

Multicore Garbage

Simon Marlow

Microsoft Research, Cambridge, U.K.
simonmar@microsoft.com

Abstract

In a parallel, shared-memory, language with a garbage collected heap, it is desirable for each processor to perform minor garbage collections independently. Although obvious, it is difficult to make this idea pay off in practice, especially in languages where mutation is common. We present several techniques that substantially improve the state of the art. We describe these techniques in the context of a full-scale implementation of Haskell, and demonstrate that our local-heap collector substantially improves scaling, peak performance, and robustness.

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

1. Introduction

In a garbage collected environment, multithreaded programs can run into an “allocation wall” (Zhao et al. 2009), in which performance is limited by the rate at which newly allocated data can be written to main memory, and adding more cores does not improve performance once the limit is reached. One way to avoid the allocation wall is to use a generational collector with per-thread nurseries each smaller than the size of the L2 cache, so that most memory accesses hit the cache rather than main memory. However, such a small nursery size entails very frequent collections, and with a stop-the-world collector this requires frequent synchronisation across processors, which also hurts performance as the number of processors increases. Moreover, performance becomes more fragile at scale, because latency in a single core can halt the whole system — and that is exactly what happens if the operating system de-schedules the language runtime in favour of another process running on that core.

To avoid the synchronisation inherent in stop-the-world collection, one might turn to concurrent GC. However, running the collector on a separate core from the mutator is also suboptimal from a cache perspective. Concurrent GC is therefore not likely to solve the problem of scaling nursery allocation, but is more appropriate for collecting a large old generation. In this work we focus on throughput rather than latency and pause-times, and hence we do not consider concurrent GC further.

To address the cost of frequent stop-the-world synchronisation while still maintaining locality, there have been several attempts

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

ISMM'11, June 4–5, 2011, San Jose, California, USA.

Copyright © 2011 ACM 978-1-4503-0263-0/11/06...\$10.00

Collection with Local Heaps

Simon Peyton Jones

Microsoft Research, Cambridge, U.K.

simonpj@microsoft.com

to design collectors in which each processor has a private heap that can be collected independently without synchronising with the other processors; there is also a global heap for shared data. Some of the existing designs are based on static analyses to identify objects whose references never escape the current thread and can therefore be allocated in the local heap (Jones and King 2005; Steensgaard 2000), while others employ a dynamic approach in which objects are allocated in the local heap, but may subsequently

be moved to the global heap if necessary (Anderson 2010; Doligez and Leroy 1993; Domani et al. 2002). In this paper we present a new garbage collector of the latter kind, with some novel techniques that improve on the existing designs. Specifically, our contributions are as follows:

- We present data quantifying the principal shortcoming of existing techniques, namely the costs associated with writing to global heap objects (Section 3).

Handling mutation (writes) well is important: Java-like languages make heavy use of *explicit* mutation in the form of writes to object fields, but even in a pure language (Haskell in our case) there is a great deal of *implicit* mutation of heap objects due to lazy evaluation.

- We present two new ideas that together help alleviate the penalty for mutation in a local-heap collector (Section 4):
 - Our heap structure allows pointers from the global heap to the local heap, protected by a read barrier. This technique reduces the cost of the write barrier by avoiding premature promotion of mutable objects into the global heap.
 - We use a combination of moving and non-moving collection in the local heap, which allows both mutable and immutable objects to be allocated in the local heap, while retaining the efficiency of bump-pointer allocation in the common case.
- Our new design gives rise to a family of policy decisions, concerning exactly when and how much to promote from the local heap to the global heap. We explore these designs, and shed some light on where the most effective solutions probably

lie (Section 6.3).

- We have implemented our collector in the Glasgow Haskell Compiler (a state-of-the-art optimising compiler for Haskell), and we demonstrate that our new collector yields improved scaling and peak throughput on a substantial collection of parallel Haskell benchmarks, on average improving performance by 15% at 24 cores, compared to the baseline parallel generational copying collector (Section 6.2). Moreover, the performance is less sensitive to having exclusive access to a fixed number of cores; performance drops less sharply when cores are removed compared with the stop-the-world collector (Section 6.2.3).

While our work is motivated by a desire to make parallel Haskell programs go faster, throughout the paper we delimit the parts of our design that are specific to Haskell and the GHC compiler, and those that should apply in other settings. In particular, we believe that our main results related to mutation should also apply in settings such as .NET and Java.

2. Garbage collection with processor-local heaps

The overall memory architecture is this. Each processor has its own *local heap*, in which it allocates, and which (crucially) it can garbage-collect independently of other processors. In addition there is a shared *global heap* which is visible to all processors, and which is only collected when all processors synchronise and cooperate in a (parallel) global garbage collection. We call this a *local heap collector*. Other terms have been used in the literature, notably *private nursery*, *thread-local* or *thread-specific heap*, and *on-the-fly collection*.

There are two established approaches to organising the heap in a local-heap collector:

- In the Doligez and Leroy (1993) design, and the later Anderson (2010) design, the local heap is collected with a copying collector. The global and local parts of the heap are segregated by address.
- In the Domani et al. (2002) design, the local heap is collected with a non-moving algorithm (mark-sweep). This allows objects to be relocated from the local heap to the global heap without physically copying them; a separate bitmap indicates which

objects in the local heap are global. Local and global objects are therefore intermingled in the address space.

In both of these designs, the key invariant is that *the processor that owns the local heap has exclusive access to its contents*. The owning processor is therefore free to do local garbage collection without disturbing objects that are being read or modified by other processors.

The invariant is maintained by banning pointers from the global heap to the local heap, because one of these would allow a mutator to follow a pointer into a foreign local heap. To maintain the invariant, whenever a local-heap pointer is written into a global-heap object, or is communicated to another processor, a *write barrier* must detect the potential breakage, and somehow fix it up.

The existing designs take different approaches to this write barrier. In the original Doligez-Leroy design, before writing a local-heap pointer into a global mutable object we first *globalise* the local-heap object by copying it from the local heap to the global heap. Since it may contain further local-heap pointers, they too must be globalised, so the net effect is to globalise the transitive closure of the local heap pointer. (Globalisation is a bit like the *promotion* of generational GC, but its timing and purpose are different, so a different term is useful.)

Mutable objects complicate globalisation. Since mutable objects have identity and cannot be copied¹, mutable objects are always allocated directly in the global heap. Hence mutable objects and mutation are likely to be costly; the setting for this design was a strict functional language in which mutation was rare, which explains the choices made here.

In the Anderson variant of Doligez-Leroy, the write barrier

triggers a local collection, with a refinement to catch some common cases where the full local GC is not necessary. This design allows mutable objects to be allocated in the local heap, but at a substantial cost: the write barrier may trigger a complete local GC, which in

¹ without using a replicating write barrier or suchlike
turn will tend to cause global GC to happen more frequently than it would otherwise have.

The Manticore system (Fluet et al. 2008), implements a variant of the Doligez-Leroy design in which each local heap is a separate Appel-style generational collector. Manticore does not provide mutation in any form to the programmer, however.

The Domani et. al. design is similar to Doligez-Leroy in that objects in the transitive closure of the local heap pointer are made global, but since the collector is non-moving, these objects are simply marked as being global and left in place. Again, this design allows mutable objects to be allocated in the local heap, which was important in this case because the setting was Java in which *all* heap objects are potentially mutable. A disadvantage is that the local heap must be collected with mark-sweep, which is known to have an impact on allocation performance (Blackburn et al. 2004).

3. The problem of mutation

Although the collectors described in the previous section allow mutation in the heap, a problem common to all of the existing designs is that a mutable object in the global heap has considerable cost: *every mutation of that object causes retention of the entire transitive closure of the pointer written, until the next global GC.* What is particularly annoying is that

1. Mutable objects are often repeatedly mutated. The write barrier preserves the transitive closure of *every single value written* into a global mutable object until the next global GC, even though these values may be overwritten almost immediately.
2. The effect is viral. If the value written into a global mutable object M1 contains (transitively) a currently-local, mutable object M2, then M2 must be globalised. Hence M2 is subject to the write barrier, and anything written to it must be globalised, and so on.

