Non-Stop Haskell

A.M. Cheadle & A.J. Field
Imperial College, London
ajf@doc.ic.ac.uk
amc4@doc.ic.ac.uk

S. Marlow & S.L. Peyton Jones
Microsoft Research, Cambridge
simonmmar@microsoft.com
simonmpj@microsoft.com

R.L. While
The University of Western Australia, Perth
lyndon@cs.

Abstract

We describe an efficient technique for incorporating Baker's incremental garbage collection algorithm into the Spineless Tagless G-machine on stock hardware. This algorithm eliminates the stop/go execution associated with bulk copying collection algorithms, allowing the system to place an upper bound on the pauses due to garbage collection. The technique exploits the fact that objects are always accessed by jumping to code rather than being explicitly dereferenced. It works by modifying the entry code-pointer when an object is in the transient state of being evacuated but not scavenged. An attempt to enter it from the mutator causes the object to "self-scavenge" transparently before resetting its entry code pointer. We describe an implementation of the scheme in v4.01 of the Glasgow Haskell Compiler and report performance results obtained by executing a range of applications. These experiments show that the read barrier can be implemented in dynamic dispatching systems such as the STG-machine with very short mutator pause times and with negligible overhead on execution time.

1 Introduction

A significant drawback of automatic garbage collection is the detrimental effect it can have on the responsiveness of a program. While the system is collecting garbage it is unavailable to user programs for useful work, and this can cause a program to appear 'dead' for extended periods of time. This problem is caused largely by the fact that garbage collection algorithms typically collect all of the system's free store in one go, an operation that takes time at least proportional to the number of live objects in the system.

This "unbounded pause" problem has been exacerbated by the tendency towards larger memories and it has discouraged the use of garbage-collected programming languages for many interactive or real-time applications. In the former, for example, the acid test is the responsiveness of the

.uwa.edu.au

system when performing intensive I/O, e.g. when tracking the mouse or performing continuous real-time animation.

An algorithm that avoids this problem, at least in the context of copying collection schemes [6], is Baker's incremental algorithm [3] (an excellent general overview is also given in [15]). Baker's scheme interleaves program execution with small incremental copying phases, causing the program to pause periodically for short periods rather than stopping completely to perform the copy in one go.

The major problem with the implementation of Baker's scheme, at least in software, is the very high cost associated with the *read barrier*, which is a check required at each pointer load to ensure that the referenced object has been copied if it needs to be.

In this paper we present a portable software technique for incremental garbage collection in language implementations that already make use of dynamic dispatch. We make the following contributions:

- We suggest a simple technique that exploits an underlying dynamic-dispatch mechanism to implement the read barrier (Section 3). The idea is that we "hijack" the method table of an object that must be scavenged before being used, thereby implementing a simple per-object read barrier. This means that there are *no* read barrier overheads associated with object references when either i. the garbage collector is off, or ii. the garbage collector is on and the object has already been scavenged.

- We extend the idea to treat stack activation records in a similar way, so that stacks, too, can be collected incrementally (Section 3.2).

- To evaluate our technique we have developed a prototype implementation for the Glasgow Haskell Compiler (Section 4). Experiments with this implementation show that the average overhead on program execution time is less than 4% for the benchmarks tested and that sub-millisecond average pause times are achieved in all cases (Section 5).

Our system is only useful where the bulk of all object accesses use dynamic dispatch – that is, the fields of an object are private, and used only by the object's methods. This is so in GHC's implementation of lazy evaluation, and in some pure object-oriented systems. Certain static optimisations, in which the target of a dispatch can be computed statically, are no longer valid; we cannot quantify this loss in general, but in GHC it is small (Section 4).

The GHC run-time system that we used has a bulk stop-and-copy collection scheme rather than a generational scheme. The scheme we describe can be implemented generationally (see Section 7), but in this paper we focus on the more straightforward bulk copying scheme for evaluation purposes.

2 Background

2.1 Baker's Algorithm

The philosophy behind Baker's algorithm is to spread the overhead of a garbage collection across the set of allocation requests issued by the user program (the *mutator*), in essence performing a series of small collections rather than one major collection.

New objects are allocated in *to-space*. When to-space becomes full, a *flip* takes place: to-space becomes *from-space*, and a few objects (the root set) are *evacuated* (copied) into *to-space*. Each evacuated object must then be *scavenged*, a process that simply evacuates any objects it points to. Objects that have been evacuated into to-space but have not yet been scavenged form what is termed the *collector queue*. When the collector queue is empty, all reachable data has been copied into to-space.

The copying process thus involves two fundamental operations on heap objects:

Evacuation The object is copied from from-space into to-space, leaving a forwarding address in the old copy and returning the address of the new copy. The object is placed on the collector queue, which holds objects in to-space waiting to be scavenged.

Scavenging An object is scavenged by evacuating each child object to which it points, overwriting the existing child pointer with a pointer to the new location of the child.

