Parallel generational-copying garbage collection with a block-structured heap

Simon Marlow
Microsoft Research
simonmar@microsoft.com

Tim Harris
Microsoft Research
tharris@microsoft.com

Roshan P. James
Indiana University
rpjames@indiana.edu

Simon Peyton Jones
Microsoft Research
simonpj@microsoft.com

Abstract

We present a parallel generational-copying garbage collector implemented for the Glasgow Haskell Compiler. We use a block-structured memory allocator, which provides a natural granularity for dividing the work of GC between many threads, leading to a simple yet effective method for parallelising copying GC. The results are encouraging: we demonstrate wall-clock speedups of on average a factor of 2 in GC time on a commodity 4-core machine with no programmer intervention, compared to our best sequential GC.

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

General Terms Languages, Performance

1. Introduction

Garbage collection (GC) involves traversing the live data structures of a program, a process that looks parallel even in sequential programs. Like many apparently-parallel tasks, however, achieving wall-clock speedups on real workloads is trickier than it sounds.

In this paper we report on parallel GC applied to the Glasgow Haskell Compiler. This area is dense with related work, as we discuss in detail in Section 6, so virtually no individual feature of our design is new. But the devil is in the details: our paper describes a tested implementation of a full-scale parallel garbage collector, integrated with a sophisticated storage manager that supports generations, finalisers, weak pointers, stable pointers, and the like. We offer the following new contributions:

- We parallelise a copying collector based on a *block-structured heap* (Section 3). The use of a block-structured heap affords great flexibility and, in particular, makes it easy to distribute work in chunks of variable size.
- We extend this copying collector to a *generational* scheme (Section 4), an extension that the block-structured heap makes quite straightforward.
- We exploit the update-once property, enjoyed by thunks in a lazy language, to reduce garbage-collection costs by using a new policy called *eager promotion* (Section 4.1).
- We implement and measure our collector in the context of an industrial-strength runtime (the Glasgow Haskell Compiler), using non-toy benchmarks, including GHC itself (Section 5). We give bottom-line wall-clock numbers, of course, but we also try to give some insight into where the speedups come from by defining and measuring a notion of *work imbalance* (Section 5.3).

To allay any confusion, we are using the term block-structured here to mean that the heap is divided into fixed-size blocks, not in the sense of a block-structured programming language.

In general, our use of a block-structured heap improves on earlier work in terms of simplicity (fewer runtime data structures) and generality (an arbitrary number of independently re-sizable generations, with aging), and yet achieves good speedups on commodity multiprocessor hardware. We see speedups of between a factor of 1.5 and 3.2 on a 4-processor machine. Against this speedup must be counted a slow-down of 20-30% because of the extra locking required by parallel collection — but we also describe a promising approach for reducing this overhead (Section 7.1).

The bottom line is that on a dual-core machine our parallel GC reduces wall-clock garbage-collection time by 20% on average, while a quad-core can achieve more like 45%. These improvements are extremely worthwhile, *given that they come with no programmer intervention whatsoever*; and they can be regarded as lower bounds, because we can already see ways to improve them. Our collector is expected to be shipped as part of a forthcoming release of GHC.

2. The challenge we address

The challenge we tackle is that of *performing garbage collection in parallel* in a shared-memory multiprocessor; that is, employing many processors to perform garbage collection faster than one processor could do alone.

We focus on *parallel*, rather than *concurrent*, collection. In a concurrent collector the mutator and collector run at the same time, whereas we only consider garbage collecting in parallel while the mutator is paused. The mutator is free to use multiple processors

ISMM'08, June 7–8, 2008, Tucson, Arizona, USA.

too, of course, but even if the mutator is purely sequential, parallel garbage collection may still be able to reduce overall run-time.

2.1 Generational copying collection

Our collector is a generational, copying collector [Ung84], which we briefly summarise here to establish terminology. The heap is divided into *generations*, with younger generations having smaller numbers. Whenever generation n is collected, so are all younger generations. A *remembered set* for generation n keeps track of all pointers from generation n into younger ones. In our implementation, the remembered set lists all *objects* that contain pointers into a younger generation, rather than listing all *object fields* that do so. Not only does this avoid interior pointers, but it also requires less administration when a mutable object is repeatedly mutated.

To collect generations $0-n$, copy into *to-space* all heap objects in generations $0-n$ that are reachable from the *root pointers*, or from the remembered sets of generations older than n. More specifically:

- *Evacuate* each root pointer and remembered-set pointer. To evacuate a pointer, copy the object it points to into to-space, overwrite the original object (in from-space) with a *forwarding pointer* to the new copy of the object, and return the forwarding pointer. If the object has already been evacuated, and hence has been overwritten with a forwarding pointer, just return that pointer.
- *Scavenge* each object in to-space; that is, evacuate each pointer in the object, replacing the pointer with the address of the evacuated object. When all objects in to-space have been scavenged, garbage collection is complete.

Objects are *promoted* from a younger to an older generation, based on a *tenuring policy*. Most objects die young (the "weak generational hypothesis"), so it is desirable to avoid promoting very young objects so they have an opportunity to perish. A popular policy is therefore to promote an object from generation n to $n+1$ only when it has survived k_n garbage collections, for some k_n chosen independently for each generation. Rather than attach an age to every object, they are commonly partitioned by address, by subdividing each generation n into k_n *steps*. Then objects from step k_n of generation n are promoted to generation $n+1$, while objects from younger steps remain in generation n, but with an increased step count.

It seems obvious how to parallelise a copying collector: different processors can evacuate or scavenge different objects. All the interest is in the details. How do we distribute work among the processors, so that all are busy but the overheads are not too high? How do we balance load between processors? How do we ensure that two processors do not copy the same data? How can we avoid unnecessary cache conflicts? Garbage collection is a very memory-intensive activity, so memory-hierarchy effects are dominant.

Before we can discuss these choices, we first describe in more detail the architecture of our heap.

2.2 The block-structured heap

Most early copying garbage collectors partitioned memory into two large contiguous areas of equal size, from-space and to-space, together perhaps with an *allocation area* or *nursery* in which new objects are allocated. Dividing the address space in this way is more awkward for generational collection, because it is not clear how big each generation should be — and indeed these sizes may change dynamically. Worse still, the steps of each generation further subdivide the space, so that we have $n*k$ spaces, each of unpredictable size. Matters become even more complicated if there are multiple mutator or garbage collector threads, because then we need multiple allocation areas and to-spaces respectively.

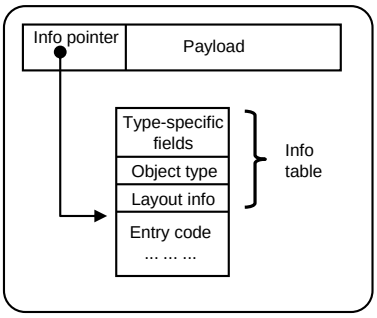

Figure 1. A heap object

Thus motivated, GHC's storage manager uses a *block-structured heap*. Although this is not a new idea [DEB94, Ste77], the literature is surprisingly thin, so we pause to describe how it works.