The first effect is unavoidable: if the value is written into a global mutable location, another processor might read it before it is overwritten, so the value must be preserved². The trick is to stop the mutable object becoming global in the first place — but the viral consequences of transitive promotion make that hard.

Intuitively, these effects seem likely to lead to a great deal of ultimately-fruitless globalisation, reducing locality, and triggering expensive global GC more often than necessary. We set out to quantify this effect in the context of Haskell. In Haskell, *explicitly* mutable objects are usually far less common than immutable objects. Nevertheless, *implicit* mutation of heap objects is rife at runtime, thanks to the implementation of lazy evaluation³. A lazy computation is represented by a *thunk*: a closure of the code to compute the value together with its free variables. When the value of a thunk is demanded, its value is computed and then the thunk is overwritten with an indirection to the value. This write operation is called an *update*, and is a frequent source of mutation in the Haskell heap.

3.1 Quantifying the effect of transitive globalisation

We measured the cost of transitive globalisation by modifying a conventional, single-threaded, two-generation collector. Table 1 shows the performance of several single-threaded benchmarks with three different configurations for the old-generation write barrier (there is no barrier for writes to the young generation):

² This is worse than the promotion semantics of most generational collectors, which only retain the data of the last update preceding a minor collection. However “snapshot-at-the-beginning” concurrent collectors (Pirinen 1998) also have the property that they retain all values written until the next GC cycle.

³ which, we admit, is somewhat ironic.

Program	% change in wall-clock time	
	promote transitive	promote immutable
circsim	+43.9	+3.7
constraints	+124.0	+2.0
fibheaps	+46.7	-0.6
fulsom	+49.6	+6.6
gc_bench	-63.0	-64.4
happy	+10.9	-0.1
lcss	+82.0	-0.3
mutstore1	-2.1	-2.4
mutstore2	+0.0	+0.2
power	+7.8	+15.7
spellcheck	+218.8	-0.8
Min	-63.0	-64.4
Max	+218.8	+15.7
Geometric Mean	+29.9	-7.1

Table 1. Comparison of write barrier promotion policies

- The baseline: standard generational collection, where writes to the old-generation are recorded in a remembered set.
- “Promote transitive”: a write to the old generation immediately promotes the transitive closure of the pointer written. This models the invariant of Doligez-Leroy and Domani *et al.*
- “Promote immutable”: a write to the old generation promotes the object it points to recursively, but avoids promoting mutable

objects (in our case, *thunks*). Mutable objects are left in the young generation and an entry in the remembered set is created for the pointer.

The “promote transitive” policy incurs a significant overhead: 29.9% on average, while “promote immutable” provides similar performance to the baseline (with one outlier that performs significantly better with eager promotion, *gc_bench*, which we discuss below). This suggests that the premature promotion caused by effect (2) above, and its knock-on effects, have a significant impact on performance.

In *gc_bench*, promoting writes eagerly has a huge benefit, because performing the promotion avoids having to extend the remembered set. This is a GC microbenchmark and we would be unlikely to see the effect on this scale in a real program.

The bottom line is this: the viral globalisation of mutable objects into the global heap carries a significant cost. These costs are almost certainly under-stated in Table 1, which is derived from modifying a *single-threaded* generational collector. Our real goal is to allow independent local-heap garbage collection in a *parallel* machine.

4. Our design: improving support for mutation

Our design has two main novelties, both focussed on improving the performance of mutation in a local heap collector.

- **Allow pointers from the global heap to a local heap, protected by a read barrier.** We replace the invariant that objects

in the global heap may not point to objects in the local heap, with a read barrier that checks for global-to-local pointers. Such pointers may only be followed directly by the processor that owns the appropriate local heap. Other processors that attempt to read the pointer are required to communicate with the owning processor to request that the data be moved into the global heap.

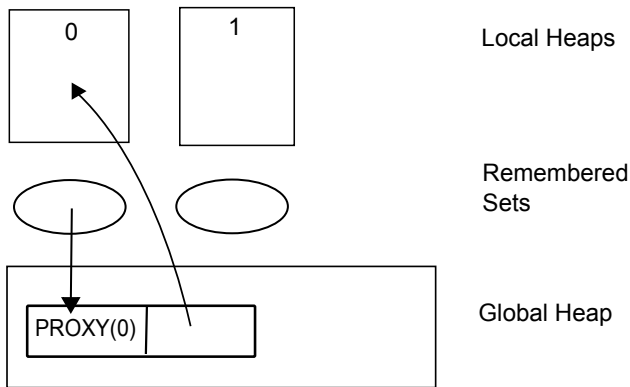


Figure 1. Heap architecture showing a proxy indirection

Admitting pointers from the global heap to the local heap allows us to avoid globalising the transitive closure of every write into the global heap, and thus avoid the performance penalty measured in the previous section.

- **The sticky heap: no read barrier.** We divide the local heap into two parts⁴. The first is a traditional nursery in which objects are allocated using bump-pointer allocation and memory is reclaimed using copying GC, as in typical generational-copying designs. A separate part of the local heap, that we call the *sticky heap*, is where we allocate objects that lack a read barrier, and hence must be immovable (Section 4.4), including mutable objects and objects with identity. The sticky heap is collected using mark-sweep GC.

This aspect of our design is a combination of the Doligez-Leroy design (immutable objects with copying GC), with the Domani et. al. approach (mutable objects with mark-sweep GC). It allows us to retain fast allocation and collection in the common case, while allowing both mutable and immutable objects to be allocated and reclaimed in the local heap.

To summarise, the read-barrier allows us to globalise fewer objects, while the sticky heap allows us to selectively choose to omit the read barrier for some objects while mostly retaining efficient bump-pointer allocation. These are the two key aspects of our design; the following sections discuss the implementation of these ideas.

4.1 Implementing the read barrier: proxy indirections

The read barrier must perform the following operations when dereferencing any pointer stored in a global heap object:

- If the pointer points to a local heap, and it is not the local heap of the current processor, then send a message to the owning

processor requesting that the pointer's referent be moved to the global heap. Block the current thread until a response is received.

In GHC⁵, *every pointer dereference already has a read-barrier*, because an object could be represented by an unevaluated thunk. Whenever a pointer is dereferenced, we test tag bits in the pointer to determine whether the pointer points to a value or not; if not, then the caller jumps to the code for the object, which is expected to perform whatever computation is necessary and eventually return the value.

The existing read barrier identifies a property of the object pointed to, whereas in our local-heap implementation we need to

⁴ In fact, there is also a large-object area (Section 5.6)

⁵ The Glasgow Haskell Compiler, <http://www.haskell.org/ghc/>

distinguish pointers into the global heap from pointers into a local heap. Nevertheless, we would like to use the same read barrier, so as to avoid adding new overhead to every pointer dereference. The technique we use is to represent global-to-local pointers by a new kind of heap object in the global heap that we call a *proxy indirection*, or just proxy. A proxy has two fields: the pointer to the local object, and an integer identifying the processor that owns that local heap. Figure 1 shows a diagram of our heap architecture including a proxy indirection.

Pointers from the global heap to the local heap are always represented by proxies. To the existing read barrier, a proxy looks like a thunk, so the caller will jump to its code, which in the case of a proxy implements the rest of the read barrier for global-to-local pointers:

- If the owner of the proxy is the current processor, continue; otherwise
- Send a message to the owner containing the addresses of the proxy and the current thread, and block the current thread until a reply is received. The details of the message communication between processors is described later in Section 5.7.

In GHC the read barrier on global-to-local pointers therefore carries no additional overhead, except that we need to create proxy objects as necessary. In other systems, a suitable read barrier would need to be used, and that would necessarily impose some overhead. It has been shown that with careful optimisation a read barrier in Java can be implemented with only 4-10% overhead (Bacon et al. 2003); this compares favourably with the 30% overhead we found for promoting mutable objects too early (previous section).

4.2 The write barrier

As in other local heap designs, our collector requires a write barrier. Fortunately the write barrier can be piggy-backed on the existing generational write barrier, since it only applies to writes to objects in the old generation (i.e. the global heap). The write barrier maintains the following invariant:

- There are no pointers from the global heap to the local heap, except for proxy indirections.

