at least when tracing complete binary trees.

The tree nodes allocated by this benchmark are 16 bytes in size, plus allocator-required overhead. This, as well as the relatively large heap size, are favorable to the garbage collecting allocators. Nonetheless, it is surprising that both SGC and GC-thread outperform the `malloc/free` implementations by such a wide margin.

7.2.3 Larson

This is a slightly modified version of a benchmark originally introduced by Larson and and Krishnan.[15] It was also used in [3] We obtained the benchmark from the Hoard web site, and modified it to call `memset` to fully initialize each newly allocated object. The original touches only the first cache line of newly allocated objects, thus producing unrealistically favorable results if the allocator also fails to touch most of the object.[20] Unlike the first benchmark, we run the garbage collector in its default mode as a malloc replacement, without explicit heap expansion.

This benchmark is a challenge to the malloc/free implementations in that it allocates memory in one thread and deallocates it in a different one. It is a bit of a challenge to our garbage collector in that it creates and destroys many threads, each of which initially allocate from the global heap.[21]

[18] These are available from **ftp://ftp.cs.colorado.edu/pub/misc/malloc-benchmarks**

[19] Both are available from **http: www.hpl.hp.com/personali/Hans_Boehm/gc /gc_bench.html**

[20] Since the garbage-collecting allocator already initializes objects, they are effectively written twice in the garbage collected case. But the second write is extremely likely to hit in the cache, this is a relatively minor cost.

[21] This points out that it is probably preferable to pass free-list headers on from completed threads to new ones, and only reclaim long-unused free-list headers occasionally. However,

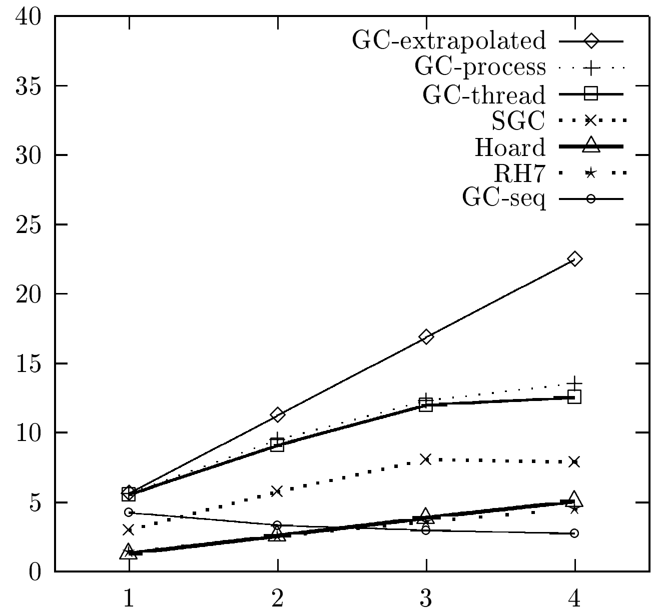

Figure 4: Throughput for MT_GCBench2 Benchmark

We ran the benchmark with the parameters suggested in the README file. This results in allocation of objects whose size is uniformly distributed between 10 and 500 bytes, i.e. probably larger than for most real applications.

Throughput measurements, expressed in allocations/second, are given in figure 5.

7.2.4 Larson-small

This is the same benchmark as above, but with parameters set to allocate objects between 10 and 50 bytes.

Throughput measurements, again expressed in allocations /second, are given in figure 6.

8. OBSERVATIONS ABOUT EXPLICIT DEALLOCATION

In the MT_GCBench2 and Larson-small benchmarks, the parallel garbage collectors significantly outperformed the parallel malloc implementations.[22] Both of these allocate primarily small objects, and gain only limited cache benefit from more immediate memory reuse.

In the single-threaded case, this also occurs occasionally, even for a conservative garbage collector, since it tends to be much cheaper to recycle large groups of objects than to process them individually. But it requires the average object size and heap occupancy to be sufficiently low that deallocation economy-of-scale outweighs the tracing cost. And this situation appears to be getting less common, due to the cache issues we observed in connection with the Ghostscript benchmark.

In the parallel case, garbage collectors appear to have some additional advantages:

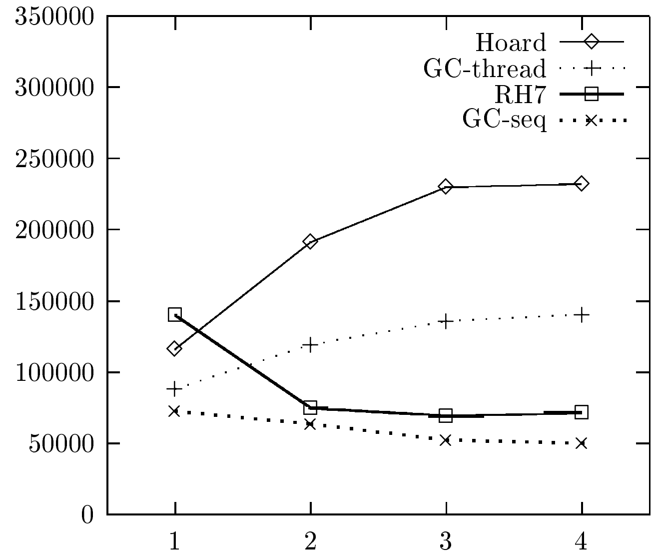

Figure 5: Throughput for Larson Benchmark

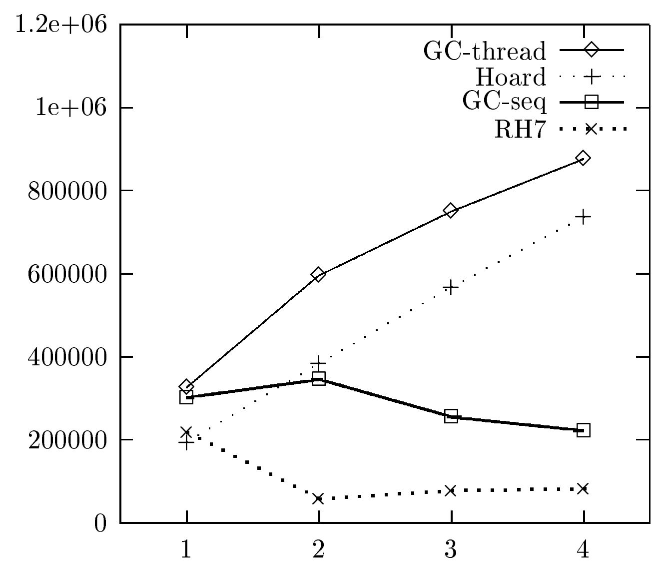

Figure 6: Throughput for Larson-small Benchmark

---

we haven't yet implemented that.

[22] We observed similar results with the threadtest benchmark used in [3], though we did not report those results here.

- No per-object lock acquisition for deallocation. Since objects are deallocated *en masse*, there is no need to acquire and release a lock for each object deallocated.
- No per-object lock acquisition for allocation. It is easy to allocate objects from memory which has previously been assigned to a thread-local free-list or arena. Thus we can also allocate with much less than one lock acquisition release cycle per object. It is relatively easy to obtain a group of approximately adjacent objects at once, since we allocate from free-lists which are naturally sorted. Other collectors are likely to allocate contiguous memory. In the explicit deallocation case, free-lists are less likely to be sorted, and the idea of operating on a group of objects is less natural.

The Hoard allocator, for example, still requires a lock acquisition and release for each of object allocation and deallocation.

This raises the question of whether a more GC-like strategy that operates mostly on larger batches of objects could also improve the performance of explicit deallocation, while preserving the performance advantages that explicit deallocation has in some cases. If objects were moved to thread-local control in groups, and then enqueued to be deallocated in groups, the lock acquisitions could be amortized over multiple objects. We do not know of an allocator that attempts this.

9. REFERENCES

[1] B. Alpern, C. R. Attanasio, J. J. Barton, M. G. Burke, P. Cheng, J.-D. Choi, A. Cocchi, S. J. Fink, D. Grove, M. Hind, S. F. Hummel, D. Lieber, V. Litvinov, M. F. Mergen, T. Ngo, J. R. Russell, V. Sarkar, M. J. Serrano, J. C. Shepherd, S. E. Smith, V. C. Sreedhar, H. Srinivasan, and J. Whaley. The Jalapeno virtual machine. *IBM Systems Journal*, 39(1), 2000.

[2] A. W. Appel, J. R. Ellis, and K. Li. Real–time concurrent collection on stock multiprocessors. In *SIGPLAN '88 Conference on Programming Language Design and Implementation*, pages 11–20, June 1988.

[3] E. Berger, K. S. McKinley, R. D. Blumofe, and P. R. Wilson. Hoard: A scalable memory allocator for multithreaded applications. In *Proceedings of the 2000 International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 117–128, November 2000.

[4] M. Blasgen, J. Gray, M. Mitoma, and T. Price. The convoy phenomenon. *Operating Systems Review*, 13(2):20–25, 1979.

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

[6] H.-J. Boehm, A. J. Demers, and S. Shenker. Mostly parallel garbage collection. In *SIGPLAN '91 Conference on Programming Language Design and Implementation*, pages 157–164, June 1991.

[7] D. Detlefs, A. Dosser, and B. Zorn. Memory allocation costs in large C and C++ programs. *Software Practice and Experience*, 24(6):527–547, 1994.

[8] E. W. Dijkstra, L. Lamport, A. J. Martin, C. S. Scholten, and E. Steffens. On-the-fly garbage collection: An exercise in cooperation. *Communications of the ACM*, 21(11):965–975, November 1978.

[9] R. Dimpsey, R. Arora, and K. Kuiper. Java server performance: A case study of building efficient, scalable Jvms. *IBM Systems Journal*, 39(1), 2000.

[10] D. Doligez and G. Gonthier. Portable unobtrusive garbage collection for multiprocessor systems. In *Conference Record of the Twenty-First Annual ACM Symposium on Principles of Programming Languages*, pages 113–123, 1994.

[11] T. Domani, E. K. Kolodner, and E. Petrank. A generational on-the-fly garbage collector for Java. In *SIGPLAN '00 Conference on Programming Language Design and Implementation*, pages 274–284, June 2000.

[12] T. Endo, K. Taura, and A. Yonezawa. A scalable mark-sweep garbage collector on large-scale shared memory machines. In *Proceedings of High Performance Networking and Computing (SC97)*, November 1997.

[13] T. L. Harris. Dynamic adaptive pre-tenuring. In *Proceedings of the International Symposium on Memory Management 2000*, pages 127–136, October 2000.

[14] A. Karlin, K. Li, M. Manasse, and S. Owicki. Empirical studies of competitive spinning for a shared-memory multiprocessor. In *Proceedings of the 1991 ACM Symposium on Operating System Principles*, 1991.

[15] P.-A. Larson and M. Krishnan. Memory allocation for long-running server applications. In *Proceedings of the International Symposium on Memory Management 1998*, pages 176–185, October 1998.

[16] T. Printezis and D. Detlefs. A generational mostly concurrent garbage collector. In *Proceedings of the International Symposium on Memory Management 2000*, pages 143–154, October 2000.

[17] B. Steensgard. Thread-specific heaps for multi-threaded programs. In *Proceedings of the 2000 International Symposium on Memory Management*, pages 18–24, 2000.