- The heap is divided into fixed-size B-byte *blocks*. Their exact size B is not important, except that it must be a power of 2. GHC uses 4kbytes blocks by default, but this is just a compile-time constant and is easily changed.
- Each block has an associated *block descriptor*, which describes the generation and step of the block, among other things. Any address within a block can be mapped to its block descriptor with a handful of instructions - we discuss the details of this mapping in Section 2.3.
- Blocks can be linked together, through a field in their descriptors, to form an *area*. For example, after garbage collection the mutator is provided with an allocation area of free blocks in which to allocate fresh objects.
- The heap contains *heap objects*, whose layout is shown in Figure 1. From the point of view of this paper, the important point is that the first word of every heap object, its *info pointer*, points to its statically-allocated *info table*, which in turn contains layout information that guides the garbage collector.
- A *heap pointer* always addresses the first word of a heap object; we do not support interior pointers.
- A large object, whose size is greater than a block, is allocated in a *block group* of contiguous blocks.

Free heap blocks are managed by a simple *block allocator*, which allows its clients to allocate and free block groups. It does simple coalescing of free blocks, in constant time, to reduce fragmentation. If the block allocator runs out of memory, it acquires more from the operating system.

Dividing heap memory into blocks carries a modest cost in terms of implementation complexity, but it has numerous benefits. We consider it to be one of the best architectural decisions in the GHC runtime:

- Individual regions of memory (generations, steps, allocation areas), can be re-sized at will, and at run time. There is no need for the blocks of a region to be contiguous; indeed usually they are not.
- Large objects need not be copied; each step of each generation contains a linked list of large objects that belong to that step, and moving an object from one step to another involves removing it from one list and linking it onto another.
- There are places where it is inconvenient or even impossible to perform (accurate) garbage collection, such as deep inside a runtime system C procedure (e.g. the GMP arbitrary-precision arithmetic library). With contiguous regions we would have to

estimate how much memory is required in the worst case before making the call, but without the requirement for contiguity we can just allocate more memory on demand, and call the garbage collector at a more convenient time.
- Memory can be recycled more quickly: as soon as a block has been freed it can be re-used in any other context. The benefit of doing so is that the contents of the block might still be in the cache.

Some of these benefits could be realised without a block-structured heap if the operating system gave us more control over the underlying physical-to-virtual page mapping, but such techniques are non-portable if they are available at all. The block-structured heap idea in contrast requires only a way to allocate memory, which enables GHC's runtime system to run on any platform.

2.3 Block descriptors

Each block descriptor contains the following fields:

- A link field, used for linking blocks together into an area.
- A pointer to the first un-allocated (free) word in the block.
- A pointer to the first pending object, used only during garbage collection (see Section 3.4).
- The generation and step of the block. Note that all objects in a block therefore reside in the same generation and step.
- If this is a block group, its size in blocks.

Where should block descriptors live? Our current design is to allocate memory from the operating system in units of an M-byte megablock, aligned on an M-byte boundary. A megablock consists of just under M/B contiguous block descriptors followed by the same number of contiguous blocks. Each descriptor is a power-of-2 in size. Hence, to get from a block address to its descriptor, we round down (mask low-order bits) to get the start of the megablock, and add the block number (mask high-order bits) suitably shifted.

An alternative design would be to have variable-sized blocks, each a multiple of B bytes, with the block descriptor stored in the first few bytes of the block. In order to make it easy to transform a heap pointer to its block descriptor, we would impose the invariant that a heap object must have its first word allocated in the first B bytes of its block.

3. Parallel Copying GC

We are now ready to describe our parallel collector. We should stress that while we have implemented and extensively tested the algorithms described here, we have not formally proven their correctness.

We focus exclusively on non-generational two-space copying in this section, leaving generational collection until Section 4.

We will suppose that GC is carried out by a set of *GC threads*, typically one for each physical processor. If a heap object has been evacuated but not yet scavenged we will describe it as a *pending object*, and the set of (pointers to) pending objects as the *pending set*. The pending set thus contains to-space objects only.

The most obvious scheme for parallelising a copying collector is as follows. Maintain a single pending set, shared between all threads. Each GC thread performs the following loop:

```
while (pending set non-empty) {
  remove an object p from the pending set
  scavenge(p)
  add any newly-evacuated objects to the pending set
}
```

3.1 Claiming an object for evacuation

When a GC thread evacuates an object, it must answer the question "has this object already been evacuated, and been overwritten with a forwarding pointer?". We must avoid the race condition in which two GC threads both evacuate the same object at the same time. So, like every other parallel copying collector known to us (e.g. [FDSZ01, ABCS01]), we use an atomic CAS instruction to claim an object when evacuating it.

The complete strategy is as follows: read the header, if the object is already forwarded then return the forwarding pointer. Otherwise claim the object by atomically writing a special value into the header, spinning if the header indicates that the object is already claimed. Having claimed the object, if the object is now a forwarding pointer, unlock it and return the pointer. Otherwise, copy it into to-space, and unlock it again by writing the forwarding pointer into the header.

There are variations on this scheme that we considered, namely:

- Claim the object before testing whether it is a forwarding pointer. This avoids having to re-do the test later, at the expense of unnecessarily locking forwarding pointers.
- Copy the object first, and then write the forwarding pointer with a single CAS instruction. This avoids the need to spin, but means that we have to de-allocate the space we just allocated (or just waste a little space) in the event of a collision.

We believe the first option would be pessimal, as it does unnecessary locking. The second option may be an improvement, but probably not a measurable one, since as we show later the frequency of collisions at the object level is extremely low.

This fine-grain per-object locking constitutes the largest single overhead on our collector (measurements in Section 5.1), which is frustrating because it is usually unnecessary since sharing is relatively rare. We discuss some promising ideas for reducing this overhead in Section 7.1.

3.2 Privatising to-space

It is obviously sensible for each GC thread to allocate in a private to-space block, so that no inter-thread synchronisation need take place on allocation. When a GC thread fills up a to-space block, it gets a new one from a shared pool of free blocks.

One might worry about the fragmentation arising from having one partly-filled block for each GC thread at the end of garbage collection. However, the number of blocks in question is small compared with the total heap size and, in any case, the space in these blocks is available for use as to-space in subsequent GCs (we discuss fragmentation further in Section 5.6).

3.3 The Pending Block Set

The challenge is to represent the pending set efficiently, because operations on the pending set are in the inner loop of the collector. Cheney's original insight [Che70] was to allocate objects contiguously in to-space, so that *the pending set is represented by to-space itself*; more precisely, the pending set is the set of objects between the to-space *allocation pointer* and the *scavenge pointer*. As each object is copied into to-space, the to-space allocation pointer is incremented; as each object is scavenged the scavenge pointer is incremented. When the two coincide, the pending set is empty, and the algorithm terminates.