The write barrier must track the proxy indirections so that they can be treated as roots and updated by local GC. Our implementation uses remembered sets, one per processor, each containing the set of proxy indirections pointing to that processor's local heap.

The write barrier must catch a write of a local pointer into a global object. When this action occurs, the write barrier is free to implement a range of policies, provided it establishes the invariant. We have implemented the following policies:

- Create a single new proxy indirection in the global heap, containing the local pointer.
- Globalise some or all of the data referred to by the local pointer into the global heap. At any point we can elect to stop globalising and create a proxy indirection.

We present measurements comparing these policies in Section 6.3.

4.3 Globalising an object

Making a local object global is called *globalising* it. Exactly how

we globalise an object depends on the kind of object:

- **Immutable objects** (constructors and functions) reside in the movable portion of the local heap, and are copied into the global heap to globalise them. There may be other pointers to the local copy, so we overwrite the header of the local copy with a *forwarding pointer* to the global object, so that the local copy will be collected at the next local collection.

The object header is used mainly by the garbage collector, and is seldom read by the mutator (Marlow et al. 2007). However, if the mutator does need to read the object header, it must be careful to dereference a forwarding pointer.

- **Mutable objects**, such as mutable variables and arrays, cannot be copied, and so (following Domani et al. (2002)) we allocate them in immovable storage: the sticky heap (we describe the sticky heap below, Section 4.4).
- **Thunks** are objects that represent an unevaluated computation (and are thus specific to lazy evaluation). A thunk is a closure over an expression, and therefore contains pointers to the free variables of the expression. After evaluation, the thunk is overwritten with an indirection to its value: this is a one-time mutation, replacing the fields pointing to the free variables with a single field pointing to the value.

Strictly speaking, thunks are mutable objects, but we treat them specially because they come with a built-in read barrier. When the value of an object is required, the mutator already has to check whether the object has been evaluated or not; if it is not a value, then it has to be evaluated, and that is achieved by jump-

ing to the object's code. Normally the object is a thunk, and jumping to its code causes its evaluation. However, an object may also be an indirection to another object, and evaluating an indirection is equivalent to evaluating the object it points to.

Hence, we can allocate thunks in the movable nursery, and to globalise a thunk, we can *move* the thunk to the global heap, replacing it in the local heap with an indirection to its new location in the global heap.

4.4 The sticky heap

The sticky heap is a part of the local heap used to store objects that cannot move. Most objects are movable: immutable objects can be copied, and thunks have a read-barrier that enables them to be replaced by indirections. The remaining class of objects, mutable objects, could only be made movable by adding a new read-barrier to their operations, and to do so would add overhead and complexity. Since these objects tend to be in the minority in Haskell, we opted for an alternative approach: mutable objects are immovable while in the local heap.

As we argued earlier, generally we would like to avoid globalising mutable objects if possible. However sometimes it is unavoidable: if a mutable object is really shared between multiple processors, it must be globalised.

Each processor therefore has its own *sticky heap*, where it allocates mutable objects. Objects in the sticky heap are born local, and can be reclaimed by local GC. However, if necessary they can be globalised without changing their address, by flipping a *global bit* attached to each sticky object (details in Section 5.5). Once a sticky

object is thus globalised it becomes part of the global heap, and can only be recovered by global GC.

Note that the lack of read barriers on sticky objects means that whenever we encounter one during globalisation we have no option but to globalise it. This seems counter to our policy of not globalising mutable objects, and indeed it is – although in our setting it is far more important that the policy applies to thunks than to these explicitly-mutable objects.

Strictly speaking the sticky heap is an optional part of our design. The alternative is to use a read barrier consistently; whether this is the right choice depends on the particular costs involved. One should think of the design space as continuous, with the Domani et al. (2002) design at one end in which the entire heap is sticky,

and at the other end there is no sticky heap but a read barrier is used consistently. In between are points in the design space in which heap objects are divided into those with a read barrier and those that are sticky. We contend that the read barrier is necessary to avoid the effects described in Section 3.1, and therefore the read barrier should be used for the majority of objects; however it may also make sense to omit the read barrier for certain objects and store them in a sticky heap instead. We cannot speculate on what the appropriate tradeoff for a different language might be, but if a read barrier is being added then it would make sense to measure the impact of that first, before deciding whether to reduce the read barrier costs by classifying certain objects as sticky.

4.5 Managing the parallel work queues

To provide load-balancing in the Parallel Haskell implementation, each processor has a *spark pool*: a circular array of pointers to heap objects supporting lock-free work-stealing (Marlow et al. 2009). Each processor may put work items in its spark pool, and processors may take work items from the local spark pool or *steal* them from other processors' spark pools.

This approach works nicely in a completely shared heap, because adding an entry to the spark pool is a straightforward write into the circular buffer. However, in the local-heap setting we must treat the spark pool as a global object where writes are subject to the write barrier, because other processors may steal from it, and they may only steal global pointers.

As with other writes to the global heap, we have to decide how much data to globalise for each write (see Section 4.2). In the case of a spark pool write, globalising more data would make stealing

cheaper at the expense of greater overhead when adding sparks to the pool. Conventional wisdom is to load costs onto the steal rather than the spark, but doing so uncritically risks increasing startup latency, because a stealing processor must first ask the originating processor to globalise the spark before it can get to work. If we can arrange to have relatively few large-granularity sparks – using lazy tree-splitting, for example (Bergstrom et al. 2010) – eager globalisation of sparked work might be a better policy.

In our system, by changing the write barrier policy, we can simulate a range of alternatives, from an approach in which sparks are cheap but every steal incurs a message exchange, to a system which has completely asynchronous steals but where sparks are relatively expensive because they have to copy data to the global heap. We present some measurements in Section 6.3.2 to compare these approaches.

5. Implementation Details

In this section we describe our implementation in greater detail.

5.1 The block layer

The lowest layer of the GHC garbage collector is the *block allocator* (Marlow et al. 2008). The block allocator’s API allows memory *blocks* to be allocated and freed, where each block is a multiple of 4Kbytes in size. Internally the block allocator requests memory from the OS in large units (typically a megabyte), and uses an efficient free-list of blocks in which most operations are $O(1)$.

Every area of memory that the garbage collector manages, including the nursery, is represented as a linked list of (possibly discontiguous) blocks. Hence, the garbage collector is completely in-

sensitive to address-space layout, which is good for portability, and it can easily manage multiple regions (for different generations, say) whose size varies over time.

Each block has a small amount of metadata associated with it, called the *block descriptor*. A simple calculation maps an arbitrary memory address to its block descriptor. The block descriptor contains a link field for chaining blocks together, other information such as which generation the block belongs to, and some flags. Our sticky heap, for example, is represented by a chain of blocks that each have the STICKY flag set in the block descriptor.

5.2 Virtual processors

The runtime system uses a number of virtual processors that we call HECs (Haskell Execution Context). The number of HECs is chosen at startup time, and cannot currently be changed during the run of a program. Typically the number of HECs is chosen to be the same as the number of hardware cores; the reader should think of a HEC as being approximately equivalent to a processor.

Each HEC is “animated” by an OS thread. In fact there may be many such OS threads for a single HEC, because our runtime creates extra OS threads on demand, to handle blocking system calls (Marlow et al. 2004). However, the scheduler allows only one OS thread per HEC to run at any one time. The OS threads (and hence the HEC) can be pinned to hardware cores using the OS’s affinity APIs, although we find in practice that this makes little difference to performance and in some cases actually degrades it.

Each HEC runs its own scheduler, and has its own queue of runnable Haskell threads, and its own local heap. A HEC may create new Haskell threads to run parallel sparks stolen from other HECs.

5.3 Local heap collections

Our collector supports *aging* in the local heap: objects have to survive at least one garbage collection in the local heap before being moved to the global heap. We found that aging objects at least one GC cycle was important for performance, because we avoid some premature promotion, but aging more than one GC cycle is a pessimisation due to the extra copying entailed. Aging is implemented by grouping objects by age: all live objects in the nursery are copied to a separate area containing objects that have survived one GC, and live objects in this area are copied to the global heap.