Cheney's neat trick sets the efficiency bar. We cannot rely on parallel performance making up for a significant constant-factor slowdown: the parallelism available depends on the data structures and the heap, and some heaps may feature linear lists without any source of GC parallelism.

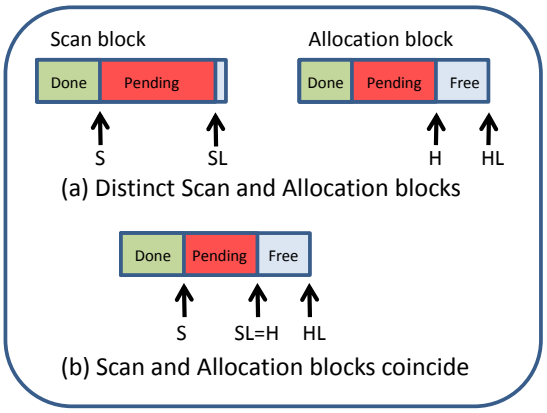

Figure 2. The state of a GC thread

Representing the pending set by a separate data structure incurs an immediate overhead. We measured the low bound on such overhead to be in the region of 7-8% - this figure was obtained by writing the address of every evacuated object into a small circular buffer, which is essentially the least that any algorithm would have to do in order to store the pending set in a data structure. Note that Flood et al [FDSZ01] use a work-stealing queue structure to store the pending set, so their algorithm presumably incurs this overhead.

If we do not store the pending set in a separate data structure, and use to-space as the pending set, then the challenge becomes how to divide up the pending set between threads in a parallel GC. Imai and Tick [IT93] divide to-space into blocks, and use the set of such blocks as the pool from which threads acquire work. In our setting this scheme seems particularly attractive, because our to-space is already divided into blocks! However, dividing the work into chunks may incur some loss of parallelism if we pick the chunk size too high, as our measurements show (Section 5.4).

We call the set of blocks remaining to be scavenged the Pending Block Set. In our implementation, it is represented by a single list of blocks linked together via their block descriptors, protected by a mutex. One could use per-thread work-stealing queues to hold the set of pending blocks [ABP98], but so far (using only a handful of processors) we have found that there is negligible contention when using a conventional lock to protect a single data structure; we give measurements in Section 5.5.

3.4 How scavenging works

During scavenging, a GC thread therefore maintains the following thread-local state (see Figure 2):

- A *Scan Block*, which is being scavenged (aka scanned).
- A *scan pointer*, `S`, which points to the next pending object in the Scan Block. Earlier objects in the Scan Block have already been scavenged (labelled "Done" in Figure 2).
- A *scan limit*, `SL`, which points one byte beyond the end of the last pending object in the Scan Block.
- An *Allocation Block*, into which objects are copied when they are evacuated from from-space.
- A *to-space allocation pointer* `H`, pointing to the first free word in the Allocation Block.
- A *to-space limit pointer* `HL`, which points one beyond the last word available for allocation in the Allocation Block.
- The Scan Block and the Allocation Block may be distinct, *but they may also be the very same block*. This can happen when there are very few pending objects, so the scan pointer `S` is close behind the allocation pointer `H`; this can happen, for example, when scavenging a linked list of heap objects.

Scan and Allocation blocks	`H` reaches `HL` (Allocation Block full)	`S` reaches `SL` (Scan Block empty)
(a) distinct	(1) Export the full Allocation Block to the Pending Block Set; get a fresh Allocation Block from the free list; initialise `H`, `HL`; stay in (a).	(2) Initialise `S`, `SL` to point to the Allocation Block; move to (b).
(b) coincide	(3) Get new Allocation Block from the free list; initialise `H`, `HL`; move to (a).	(4) Get new Scan Block from Pending Block Set; initialise `S`, `SL`; move to (a).

Figure 3. State transactions in scavenging

Figure 2 illustrates the two situations: (a) when the Scan Block and Allocation Block are distinct, and (b) when the two coincide. In case (a), when `S` reaches `SL`, the Scan Block has been fully scanned. When the two blocks coincide (case (b)), the allocation pointer `H` is the scan limit; that is, we assume that `SL` is always equal to `H`, although in the actual implementation we do not move two pointers, of course.

As the scavenging algorithm proceeds there are two interesting events that can take place: `H` reaches `HL`, meaning that the allocation block is full; and `S` reaches `SL` (or `H` in case (b)), meaning that there are no more pending objects in the Scan Block. The table in Figure 3 shows what happens for these two events.

Work is exported to the Pending Block Set in transition (1) when the Allocation Block is full, and is imported in transition (4) when the GC thread has no pending objects of its own. Notice that in transition (4) the Allocation Block remains unchanged, and hence *the Allocation Block may now contain fully-scavenged objects, even though it is distinct from the Scan Block.* That is why there is a "Done" region in the Allocation Block of Figure 2(a). The block descriptor contains a field (used only during garbage collection) that points just past these fully-scavenged objects, and this field is used to initialise `S` when the block later becomes the Scan Block in transitions (2) or (4).

3.5 Initiation and termination

GC starts by evacuating the root pointers, which in the case of a sequential Haskell program will often consist of the main thread's stack only. Traversal of the stack cannot be parallelised, so in this case GC only becomes parallel when the first GC thread exports work to the Pending Block Set. However, when running a *parallel* Haskell program [HMP05], or even just a Concurrent Haskell program, there will be more roots, and GC can proceed in parallel from the outset.

The scavenging algorithm of the previous sub-section is then executed by each GC thread until the Pending Block Set is empty. That does not, of course, mean that garbage collection has terminated, because other GC threads might be just about to export work into the Pending Block Set. However the algorithm to detect global termination is fairly simple. Here is a sketch of the outer loop of a GC thread:

```
gc_thread()
{ loop:
   scan_all();
      // Returns when Pending Block Set is empty

   running_threads--; // atomically
   while (running_threads != 0) {
     if (any_pending_blocks()) {
```

```
      // Found non-empty Pending Block Set
      running_threads++; // atomically
      goto loop;
    }
  } // only exits when all threads are idle
}
```

Here, `running_threads` is initialised to the number of GC threads. The procedure `scan_all` repeatedly looks for work in the Pending Block Set, and returns only when the set is empty. Then `gc_thread` decrements the number of running threads, and waits for `running_threads` to reach zero. (There is no need to hold a mutex around this test, because once `running_threads` reaches zero, it never changes again.) While other threads are active, the current thread polls the Pending Block Set to look for work. It is easy to reason that when `running_threads` reaches zero, all blocks are scanned. This termination algorithm was originally proposed by Flood et. al. [FDSZ01], although it differs slightly from the one used in their implementation.