The sticky heap has to be collected using mark-sweep: we cannot move any of the global objects in it, because other processors may be accessing them concurrently. However, we could move *local* objects in the sticky heap to reduce fragmentation; currently our implementation does not do this (though the Domani et al. (2002) collector does).

Our sweeping reclamation algorithm is based on the Immix mark-region strategy (Blackburn and McKinley 2008), in which memory is reclaimed at a granularity larger than a single object in order to speed up sweeping and allocation. GHC's block-based memory allocation scheme (Section 5.1) is a perfect fit: when mark-sweep finds a complete block with no live objects, it can simply return it to the block allocator. The granularity at which we can free memory is somewhat larger than that used by Immix, which may lead to fragmentation. However, in our case this is not a serious problem, since at the next global GC we will compact the sticky heap anyway, and fragmentation will simply cause the global GC

to happen a bit sooner. The sweeping algorithm therefore classifies blocks in the sticky heap as

- **Free:** the block has no live objects at all, and can be immediately reclaimed.
- **Global:** a block that contains at least one global object is marked global, and never swept again. We do not attempt to reclaim unused memory in these blocks until the next global GC, when live objects in the block will be copied out and the block can be re-used. We found that this optimisation was particularly important for programs that make heavy use of mutable objects, otherwise each local collection sweeps an ever-growing immovable region.

- **Local:** the block has live local objects in it; we aggregate free space in the block into extents, in order to speed up future sweeps, but otherwise do not attempt to re-use it.

We cannot age objects in the sticky heap, because the aging implementation relies on copying in order to group objects by age. A variety of policies are possible but our current policy is this: sticky objects are never promoted by local GC but are always promoted by global GC. So after global GC all the sticky heaps are empty.

5.4 Global heap collections

The global heap is collected with stop-the-world parallel collection, exactly as described in Marlow et al. (2008). In the default configuration, the global heap is collected when it has doubled in size since the last global collection, with a minimum of 1MB. This provides a reasonable tradeoff between collection frequency and memory usage. However, for the purposes of comparative measurements between collectors in Section 6, we use a fixed heap size configuration and collect the global heap when the total heap size has grown to half of this size (to allow for copying).

5.5 Where to store the “global” bit in the sticky heap

Each object in the sticky heap needs to have an associated flag to indicate whether it belongs to the local or the global heap. The approach we chose is to allocate an extra word *before* each object in the sticky heap, which is zero if the object is local and non-zero otherwise. We considered two alternative approaches:

- Store the global bit in the object itself: either a bit in the object's header, or a bit in the object's metadata (which is pointed to by the header). This would be more complex than the approach we took: we would either need to modify code that inspects object headers to mask out the bit, or we would need to change an object's metadata when we globalise it, which would be dependent on the kind of object being globalised.
- Store the global bit in a separate bitmap. This is the approach taken in Domani et al. (2002). This would be slower than the approach we took, although it would waste less memory. We considered this to be an appropriate tradeoff, given that in our setting we expect objects in the sticky heap to be in the minority. The wasted memory in our case only applies while the object is in the sticky heap; there are no extra words once a global collection has taken place and the objects are moved to the global heap proper.

5.6 Large objects

Objects larger than a certain threshold (currently about 3KB) are classed as “large objects” and are never copied by the GC. Instead they are allocated in a contiguous region of blocks, and stored in a linked list associated with the heap to which they belong. Moving a large object from the local heap to the global heap therefore consists of removing it from the linked list in the local heap, and adding it to the global heap's list.

The fact that large objects are immovable is useful, because it means that a large mutable object (such as an array) does not need to be allocated in the sticky heap, and it can be managed in the same

way as other large objects.

5.7 Requesting private data from another processor

When one processor encounters a proxy indirection that belongs to another processor, it sends a message to the other processor to request globalisation of the data referred to by the proxy. The (Haskell) thread making the request is placed into a blocked state until the other processor replies; meanwhile the HEC runs some other thread.

On receipt of the message, the owner of the proxy globalises the data. Just as for the write barrier in Section 4.2, there is a policy decision to make about how far to globalise, but in this case the owner must globalise at least *some* of the data because it is required by the other processor.

Having globalised the data, the owner then overwrites the proxy with a plain indirection to the now-global object, and sends a reply to wake up the blocked thread on the original processor.

As a special case, if the other processor is idle, then the processor that encountered the proxy can simply take control of the other processor's local heap temporarily in order to perform the globalisation without the need to incur the cost of the message exchange and waking up the idle OS thread. This is quite an important optimisation: we found that in some of our benchmarks it was common for a processor to run out of work and become idle while holding data in the local heap needed by other processors. We experimented with having idle processors do a local GC before sleeping, but found that this lead to a large number of local GCs and thrashing in some cases.

6. Measurements

Our measurements were made on a 24-core machine consisting of 4 Intel Xeon E7450 processors (2.4GHz), running Windows Server 2008. We compiled our benchmarks to 32-bit code. The results for 64-bit code are broadly similar, but differences in performance tend to be amplified at 64 bits due to the greater stress put on the memory system.

As our baseline for comparison we use GHC HEAD as of 21 January 2011, and our implementation of the local-heap GC is based directly on this GHC version.

6.1 Benchmarks

Our benchmarks are a collection of parallel Haskell programs. They are all deterministically parallel, and use the internal spark mechanism for parallelism, rather than explicit threads.

- **blackscholes**: An implementation of the Black-Scholes algorithm for modelling financial contracts.
- **coins**: computes the list of ways in which a set of coins can be combined to make an amount of money.
- **gray**: a ray-tracer with an interpretive mini-language for specifying the scene. Only the rendering part of the computation is parallelised, so we do not expect to achieve full speedup here.
- **mandel**: a mandelbrot set renderer.
- **matmult**: matrix multiplication (unoptimised; using a list-of-lists representation for the matrices).
- **minimax**: a program to find the best move in a game of 4×4

noughts-and-crosses, using alpha-beta searching of the game tree to a depth of 7 moves.

- **nbody**: calculate the forces due to gravity between a collection of bodies in 3-dimensional space.
- **parfib**: the standard **nfib** microbenchmark in which the tree of recursive calls is evaluated in parallel down to a fixed depth, beyond which the calls are evaluated sequentially.
- **partree**: build a tree in which each node contains an expensive computation, and evaluate it in parallel.
- **prsa**: perform an RSA encoding in parallel.
- **queens**: calculate the number of solutions to the N-queens problem for 14 queens on a 14x14 board.

Program	Allocated (MB)	Rate (MB/s) 24-core	Heap size (MB)
blackscholes	4014	919	700
coins	2509	2669	500
gray	1937	1655	48
mandel	6510	3720	64
matmult	95	61	90
minimax	28465	6859	64
nbody	11777	12267	48
parfib	287	251	32
partree	2106	2106	512
prsa	2754	2899	32
queens	1794	1602	128
ray	6721	2721	32
sumeuler	4642	3439	32
transclos	4304	5448	32

Table 2. Memory usage of benchmark programs

- **ray**: a basic ray-tracer with a very fine granularity (each pixel is a separate spark). This benchmark is included mainly to test how well the system copes with fine-grained parallelism; it is not expected to achieve optimal performance for a ray tracer.
- **sumeuler**: compute the sum of the value of Euler’s function applied to each integer up to a given bound.
- **transclos**: computes the transitive closure of a relation over an initial set of values.

The programs vary in size with the smallest being 18 lines of non-comment code (**parfib**) and the largest 1738 lines (**gray**); most are around 100 non-comment lines.

In two cases (**blackscholes** and **nbody**) the benchmark code is taken from the suite of examples that comes with the Haskell CnC distribution⁶, and adapted to use the standard Parallel Haskell API instead of Haskell CnC. The code differences are minimal, but our versions of the benchmarks perform slightly better than the Haskell CnC originals⁷. In the case of **nbody**, we deliberately de-optimised the program because in its fully optimised form it does no allocation in its inner loop and hence virtually no GC, which made it a poor benchmark for our purposes (we already have a benchmark like this: **parfib**). To de-optimise the program we avoided using some specialised versions of overloaded numerical functions in the inner loop, which lead to some temporary allocation being required, which in turn exercises the young-generation GC.