3.6 When work is scarce

The block-at-a-time load-balancing scheme works fine when there is plenty of work. But when work is scarce, it can over-sequentialise the collector. For example, if we start by evacuating all the roots into a single block, then work will only spread beyond one GC thread when the scavenge pointer and the to-space allocation pointer get separated by more than a block, so that the GC thread makes transition (1) (Figure 3). And this may never happen! Suppose that the live data consists of two linear lists, whose root cells are both in a block that is being scavenged by GC thread A. Then there will always be exactly two pending objects between `S` and `H`, so thread A will do all the work, even though there is clearly enough work for two processors. We have found this to be an important problem in practice, as our measurements show (Section 5.4).

The solution is inescapable: when work is scarce, we must export partly-full blocks into the Pending Block Set. We do this when (a) the size of the Pending Block Set is below some threshold, (b) the Allocation Block has a reasonable quantum, Q, of un-scanned words (so that there is enough work to be worth exporting), and (c) the Scan Block also has at least Q un-scanned words (so that the current GC thread has some work to do before it goes to get more work from the Pending Block Set). We set the parameter Q by experimental tuning, although it would also be possible to change it dynamically.

This strategy leads to a potential fragmentation problem. When a pending block is fully scanned, it normally plays no further role in that garbage collection cycle. But we do not want to lose the space in a partly-full, but now fully-scanned block! The solution is easy. We maintain a Partly Free List of such partly-full, but fully-scanned blocks. When the GC wants a fresh Allocation Block, instead of going straight to the block allocator, it first looks in the Partly Free List. To reduce synchronisation we maintain a private Partly Free List for each GC thread.

3.7 Experiences with an early prototype

Ideas that work well in theory or simulation may not work well in real life, as our first parallel collector illustrated perfectly. With one processor it took K instructions to complete garbage collection. With two processors, each processor executed roughly $K/2$ instructions, but the elapsed time was unchanged! This was not simply the fixed overhead of atomic instructions, because we arranged that our one-processor baseline executed those instructions too.

The problem, which was extremely hard to find, turned out to be that we were updating the block descriptors unnecessarily heavily, rather than caching their fields in thread-local storage. Two adjacent block descriptors often share a common cache line, so two processors modifying adjacent block descriptors would cause stalls as the system tried to resolve the conflict.

It is hard to draw any general lessons from this experience, except to say that no parallel algorithm should be trusted until it demonstrates wall-clock speedups against the best sequential algorithm running on the same hardware.

4. Parallel generational copying

We have so far concentrated on a single-generation collector. Happily, it turns out that our parallel copying scheme needs little modification to be adapted to a multi-generational copying collector, including support for multiple steps in each generation. The changes are these:

- Each GC thread maintains one Allocation Block for each step of each generation (there is still just a single Scan block).
- When evacuating an object, the GC must decide into which generation and step to copy the object, a choice we discuss in Section 4.1.
- The GC must implement a write-barrier to track old-to-new pointers, and a remembered set for each generation. Currently we use a single, shared remembered set for each generation protected by a mutex. It would be quite possible instead to have a thread-local remembered set for each generation if contention for these mutexes became a problem.
- When looking for work, a GC thread always seeks to scan the oldest-possible block (we discuss this choice in Section 4.2).

That's all there is to it!

4.1 Eager promotion

Suppose that an object `W` in generation 0 is pointed to by an *immutable* object in generation 2. Then there is no point in moving `W` slowly through the steps of generation 0, and thence into generation 1, and finally into generation 2. *Object* `W` *cannot die until generation 2 is collected*, so we may as well promote it, and everything reachable from it, into generation 2 immediately, regardless of its current location. This is the idea we call *eager promotion*.

One may wonder how often we have an object that is both (a) immutable and (b) points into a younger generation. After all, immutable objects such as list cells are allocated with pointers that are, by definition, older than the list cell itself. Only mutable objects can point to younger objects. But for objects that are repeatedly mutated, eager promotion may not be a good idea because it may promote a data structure that then becomes unreachable when the old-generation cell is mutated again.

For a lazy functional language, however, eager promotion is precisely right for *thunks*. A thunk is a suspended computation that is updated, exactly once, when the thunk is demanded. The update means that there may be an old-to-new pointer, while the semantics of the language means that the thunk will not be updated again.

As a result, when evacuating an object `W` into to-space, that has not already been evacuated, we choose its destination as follows:

- If the object that points to `W` is immutable, evacuate `W` into the same generation and step as that object.
- Otherwise, move the object to the next step of the current generation, or if it is already in the last step, to the first step of the next generation.

Notice the phrase "that has not already been evacuated". There may be many pointers to `W`, and if we encounter the old-to-new pointer late in the game, `W` may have already been copied into to-space and replaced with a forwarding pointer. Then we cannot copy it again, because other to-space objects might by now be pointing to

the copy in to-space, so the old-to-new pointer remains. In general, when completing the scavenging of an object, the GC records the object in its generation's remembered set if any of the pointers in the object point into a younger generation.

We quantify the benefit of doing eager promotion in Section 5.7.

4.2 Eager promotion in parallel collection

Eager promotion makes it advantageous to scavenge older generations first, so that eager promotion will promote data into the oldest possible generation. What are the implications for a parallel generational system?

First, we maintain a Pending Block Set for each generation, so that in transition (4) of Figure 3 we can pick the oldest Pending Block.

Transition (2) offers two possible alternatives (remembering that each GC thread maintains an Allocation Block for each step of each generation):

1. Pick the oldest Allocation Block that has work. If none of them have pending objects, pick the the oldest Pending Set block.
2. Pick the oldest block we can find that has work, whether be it from a Pending Block Set or one of our own Allocation Blocks.

The first policy attempts to maximise parallelism, by not taking blocks from the shared pending set if there is local work available. The second attempts to maximise eager promotion, by always picking the oldest objects to scan first.

Note that eager promotion introduces some non-determinism into the parallel GC. Since it now matters in which *order* we scavenge objects, and the order may depend on arbitrary scheduling of GC threads, the total amount of work done by the GC may vary from run to run. In practice we have found this effect to be small on the benchmarks we have tried: the number of bytes copied in total varies by up to 2%.

5. Measurements

We chose a selection of programs taken from the nofib [Par92] and nobench [Ste] Haskell benchmarking suites, taking those (disappointingly few) programs that spent a significant amount of time in the garbage collector. The programs we measured, with a count of the number of source code lines in each, are:

- *GHC*, the Haskell compiler itself (190,000 lines)
- *circsim*, a circuit simulator (700 lines)
- *constraints*, a constraint solver (300 lines)
- *fibheaps*, a fibonacci heap benchmark (300 lines)
- *fulsom*, a solid modeller (1,400 lines)
- *gc_bench*, an artificial GC benchmark[1] (300 lines)
- *happy*, a Yacc-style parser generator for Haskell (5,500 lines)
- *lcss*, Hirschberg's LCSS algorithm (60 lines)
- *power*, a program for calculating power series (140 lines)
- *spellcheck*, checks words against a dictionary (10 lines)

One thing to note is that the benchmarks we use are all pre-existing *single-threaded* programs. We expect to see better results from multithreaded or parallel programs, because such programs will typically have a wider root set: when there are multiple thread stacks to treat as roots, we can start multiple GC threads to scan them in parallel, whereas for a single-threaded programs the root set usually consists of the main thread's stack only, so it can be longer until there is enough work to supply to the other GC threads.

[1] Translated from the Java `gc_bench` by Hans Boehm, who credits John Ellis and Pete Kovac of Post Communications as the original creators. This benchmark uses a lot of mutation, which is atypical for a Haskell program.

Program	Δ Time (%)
circsim	+29.5
constraints	+29.4
fibheaps	+26.3
fulsom	+19.3
gc_bench	+34.6
happy	+36.9
lcss	+21.7
power	+25.9
spellcheck	+28.9
Min	+19.3
Max	+36.9
Geometric Mean	+27.9

Figure 4. Increase in GC time due to atomic evacuation

By default, GHC uses two generations, two steps in the youngest generation, a fixed-size nursery of 0.5MB, and increases the size of the old generation as necessary. This configuration is designed to be cache-friendly and memory-frugal, but we found it to be suboptimal for parallel GC: the nursery is too small to make it worthwhile starting up multiple threads for the young-generation collections (but see Section 7.2). So instead of using a fixed-size nursery and a variable sized old-generation, we gave each program a fixed-size total heap. In this configuration the GC allocates all unused memory to the nursery, and hence will collect the nursery less frequently.

We set the size of the heap given each program to be the same as the maximum heap size attained when the program was run in the default (variable-sized heap) configuration. Typically this value is about 3 times the maximum amount of live data encountered in any GC.

It is worth noting that strange artifacts abound when measuring GC. If a change to the collector causes a GC to take place at a different time, this can affect the cost of GC dramatically. The volume of live data can change significantly over a short period of time, for example when a large data structure suddenly becomes unreachable. To minimize these effects in our measurements, we aim to always collect at the same time, by measuring the amount of live data and scheduling collections accurately.

We made all our measurements on a machine with dual quad-core Intel Xeon processors, for a total of 8 cores. The OS was Windows Server x64, but we compiled and ran 32-bit binaries.

5.1 Locking overhead

Figure 4 compares our best sequential GC with the parallel GC executing on a single processor, and therefore shows the fixed overhead of doing GC in parallel. (Unfortunately we did not collect results for the GHC benchmark, but have no reason to believe they would differ dramatically from the others shown). The majority of this overhead comes from the per-object locking that is necessary to prevent two processors from evacuating the same object (Section 3.1).

The overhead is about 30%, which means the parallel GC must achieve at least a speedup of 1.3 just to beat the single-threaded GC.

5.2 Speedup

Figure 5 shows the speedup obtained by our parallel GC, that is, the ratio of the wall-clock time spent in GC when using a single CPU to the wall-clock time spent in GC when using N CPUs, for values of N between 1 and 8.

The baseline we're using here is the parallel version of the GC with a single thread; that is, the per-object locking measured in the previous section is included.

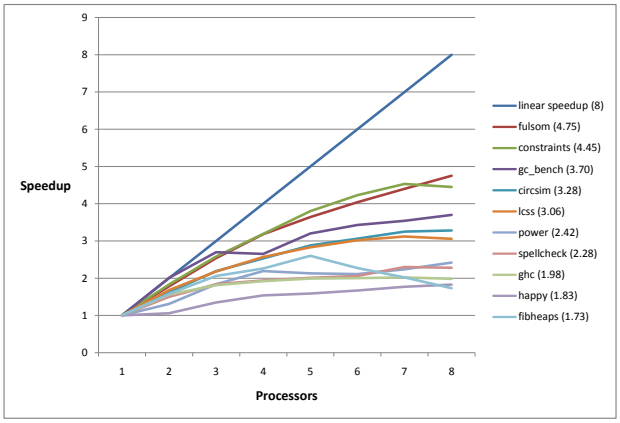

Figure 5. Speedup on various benchmarks

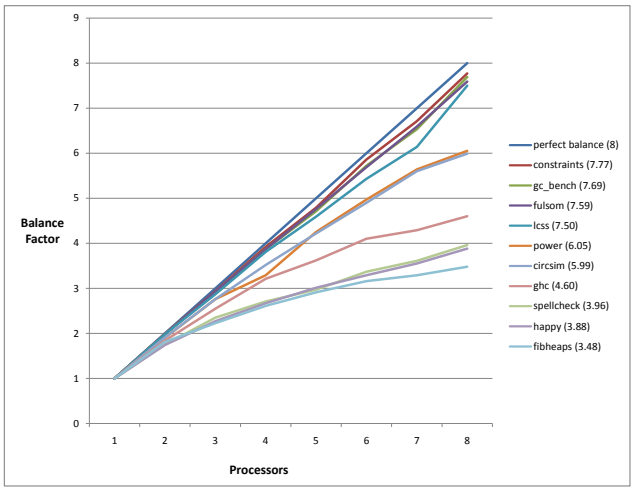

Figure 6. Work balance

The amount of speedup varies significantly between programs. We aim to qualify the reasons for these differences in some of the measurements made in the following sections.

5.3 Measuring work-imbalance

If the work is divided unevenly between threads, then it is not possible to achieve the maximum speedup. Measuring work imbalance is therefore useful, because it gives us some insight into whether a less-than-perfect speedup is due to an uneven work distribution or to other factors. The converse doesn't hold: if we have perfect work distribution it doesn't necessarily imply perfect wall-clock speedup. For instance, the threads might be running in sequence, even though they are each doing the same amount of work.

In this section we quantify the work imbalance, and measure it for our benchmarks. An approximation to the amount of work done by a GC thread is the *number of bytes it copies*. This is an approximation because there are operations that involve copying no bytes: scanning a pointer to an already-evacuated object, for example. Still, we believe it is a reasonable approximation.

We define the *work balance factor* for a single GC to be C_{tot}/C_{max} where C_{tot} is the total number of bytes copied by all threads, and C_{max} is the maximum number of bytes copied by any one thread. Perfect balance is N (the number of GC threads), perfect imbalance is 1. This figure is the maximum real speedup we could expect, given the degree to which the work was distributed across the threads.

Generalising this to multiple GCs is done by treating all the GCs together as a single large GC - that is, we take the sum of C_{tot} for all GCs and divide by the sum of C_{max} for all GCs.