Table 2 summarises the memory requirements of these benchmarks. These figures reflect the memory requirements on a 32-bit platform; requirements when compiled for a 64-bit platform are approximately double.

The first column shows the total amount of memory allocated over the benchmark run; these results hardly change when running in parallel, so we give only the 1-core figures. Some of our benchmarks allocate relatively little (**parfib**, **matmult**, **queens**), while others allocate over 10GB during the run (**mandel**, **minimax**, **ray**).

The second column of Table 2 gives the allocation *rate* that the benchmarks achieve, using our local heap collector on 24 cores. The main memory write bandwidth on this machine for sequential

writes is approximately 1.5GB/s, and yet we see that many of these benchmarks are exceeding that, in one case by a factor of 8 (nbody). This indicates that our collector is successfully avoiding

⁶<http://hackage.haskell.org/package/haskell-cnc>

⁷ the Haskell CnC versions of these algorithms are competitive with the C++ implementations

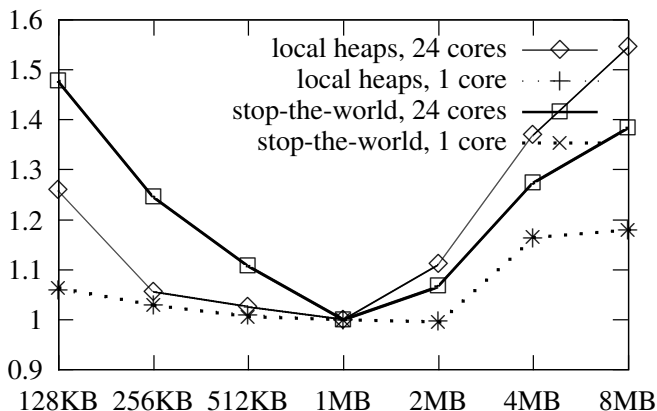


Figure 2. Comparison of nursery sizes

the “allocation wall” of the main memory bandwidth by making effective use of the caches.

The final column shows the heap size we used for each benchmark. Our collector normally runs with a variable heap size, but for the purposes of obtaining a like-for-like comparison we ran both collectors with a fixed heap size. The heap size in each case was chosen to be approximately 3-4 times the maximum residency, plus additional space for 24 nurseries at 1MB (24MB). This gives enough space for each program not to encounter slowdowns due to memory starvation and excessive collection of the global heap. In practice the memory requirements of our local heap collector are very similar to those of our baseline stop-the-world collector.

Many of these programs use lazy streams to run in constant heap space, while generating large output files, and hence do not have large residencies. So for the most part our measurements are not significantly dependent on the performance of the global GC; the one notable exception being `coins` which generates a large list of results in memory (we included this benchmark deliberately because it had a large residency). The use of parallelism does cause an increase in residency, but in most cases it is not significant, and in one case (`coins`) the residency is actually decreased; we believe this is merely an accident due to the timing of global collections.

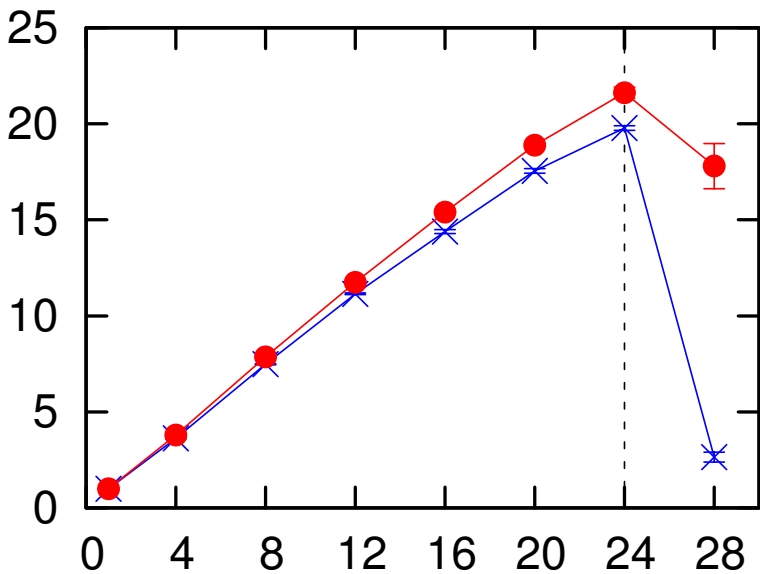
6.1.1 Choosing the nursery size

In order to determine the fixed nursery size that would give the best performance on average for our benchmarks, we measured the relative performance of the benchmark suite with different nursery sizes (Figure 2). We plotted geometric means of normalised run-times across all benchmarks, for combinations of stop-the-world and local heaps with 1 and 24 cores. For each configuration we arbitrarily normalised against the 1MB result set.

Note that these results are averaged over all benchmarks, and as such should not be considered to be representative of the behaviour for any one benchmark. Nevertheless, we do find that on average there is a local minimum around 1MB on this hardware. The dip is more pronounced when running on multiple cores, as we might expect: staying within the cache becomes more beneficial as contention for main memory increases. Interestingly, the results for the local heap collector show that it favours slightly smaller nurseries than the stop-the-world collector, indicating perhaps that it is able to benefit from the greater locality.

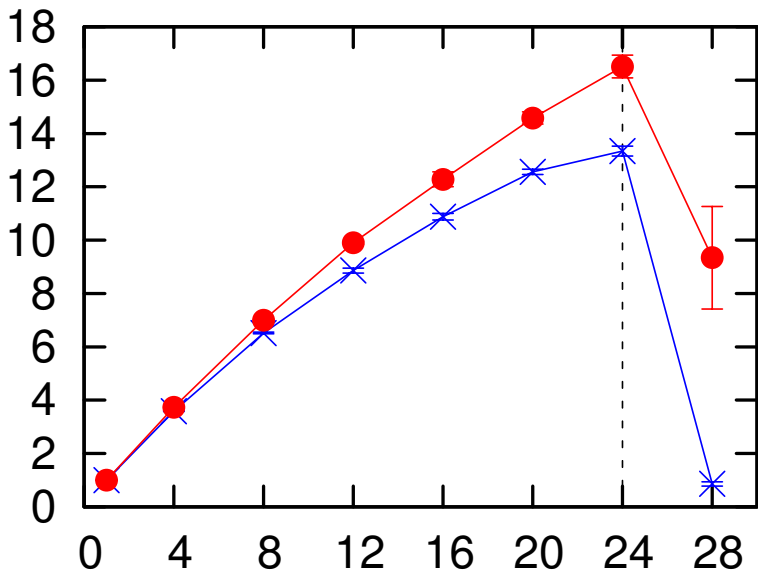
Our results contradict those of Zhao et al. (2009), which found no local maximum in the performance of different young-generation sizes for multithreaded Java programs. Accounting for this discrepancy is beyond the scope of this paper, but we conjecture that it may be due to a different lifetime distribution.

24 cores



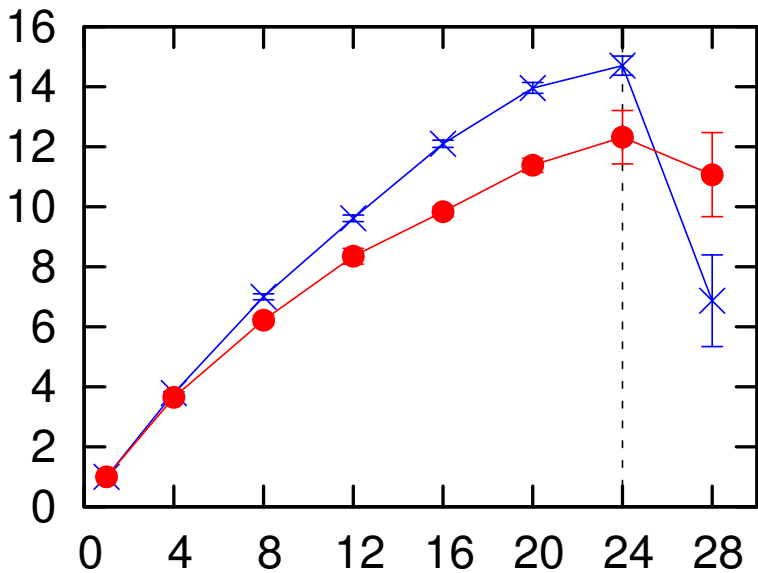
blacksholes

24 cores



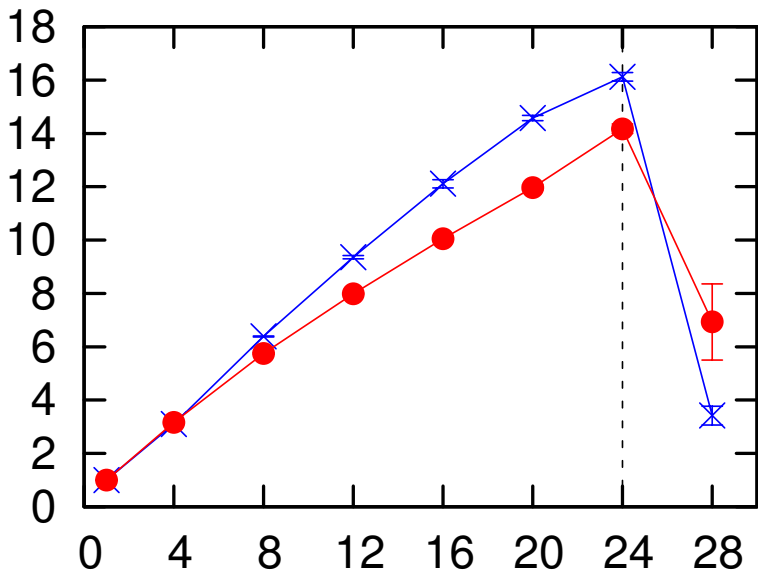
mandel

24 cores



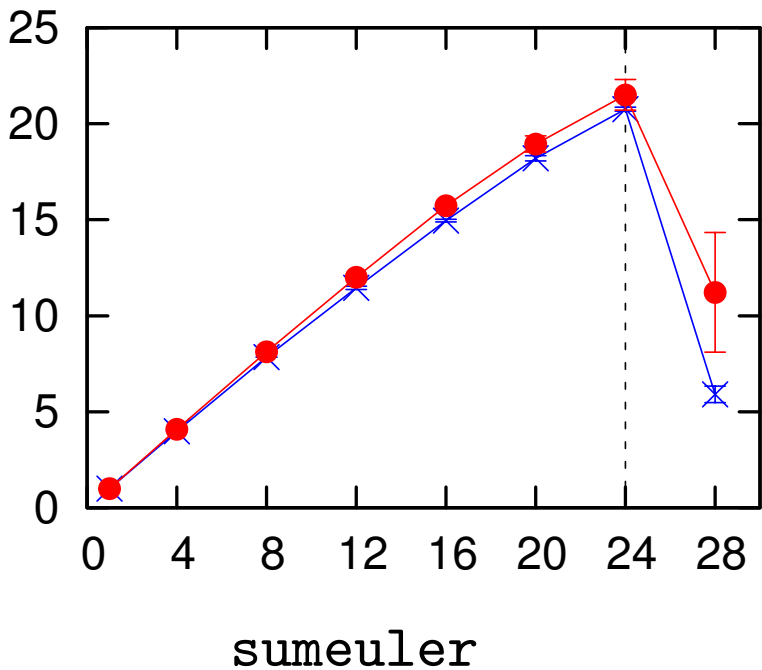
nbody

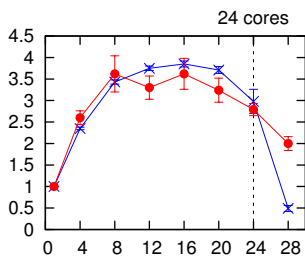
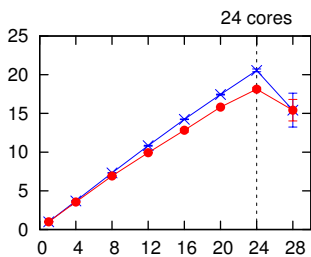
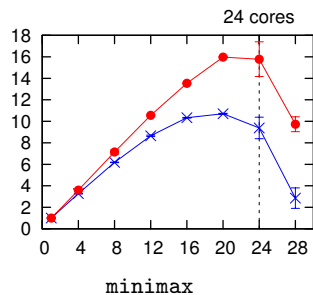
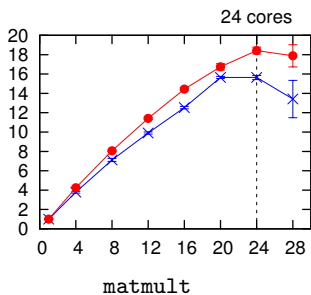
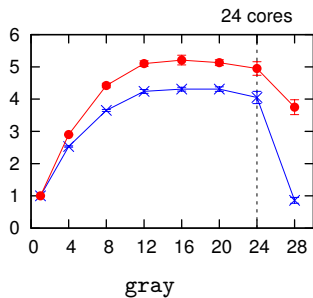
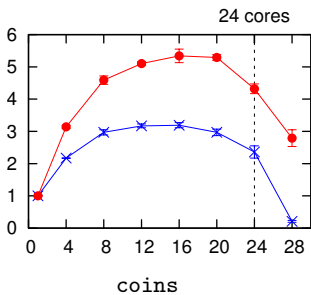
24 cores



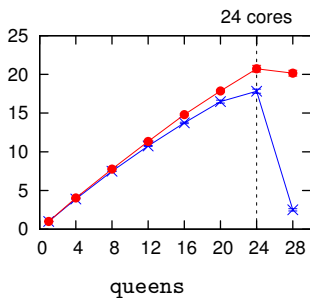
prsa

24 cores

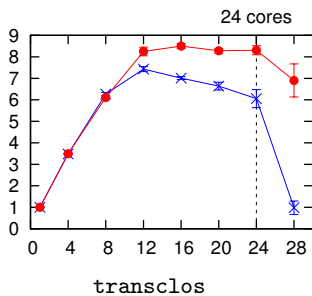
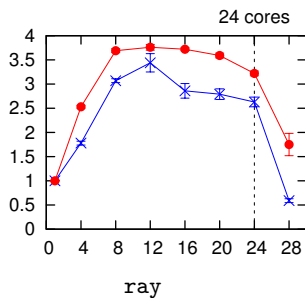






parfib



partree



stop-the-world 
local heaps 

Key

Figure 3. Speedup results on 24 cores (Y axis is speedup, X axis is number of OS threads)

6.2 Scaling

Figure 3 shows the scaling results for our benchmarks on the 24-core 32-bit hardware, comparing GHC’s existing stop-the-world parallel collector against the new local heap implementation. The 1-core baseline in all cases was the stock GHC compiling for single-threaded execution. We measured the wall-clock elapsed time to run each benchmark, averaged over 5 runs. Error bars are shown at one standard deviation.

Note that although our machine has 24 cores, we took measurements up to 28 OS threads. Normally the runtime system would be configured to use no more OS threads than there are hardware threads, but here we wanted to simulate the behaviour of the system when hardware resources are being shared with other processes on the machine (we discuss the 28-core results below in Section 6.2.3).

6.2.1 Analysis of scaling results

At 24 cores, the local heap collector delivers better performance in 10 out of 14 benchmarks. The median improvement across all benchmarks was 15%.

We expect performance differences between the two collectors to be attributable to some combination of the following effects:

1. **Synchronisation.** The local heap collector does not have to synchronise all processors to perform a local collection.
2. **Locality.** Although the stop-the-world collector achieves good locality by not load-balancing during young-generation collections (Marlow et al. 2009), there may be benefits to managing

explicit processor-local heaps.