Figure 6 shows the measured work balance factor for our benchmarks, in the same format as Figure 5 for comparison. The results are broadly consistent with the speedup results from Figure 5: the top 3 and the bottom 4 programs are the same in both graphs. Work imbalance is clearly an issue affecting the wall-clock speedup, although it is not the only issue, since even when we get near-perfect work balance (e.g. constraints, 7.7 on 8 processors), the speedup we see is still only 4.5.

Work imbalance may be caused by two things:

- Failure of our algorithm to balance the available work
- Lack of actual parallelism in the heap: for example when the heap consists mainly of a single linked list.

When the work is balanced evenly, a lack of speedup could be caused by contention for shared resources, or by threads being idle (that is, the work is balanced but still serialised).

Gaining more insight into these results is planned for future work. The results we have presented are our current best effort, after identifying and fixing various instances of these problems (see for instance Section 3.7), but there is certainly more that can be done.

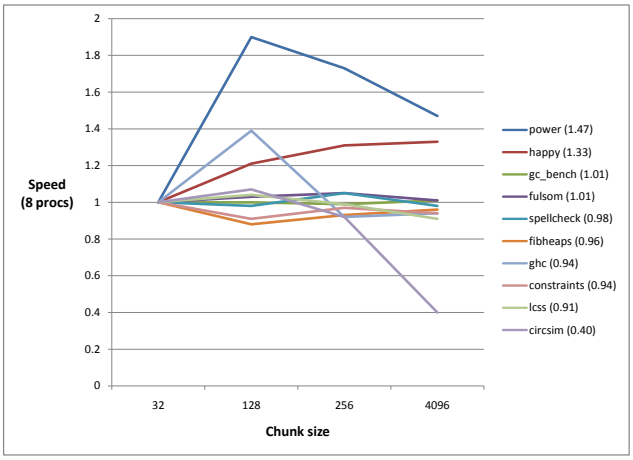

Figure 7. Varying the chunk size

5.4 Varying the chunk size

The collector has a "minimum chunk size", which is the smallest amount of work it will push to the global Pending Block Set. When the Pending Block Set has plenty of work on it, we allow the allocation blocks to grow larger than the minimum size, in order to reduce the overhead of modifying the Pending Block Set too often.

Our default minimum chunk size, used to get the results presented so far, was 128 words (with a block size of 1024 words). The results for our benchmarks on 8 processors using different chunk sizes are given in Figure 7. The default chunk size of 128 words seems to be something of a sweet spot, although there is a little more parallelism to be had in `fibheaps` and `constraints` when using a 32-word chunk size.

5.5 Lock contention

There are various mutexes in the parallel garbage collector, which we implement as simple spinlocks. The spinlocks are:

- The block allocator (one global lock)
- The remembered sets (one for each generation)
- The Pending Block Sets (one for each step)
- The large-object lists (one for each step)
- The per-object evacuation lock

The runtime system counts how many times each of these spinlocks is found to be contended, bumping a counter each time the requestor spins. We found very little contention for most locks. In particular, it is extremely rare that two threads attempt to evacuate the same object simultaneously: the per-object evacuation lock typically counts less than 10 spins per second during GC. This makes it all the more painful that this locking is so expensive, and it is why we plan to investigate relaxing this locking for immutable objects (Section 7).

There was significant contention for the block allocator, especially in the programs that use a large heap. To reduce this contention we did two things:

- We rewrote the block allocator to maintain its free list more efficiently.
- We allocate multiple blocks at a time, and keep the spare ones in the thread's Partly Free List (Section 3.6). At the end of GC any unused free blocks are returned to the block allocator.

5.6 Fragmentation

A block-structured heap will necessarily waste some memory at the end of each block. This arises when

- An object to be evacuated is too large to fit in the current block, so a few words are often lost at the end of a to-space block.
- The last to-space block to be allocated into will on average be half-full, and there is one such block for each step of each generation for each thread. These partially-full blocks will be used for to-space during the next GC, however.
- When work is scarce (see Section 3.6), partially-full blocks are exported to the pending block set. The GC tries to fill up any partially-full blocks rather than allocating fresh empty blocks, but it is possible that some partially-full blocks remain at the end of GC. The GC will try to re-use them at the next GC if this happens.

We measured the amount of space lost due to these factors, by comparing the actual amount of live data to the number of blocks allocated at the end of each GC. The runtime tracks the maximum amount of fragmentation at any one time over the run of a program and reports it at the end; we found that in all our benchmarks, the fragmentation was never more than 1% of the total memory allocated by the runtime. To put this in perspective, remember that copying GC wastes at least half the memory.

5.7 Eager Promotion

To our knowledge this is the first time the idea of eager promotion has been presented, and it is startlingly effective in practice. Figure 8 shows the benefit of doing eager promotion (Section 4.1), in the single-threaded GC. On average, eager promotion reduces the time spent in garbage collection by 6.8%. One example (`power`) went slower with eager promotion turned on - this program turns out to be quite sensitive to small changes in the times at which GC strikes, and this effect dominates.

Program	Δ GC time (%)
circsim	-1.0
constraints	-18.4
fibheaps	-2.7
fulsom	-14.8
gc_bench	-6.2
ghc	-3.6
happy	-14.2
lcss	-15.6
power	+36.6
spellcheck	-17.4
Min	-18.4
Max	+36.6
Geometric Mean	-6.8

Figure 8. Effect of adding eager promotion

5.8 Miscellany

Here we list a number of other techniques or modifications that we tried, but have not made systematic measurements for.

- Varying the native block size used by the block allocator. In practice this makes little difference to performance until the block size gets too small, and too large increases the amount of fragmentation. The current default of 4kbytes is reasonable.
- When taking a block from the Pending Block Set, do we take the block most recently added to the set (LIFO), or the block added first (FIFO)? Right now, we use FIFO, as we found it increased parallelism slightly, although LIFO might be better from a cache perspective. All other things being equal, it would make sense to take blocks of work recently generated by the current thread, in the hope that they would still be in the cache. Another strategy we could try is to take a random block, on the grounds that it would avoid accidentally hitting any worst-case behaviour.
- Adding more generations and steps doesn't help for these benchmarks, although we have found in the past that adding a generation is beneficial for very long-running programs.
- At one stage we used to have a separate to-space for objects that do not need to be scavenged, because they have no pointer fields (boxed integers and characters, for example). However, the runtime system has statically pre-allocated copies of small integers and characters, giving a limited form of hash-consing, which meant that usually less than 1% of the dynamic heap consisted of objects with no pointers. There was virtually no benefit in practice from this optimisation, and it added some complexity to the code, so it was discarded.
- We experimented with pre-fetching in the GC, with very limited success. Pre-fetching to-space ahead of the allocation pointer is easy, but gives no benefit on modern processors which tend to spot sequential access and pre-fetch automatically. Pre-fetching the scan block ahead of the scan pointer suffers from the same problem. Pre-fetching fields of objects that will shortly be scanned can be beneficial, but we found in practice that it was extremely difficult and processor-dependent to tune the distance at which to prefetch. Currently, our GC does no explicit prefetching.