3. **Single-threaded copying.** Local GC can use a single-threaded algorithm with no atomic memory instructions, unlike the parallel collector which must use atomic instructions to avoid duplicating objects. However, our base parallel collector already forgoes atomic copying for immutable objects, so we expect the difference here to be small.
4. **Fewer young-generation collections.** A local GC is only performed when the local nursery is full, whereas the stop-the-world collector performs a young-generation collection whenever *any* nursery is full. The latter policy could cause premature promotion and extra copying, particularly if different processors are allocating at different rates.
5. **Sharing requires copying.** The local heap collector is required to copy data into the global heap whenever it is shared between processors, and to maintain the global-heap invariant. This will cause the local-heap collector to copy more data in general.

We collected additional metrics to provide some insight into the extent to which each of the above factors affect the results. Unfortunately it is not possible to measure the proportion of the wall-clock elapsed time of the benchmark spent in local heap collections, because local collections are overlapped with mutator activity. Moreover, the local heap collector performs globalisation on demand, so mutator activity is interleaved with GC at a fine granularity. However, the metrics that we can measure are summarised below.

6.2.1.1 Number of collections. We used the same fixed nursery size in both collectors (1MB), and a fixed total heap size. Table 3

shows the number of young-generation collections performed by each collector at 1 and 24 cores, with the 1-core stop-the-world results as the baseline.

At 1-core the local-heap collector performs almost exactly the same number of young-generation collections as the stop-the-world collector, as we expect; this is a good sanity-check that the local-heap collector has similar allocation behaviour.

The stop-the-world collector performs 87% more young-generation collections than it does at 1 core, due to effect (4) above.

Program	stop-the-world		local heaps	
	1 core	24 cores	1 core	24 cores
blackscholes	3758	+78.8	+0.0	+0.3
coins	2506	+64.7	+0.0	+0.5
gray	1950	+347.7	+0.0	-0.7
mandel	6519	+108.0	+0.0	+0.7
matmult	96	+1100.0	+0.0	+18.8
minimax	28499	-0.6	-0.0	-2.8
nbody	11789	+9.0	+0.0	-0.1
parfib	281	+17.8	+0.0	-67.3
partree	2033	+44.2	+0.0	-0.2
prsa	2789	+30.1	+0.1	-15.0
queens	1800	+22.4	+0.0	+8.3
ray	6741	+136.1	-0.0	-6.8
sumeuler	4660	+15.7	+0.0	-0.8
transclos	4284	+159.3	+0.0	+0.8
Min		-0.6	-0.0	-67.3
Max		+1100.0	+0.1	+18.8
Geometric Mean		+87.9	+0.0	-7.7

Table 3. Number of young-generation collections

Program	stop-the-world	local heaps
	24 cores	24 cores
blackscholes	0.16	-6.4
coins	0.15	+36.5
gray	1.61	-8.8
mandel	0.92	-31.8
matmult	0.22	+25.0
minimax	3.38	+126.6
nbody	0.28	+121.8
parfib	0.00	+0.0
partree	2.59	+126.5
prsa	0.94	+90.4
queens	0.52	-80.9
ray	0.55	+166.7
sumeuler	0.07	+136.4
transclos	0.36	-54.9
Min		-80.9
Max		+166.7
Geometric Mean		+19.2

Table 4. Wall-clock time spent in old-generation collections

The local-heap collector, however, actually performed 8% *fewer* on average. The reduction is because a global collection collects the

local heaps too, so depending on the timing of global collections we may need fewer local collections.

The number of old-generation collections is too small to gain any useful insight from (single figures in most cases). Table 4 shows the difference in wall-clock time spent in old-generation collections between the stop-the-world and local-heap collectors. The time spent in old-generation collections increased by 19% on average with the local-heap collector, although the variability across the benchmark set was very high and this may not be a robust effect.

6.2.1.2 *Amount of data copied by GC* The amount of data copied by GC is a reasonable proxy for the cost of GC, and hence GC time. We expect there to be less copying due to there being fewer young-generation collections, but balanced against that is the need to do more copying to globalise data that is shared between processors, and to maintain the global-heap invariant.

Program	stop-the-world		local heaps	
	1 core	24 cores	1 core	24 cores
blackscholes	1997188	+85.7	+9.1	+48.6
coins			402911340	
gray			53900792	
mandel			50358496	
matmult			32577184	
minimax			2288432768	
nbody			2389432	
parfib			58752	
partree			301763736	
prsa			26268076	
queens			53848299	
ray			145387572	
sumeuler			2077124	

transclos		9043744
+0.2	+0.0	+0.1
+38.0	+0.0	+31.8
-25.8	+0.3	-25.3
-28.1	+0.1	+1.4
-19.6	+0.3	-9.2
+701.1	+10.7	+1694.8
+1046.4	-0.4	+378602.4
+6.4	+0.1	+17.6
+5.9	+0.8	+52.4
+11.8	+24.0	+28.3
-57.7	+16.7	+21.5
+1.3	+0.1	+209.4
+23.9	+0.9	+3.2

Min	-57.7	-0.4	-25.3
Max	+1046.4	+24.0	+378602.4

Table 5. Total data copied by GC (bytes)

Table 5 shows the total amount of data copied by each collector (including globalisation in the local-heap collector) at 1 and 24 cores, with the 1-core stop-the-world results as the baseline.

The amount of data copied increased in the local-heap collector compared with stop-the-world. Discounting two outliers: *parfib*, where the amount of copying increased by 32,000%, and *sumeuler*, where the amount of copying was small but highly variable, the average over the rest of the benchmarks was a **23% increase** (ranging from -20% to +187%).

6.2.1.3 Summary To summarise these results, it seems that although the local heap collector performs many fewer young-generation collections and fewer synchronisations, the benefits are offset to some extent by the extra work being done by the collector to maintain the global-heap invariant. These results are by no means conclusive; in future work we plan to gain further insights into the performance differences between the two collectors by measuring additional metrics. For example, measuring the time spent at synchronisation barriers would give some insight into effect (1), measuring cache misses would quantify effect (2), and effect (3) could be measured by turning on atomic copying in the local-heap collector.

6.2.2 Individual benchmarks

Some benchmarks exhibit worse performance with the local-heap

collector. We examined these with our ThreadScope profiling tool, and found:

- `nbody` incurs a message-passing overhead when the main thread requests results from the other processors. In contrast, the stop-the-world collector just shares the data directly.
- `partree` develops some threads with deep stacks. Migrating a thread from one processor to another in the local-heap collector takes time proportional to the depth of the stack, because the thread's local data must first be globalised, whereas in the stop-the-world collector migration was a constant-time operation. The scheduler's load-balancing heuristics are currently not sophisticated enough to avoid expensive migration. We can improve performance on `partree` by disabling automatic migration, but we felt it was important to highlight the issue and raise it as a topic for future work.
- `prsa` appears to be slower with the local-heap collector due to imperfect load-balancing, perhaps related to communication overhead as with `nbody`.
- `parfib` incurs overhead in the local heap collector due to the requirement to globalise data when creating sparks in the spark pool.

6.2.3 Robustness to processor unavailability

In a shared computing environment such as a modern desktop OS, it is highly unlikely that a program will have uninterrupted access to all the processors cores during its runtime. A parallel program should degrade gracefully when the OS deschedules one or more

of its threads so as to run other processes.

With a stop-the-world collector, the cost of synchronisation imposes a severe performance penalty when one or more of the HECs is descheduled by the OS, because when all the other HECs want to garbage collect they stall until the sleeping HEC is reawakened by the OS. We expect that processor-independent GC should ameliorate this effect considerably, by reducing the frequency of synchronisation. We modelled this by running 28 HECs on 24 hardware cores, where it is certain that four will be descheduled at any one moment.

Our results in Figure 3 show that in all cases our local-heap collector outperforms the stop-the-world collector when using 28 threads on the 24-core machine. The median improvement was **73%**.

All benchmarks incur some dropoff in performance with 28 HECs even with the local heap collector, and we believe this is accounted for largely by cache and OS scheduling effects.

6.3 Globalisation policies

A key decision is the globalisation policy: when globalising a pointer, how much of the transitive closure should we globalise? There are a range of possibilities, including:

1. Globalise nothing; just create a new proxy indirection for each write (optimising the case where the pointer written is to a global object).
2. Globalise recursively, but do not globalise mutable objects - leave proxy indirections instead.