6. Related work

There follows a survey of the related work in this area. Jones provides an introduction to classical sequential GC [JL96]. We focus on *tracing collectors* based on exploring the heap by reachability from root references. Moreover, we consider only parallel *copying* collectors, omitting those that use compaction or mark-sweep.

We also restrict our discussion to algorithms that are practical for general-purpose use.

Halstead [RHH85] developed a parallel version of Baker's incremental semispace copying collector. During collection the heap is logically partitioned into per-thread from-spaces and to-spaces. Each thread traces objects from its own set of roots, copying them into its own to-space. Fine-grained locking is used to synchronize access to from-space objects, although Halstead reports that such contention is very rare. As Halstead acknowledges, this approach can lead to *work imbalance* and to *heap overflow*. Halstead makes heap overflow less likely by dividing to-spaces into 64K-128K chunks which are allocated on demand.

Many researchers have explored how to avoid the work imbalance problems in early parallel copying collectors. It's impractical to avoid work imbalance: the collector does not know ahead of time which roots will lead to large data structures and which to small. The main technique therefore is to dynamically re-balance work from busy threads to idle threads.

Imai and Tick [IT93] developed the first parallel copying GC algorithm with dynamic work balancing. They divide to-space into blocks with each active GC thread having a "scan" block (of objects that it is tracing from) and a "copy" block (into which it copies objects it finds in from-space). If a thread fills its copy block then it adds it to a shared work pool, allocates a fresh copy block, and continues scanning. If a thread finishes its scan block then it fetches a fresh block from the work-pool. The size of the blocks provides a trade-off between the time spent synchronizing on the work-pool and the potential work imbalance. Siegwart and Hirzel [SH06] extend this approach to copy objects in hierarchical order.

Endo et al [ETY97] developed a parallel mark-sweep collector based on Boehm-Demers-Weiser conservative GC. They use work-stealing to avoid load-imbalance during the mark phase: GC threads have individual work queues and if a thread's own queue becomes empty then it steals work from another's. Endo et al manage work at a finer granularity than Imai and Tick: they generally use per-object work items, but also sub-divide large objects into 512-byte chunks for tracing. They found fine-granularity work management valuable because of large arrays in the scientific workloads that they studied. They parallelise the sweep phase by *over-partitioning* the heap into batches of blocks that are processed in parallel, meaning that there are more batches than GC threads and that GC threads dynamically claim new batches as they complete their work.

Flood et al [FDSZ01] developed a parallel semispace copying collector. As with Endo et al, they avoid work-imbalance by per-object work-stealing. They parallelise root scanning by over-partitioning the root set (including the card-based remembered set in generational configurations). As with Imai and Tick, each GC thread allocates into its own local memory buffer. Flood et al also developed a parallel mark-compact collector which we do not discuss here.

Concurrent with Flood et al, Attanasio et al [ABCS01] developed a modular GC framework for Java on large symmetric multiprocessor machines executing server applications. Unlike Flood et al, Attanasio et al's copying collector performed load balancing using work buffers of multiple pointers to objects. A global list is maintained of full buffers ready to process. Attanasio reports that this coarser mechanism scales as well as Flood et al's fine-grained design on the `javac` and SPECjbb benchmarks; as with Imai and Tick's design, the size of the buffers controls a trade-off between synchronization costs and work imbalance.

Also concurrent with Flood et al, Cheng and Blelloch developed a parallel copying collector using a shared stack of objects waiting to be traced [BC99, CB01]. Each GC thread periodically pushed part of its work onto the shared stack and took work from the shared stack when it exhausted its own work. The implementation of the stack is simplified by a gated synchronization mechanism so that pushes are never concurrent with pops.

Ben-Yitzhak et al [BYGK+02] augmented a parallel mark-sweep collector with periodic clearing of a "evacuated area" (EA). The basic idea is that if this is done sufficiently often then it prevents fragmentation building up. The EA is chosen before the mark phase and, during marking, references into the EA are identified. After marking the objects in the EA are evacuated to new locations (parallelized by over-partitioning the EA) and the references found during the mark phase are updated. In theory performance may be harmed by a poor choice of EA (e.g. one that does not reduce fragmentation because all the objects in it are dead) or by workload imbalance when processing the EA. In practice the first problem could be mitigated by better EA-selection algorithm and, for modest numbers of threads, the impact of the second is not significant.

This approach was later used in Barabash et al's parallel framework [BBYG+05]. Barabash et al also describe the "work packet" abstraction they developed to manage the parallel marking phases. As with Attanasio et al's work buffers, this provides a way to batch communication between GC threads. Each thread has one input packet from which it is taking marking work and one output packet into which it places work that it generates. These packets remain distinct (unlike Imai and Tick's chunks [IT93]) and are shared between threads only at a whole-packet granularity (unlike per-object work stealing in Endo et al's and Flood et al's work). Barabash et al report that this approach makes termination detection easy (all the packets must be empty) and makes it easy to add or remove GC threads (because the shared pool's implementation is oblivious to the number of participants).

Petrank and Kolodner [PK04] observe how existing parallel copying collectors allocated objects into per-thread chunks, raising the possibility of a fragmentation of to-space. They showed how this could be avoided by "delayed allocation" of to-space copies of objects: GC threads form batches of proposed to-space allocations which are then performed by a single CAS on a shared allocation pointer. This guarantees that there is no to-space fragmentation while avoiding per-allocation CAS operations. In many systems the impact of this form of fragmentation is small because the number of chunks with left-over space is low.

There are thus two kinds of dynamic re-balancing: fine-grained schemes like Endo et al [ETY97] and Flood et al [FDSZ01] which work at a per-object granularity, and batching schemes like Imai and Tick [IT93], Attanasio et al [ABCS01] and Barabash et al [BBYG+05] which group work into blocks or packets. Both approaches have their advantages. Fine-grained schemes reduce the latency between one thread needing work and it being made available and, as Flood et al argue, the synchronization overhead can be mitigated by carefully designed work-stealing systems. Block-based schemes may make termination decisions easier (particularly if available work is placed in a single shared pool) and make it easier to express different traversal policies by changing how blocks are selected from the pool (as Siegwart and Hirzel's work illustrated [SH06]).

We attempt to combine the best of these approaches. In particular we try to keep the latency in work distribution low by using producing incomplete blocks if there are idle threads. Furthermore, as with Imai and Tick's [IT93] and Siegwart and Hirzel's [SH06] work, we represent our work items by areas of to-space, avoiding the need to reify them in a separate queue or packet structure.