Program	Elapsed time ($\Delta\%$)	
	globalise nothing	globalise transitive
blackscholes	-0.8	+0.1
coins	-1.6	-1.0
gray	+2.0	+415.9
mandel	-0.9	+158.3
matmult	+5.1	+2.3
minimax	-0.2	+93.9
nbody	-0.4	-6.8
parfib	-1.8	-0.4
partree	+34.4	+127.6
prsa	+0.3	+536.7
queens	+0.3	+10.7
ray	+21.1	+283.1
sumeuler	-0.2	-4.8
transclos	+4.9	+576.9
Min	-1.8	-6.8
Max	+34.4	+576.9
Geometric Mean	+4.0	+92.6

Table 6. Comparison of write barrier promotion policies on 8 cores

- Globalise recursively, up to a maximum amount of data. Since globalisation is naturally breadth-first (as with copying GC), applying a cut-off to the amount of data copied is similar to a depth-bound. Once the limit has been reached, new proxies would be created for any remaining unglobalised references.

(To date we have not implemented this strategy, but mention it here for completeness.)

4. Globalise the full transitive closure.

We can make an independent choice for different write barriers; for example, updating a thunk could use a different policy than writes to a mutable array.

The tradeoff is not straightforward. From our measurements in Section 3 we know that promoting the full transitive closure incurs a significant overhead for single-threaded programs. However, in a parallel program there is a countervailing effect: when data is needed by multiple processors, it is important that it is moved to the global heap quickly, to avoid latency and excessive communication.

6.3.1 Thunk updates

We measured the effect of modifying the globalisation policy for thunk updates at 8 cores; Table 6 gives the results. To summarise the table:

- Globalising only immutable data was the best (the baseline in the table).
- Globalising nothing at all was on average 4% slower.
- Globalising the entire transitive closure was on average 93% slower, and results ranged from 7% faster to 577% slower.

These results mirror those that we found for single-threaded programs in Section 3, but we find that the negative effect of promoting mutable objects is more extreme when running in parallel.

6.3.2 Spark pool writes

Recall from Section 4.5 that the spark pool (our parallel work queue) is a global heap object and is subject to a globalisation policy in the same way as other writes to the global heap.

We measured the difference between the three different globalisation policies for spark pool writes, and to our surprise there was very little difference. Globalising the full transitive closure was very slightly better on average, but the difference was less than 2% and hence difficult to measure accurately.

Why should there be so little difference, while we see a significant difference in the policies for thunk updates? We believe this is due to several reasons:

- spark pool writes are much less frequent than thunk updates;
- sparks are by their nature unevaluated computations, so globalising thunks referred to by a spark is likely to be the right choice;
- spark pool writes tend to be more persistent: the spark pool entry will typically remain live until it is evaluated; the only way a spark pool entry could become garbage is if the spark was speculative, and our benchmarks here do not use speculation for the most part (`minimax` has a little);
- reducing communication latency is worthwhile, so globalising early is a good idea.

7. Conclusion

A garbage collector with local per-processor heaps can outperform

a stop-the-world parallel garbage collector in raw parallel throughput, and exhibits more robust performance through having fewer all-core synchronisations. Our collector performs best with a small (1MB) nursery size, and does not suffer from the “allocation wall” imposed by main-memory bandwidth.

Our scaling results are not as dramatic as we had hoped when embarking on this line of research, and if we consider parallel throughput alone, it is not clear whether the improvements are worth the (substantial) increase in complexity imposed by the local-heap collector over a stop-the-world implementation. However, the reduction in synchronisation frequency leads to a more significant improvement when the machine is being shared with other processes. Furthermore, although we have not measured pause times here, we believe that the local heap collector together with an incremental or concurrent old-generation collector could be an effective way to control pause times.

Throughout the paper we have identified the aspects of the design that are specific to our particular setting, although it remains unclear whether similar results could be obtained in, say, Java. We firmly believe that a read barrier is necessary for local heap collection in a mutation-rich environment. The sticky-heap aspect of our design is strictly speaking optional, but allows the read barrier to be omitted for some objects, in exchange for not moving them between global collections. A Java implementation could choose to use a read barrier consistently and no sticky heap, or it could identify a class of objects (e.g. arrays) that it is not important to move between global collections and would benefit from having no read barrier.

In future work, we would like to generalise the heap structure to allow multiple generations both local and global.

Acknowledgements

We would like to thank John Reppy and Mike Rainey for several useful discussions about the GC architectures of Manticore and GHC, and we thank the anonymous reviewers of earlier versions of this paper for many helpful insights.

References

- Todd A. Anderson. Optimizations in a private nursery-based garbage collector. In *ISMM '10: Proceedings of the 2010 international symposium on Memory management*, pages 21–30. ACM, 2010.
- David F. Bacon, Perry Cheng, and V. T. Rajan. A real-time garbage collector with low overhead and consistent utilization. In *Proceedings of the 30th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '03, pages 285–298, 2003.
- Lars Bergstrom, Mike Rainey, John Reppy, Adam Shaw, and Matthew Fluet. Lazy tree splitting. In *Proceedings of the 15th ACM SIGPLAN international conference on Functional programming*, ICFP '10, pages 93–104, 2010.
- Stephen M. Blackburn and Kathryn S. McKinley. Immix: a mark-region garbage collector with space efficiency, fast collection, and mutator performance. In *Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '08, pages 22–32, 2008.
- Stephen M. Blackburn, Perry Cheng, and Kathryn S. McKinley. Myths and realities: the performance impact of garbage collection. In *Proceedings of the joint international conference on Measurement and modeling of computer systems*, SIGMETRICS '04/Performance '04, pages 25–36. ACM, 2004.
- 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, 1993.
- Tamar Domani, Gal Goldshtein, Elliot K. Kolodner, Ethan Lewis, Erez Pe-trank, and Dafna Sheinwald. Thread-local heaps for java. In *ISMM '02: Proceedings of the 3rd international symposium on Memory manage-ment*, pages 76–87. ACM, 2002.
- Matthew Fluet, Mike Rainey, John Reppy, and Adam Shaw. Implicitly-threaded parallelism in manticore. In *Proceeding of the 13th ACM SIGPLAN international conference on Functional programming*, ICFP '08, pages 119–130, 2008.
- Richard Jones and Andy C. King. A fast analysis for thread-local garbage collection with dynamic class loading. In *Proceedings of the Fifth IEEE International Workshop on Source Code Analysis and Manipulation*, pages 129–138, Washington, DC, USA, 2005. IEEE Computer Society.
- Simon Marlow, Simon Peyton Jones, and Wolfgang Thaller. Extending the haskell foreign function interface with concurrency. In *Proceedings of the 2004 ACM SIGPLAN workshop on Haskell*, Haskell '04, pages 22–32, 2004.
- Simon Marlow, Alexey Rodriguez Yakushev, and Simon Peyton Jones. Faster laziness using dynamic pointer tagging. In *Proceedings of the 12th ACM SIGPLAN international conference on Functional program-ming*, ICFP '07, pages 277–288, 2007.
- Simon Marlow, Tim Harris, Roshan P. James, and Simon Peyton Jones. Parallel generational-copying garbage collection with a block-structured heap. In *Proceedings of the 7th international symposium on Memory management*, ISMM '08, pages 11–20. ACM, 2008.
- Simon Marlow, Simon Peyton Jones, and Satnam Singh. Runtime sup-port for multicore haskell. In *Proceedings of the 14th ACM SIGPLAN international conference on Functional programming*, ICFP '09, pages 65–78, 2009.
- Pekka P. Pirinen. Barrier techniques for incremental tracing. In *Proceedings of the 1st international symposium on Memory management*, ISMM '98, pages 20–25. ACM, 1998.
- Bjarne Steensgaard. Thread-specific heaps for multi-threaded programs. In *ISMM '00: Proceedings of the 2nd international symposium on Memory*

management, pages 18–24. ACM, 2000.

Yi Zhao, Jin Shi, Kai Zheng, Haichuan Wang, Haibo Lin, and Ling Shao. Allocation wall: a limiting factor of java applications on emerging multi-core platforms. In *Proceeding of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications*, OOPSLA '09, pages 361–376, 2009.