A number of collectors have exploited the immutability of most data in functional languages. Doligez and Leroy's concurrent collector for ML [DL93] uses per-thread private heaps and allows multiple threads to collect their own private heaps in parallel. They simplify this by preserving an invariant that there are no inter-private-

heap references and no references from a common shared heap into any thread's private heap: the new referents of mutable objects are copied into the shared heap, and mutable objects themselves are allocated in the shared heap. This exploits the fact that in ML (as in Haskell) most data is immutable. Huelsbergen and Larus' concurrent copying collector [HL93] also exploits the immutability of data in ML: immutable data can be copied in parallel with concurrent accesses by the mutator.

7. Conclusion and further work

The advent of widespread multi-core processors offers an attractive opportunity to reduce the costs of automatic memory management with zero programmer intervention. The opportunity is somewhat tricky to exploit, but it can be done, and we have demonstrated real wall-clock benefits achieved by our algorithm. Moreover, we have made progress toward explaining the lack of perfect speedup by measuring the load imbalance in the GC and showing that this correlates well with the wall-clock speedup.

There are two particular directions in which we would like to develop our collector.

7.1 Reducing per-object synchronisation

As noted in Section 3.1, a GC thread uses an atomic CAS instruction to gain exclusive access to a from-space heap object. The cost of atomicity here is high: 20-30% (Section 5.1), and we would like to reduce it.

Many heap objects in a functional language are *immutable*, and the language does not support pointer-equality. If such an immutable objects is reachable via two different pointers, it is therefore semantically acceptable to copy the object *twice* into to-space. Sharing is lost, and the heap size may increase slightly, but the mutator can see no difference.

So the idea is simple: for immutable objects, we avoid using atomic instructions to claim the object, and accept the small possibility that the object may be copied more than once into to-space. We know that contention for individual objects happens very rarely in the GC (Section 5.5), so we expect the amount of accidental duplication to be negligible in practice.

7.2 Privatising minor collections

A clear shortcoming of the system we have described is that all garbage collection is global: all the processors stop, agree to garbage collect, perform garbage collection, and resume mutation. It would be much better if a mutator thread could perform local garbage collection on its private heap without any interaction with other threads whatsoever. We plan to implement such a scheme, very much along the lines described by Doligez and Leroy [DL93].

References

[ABCS01] C. Attanasio, D. Bacon, A. Cocchi, and S. Smith. A comparative evaluation of parallel garbage collectors. In *Fourteenth Annual Workshop on Languages and Compilers for Parallel Computing*, pages 177–192, Cumberland Falls, KT, 2001. Springer-Verlag.

[ABP98] Nimar S. Arora, Robert D. Blumofe, and C. Greg Plaxton. Thread scheduling for multiprogrammed multiprocessors. pages 119–129. ACM Press, June 1998.

[BBYG+05] Katherine Barabash, Ori Ben-Yitzhak, Irit Goft, Elliot K. Kolodner, Victor Leikehman, Yoav Ossia, Avi Owshanko, and Erez Petrank. A parallel, incremental, mostly concurrent garbage collector for servers. *ACM Trans. Program. Lang. Syst.*, 27(6):1097–1146, 2005.

[BC99] Guy E. Blelloch and Perry Cheng. On bounding time and space for multiprocessor garbage collection. In *PLDI '99: Proceedings of the ACM SIGPLAN 1999 conference on Programming language design and implementation*, pages 104–117. ACM, 1999.

[BYGK+02] Ori Ben-Yitzhak, Irit Goft, Elliot K. Kolodner, Kean Kuiper, and Victor Leikehman. An algorithm for parallel incremental compaction. In *ISMM '02: Proceedings of the 3rd international symposium on Memory management*, pages 100–105. ACM, 2002.

[CB01] Perry Cheng and Guy E. Blelloch. A parallel, real-time garbage collector. In *PLDI '01: Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation*, pages 125–136. ACM, 2001.

[Che70] C. J. Cheney. A nonrecursive list compacting algorithm. *Commun. ACM*, 13(11):677–678, 1970.

[DEB94] R. Kent Dybvig, David Eby, and Carl Bruggeman. Don't stop the BIBOP: Flexible and efficient storage management for dynamically-typed languages. Technical Report 400, Indiana University Computer Science Department, 1994.

[DL93] Damien Doligez and Xavier Leroy. A concurrent, generational garbage collector for a multithreaded implementation of ML. In *POPL '93: Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 113–123. ACM, 1993.

[ETY97] Toshio Endo, Kenjiro Taura, and Akinori Yonezawa. A scalable mark-sweep garbage collector on large-scale shared-memory machines. In *Supercomputing '97: Proceedings of the 1997 ACM/IEEE conference on Supercomputing (CDROM)*, pages 1–14. ACM, 1997.

[FDSZ01] Christine Flood, Dave Detlefs, Nir Shavit, and Catherine Zhang. Parallel garbage collection for shared memory multiprocessors. In *Usenix Java Virtual Machine Research and Technology Symposium (JVM '01)*, Monterey, CA, 2001.

[HL93] Lorenz Huelsbergen and James R. Larus. A concurrent copying garbage collector for languages that distinguish (im)mutable data. *SIGPLAN Not.*, 28(7):73–82, 1993.

[HMP05] Tim Harris, Simon Marlow, and Simon Peyton Jones. Haskell on a shared-memory multiprocessor. In *ACM Workshop on Haskell*, Tallin, Estonia, 2005. ACM.

[IT93] A. Imai and E. Tick. Evaluation of parallel copying garbage collection on a shared-memory multiprocessor. *IEEE Trans. Parallel Distrib. Syst.*, 4(9):1030–1040, 1993.

[JL96] Richard Jones and Rafael Lins. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. John Wiley and Sons, July 1996.

[Par92] WD Partain. The `nofib` benchmark suite of Haskell programs. In J Launchbury and PM Sansom, editors, *Functional Programming, Glasgow 1992*, pages 195–202. 1992.

[PK04] Erez Petrank and Elliot K. Kolodner. Parallel copying garbage collection using delayed allocation. *Parallel Processing Letters*, 14(2), June 2004.

[RHH85] Jr. Robert H. Halstead. Multilisp: a language for concurrent symbolic computation. *ACM Trans. Program. Lang. Syst.*, 7(4):501–538, 1985.

[SH06] David Siegwart and Martin Hirzel. Improving locality with parallel hierarchical copying gc. In *ISMM '06: Proceedings of the 5th international symposium on Memory management*, pages 52–63. ACM, 2006.

[Ste] Don Stewart. nobench: Benchmarking haskell implementations. `http://www.cse.unsw.edu.au/~dons/nobench.html`.

[Ste77] Guy Lewis Steele Jr. Data representations in PDP-10 MacLISP. Technical report, MIT Artificial Intelligence Laborotory, 1977. AI Memo 420.

[Ung84] D Ungar. Generation scavenging: A non-disruptive high performance storage management reclamation algorithm. In *ACM SIGPLAN Software Engineering Symposium on Practical Software Development Evironments*, pages 157–167. Pittsburgh, Pennsylvania, April 1984.