The garbage collector implements these two operations using layout information that can be derived somehow from the object itself (details vary).

In the incremental version of Baker's collector, scavenging is carried out in piecemeal fashion. When the mutator tries to allocate an object, the collector is given the opportunity to scavenge a few more objects on the collector queue, before returning to the mutator. The number of objects traced at each allocation (the *mark-ratio*) is set to ensure that the garbage collection is completed before the free store is exhausted again. An upper bound can then be placed on the time taken for an allocation request to be serviced.

The *read barrier* maintains the following invariant, the *read-barrier invariant*: *every pointer manipulated by the mutator points into to-space*. If this invariant can be maintained, then newly-allocated objects will contain to-space pointers only and need not be scavenged. Similarly, there is no problem if the mutator writes to an existing object, even if the latter has already been scavenged, because it can only store a to-space pointer. In short, the mutator can operate under the illusion that the garbage collection cycle happens instantaneously.

How does the read barrier maintain the invariant? Whenever the mutator wants to load a field from an object to which it has a pointer, it must check whether the object has been scavenged and, if not, arrange to scavenge it first. The principal disadvantage of the scheme is the CPU overhead of these checks. Three main approaches have been used to implement the read-barrier.

In software This involves inserting extra instructions to perform address range checks at each pointer-load in the program. It carries a typical overhead of 40–50% of the execution time of a program [22].

Using virtual-memory This approach uses the machine's virtual memory protection mechanism to lock down obsolete copies of objects [1]. Any attempt to access such an object causes a trap to the run-time system, the object is traced, the lock is removed and the mutator is resumed. The mechanism is operating system specific and so is not portable. The overhead is very sensitive to the trapping architecture of the system: an average range is around 13–63% [22].

In hardware A small amount of extra hardware on a processor enables pointer-loads to be checked in parallel with normal execution [12]. Again a trap to the run-time system occurs when the mutator trips over the barrier. Typical overheads with this approach are 9–11% [22] although in all but a few customized hardware systems the option is simply not available.

A read-barrier implementation offers a potential advantage to the system [7, 13]. With a read-barrier, objects are traced by two different mechanisms during a garbage collection. "Active" objects are traced by the mutator when it attempts to access them and trips over the barrier: "passive" (though of course live) objects are traced by the garbage collector itself when it is triggered by an allocation request. Thus active objects are naturally distinguished from passive objects and we can improve the dynamic locality of a program by grouping them together. This can reduce the paging costs of a program substantially, and can also improve its cache performance. We do not evaluate this opportunity here.

2.2 The STG-machine

The STG machine [19, 16, 17] is a model for the compilation of lazy functional languages. It is derived from the G-machine [2, 14] and the Spineless G-machine [5].

In the STG-machine every program object is represented as a *closure*. The first field of each closure is a pointer to statically-allocated *entry code*, above which sits an *info table* that contains static information relating to the object's type, notably its layout. An example is shown in Figure 1 for an object with four fields, the first two of which are pointers (pointers always precede non-pointers). The layout information is used by the collector when evacuating or scavenging the closure.

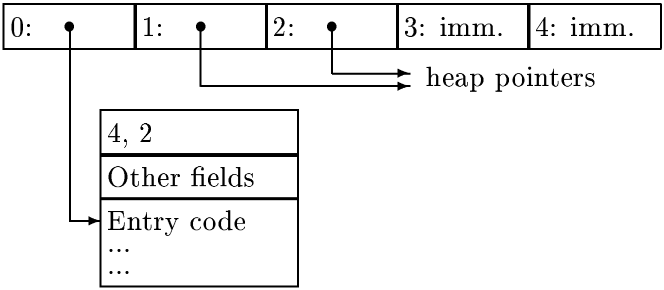

Figure 1: The layout of a closure with four fields, the first two of which are pointers (the other two are immediate values).

Some closures represent *functions*; their entry code is simply the code to execute when the function is called. Others represent *constructors*, such as list cons cells. Their entry code returns immediately with the returned value being the original closure (the closure and the list cell are one and the same). Some closures represent *thunks* (known by some as "suspensions" or "futures"). When the value of a thunk is demanded, the mutator simply *enters* the closure by jumping to its entry-code without performing any test. When the evaluation of the thunk is complete, the thunk is overwritten in-place with a new closure representing its value. If the mutator tries to evaluate the thunk again, execution will now land in the code for the value (i.e. return code), instead of code to compute the value.[1]

The most important feature of the STG-machine for our purposes is that a closure has some control over its own operational behaviour, via its entry code pointer. This representation of heap objects is quite typical in object-oriented systems, except that the header word of an object typically points to a static method table rather than to a single, distinguished method (our "entry code").

3 The New Scheme

In this section we introduce our new scheme. As in the original Baker scheme, each time the mutator claims some space from the heap the garbage collector is called to scavenge k

closures on the collector queue. k is called the *mark-ratio* and it must be large enough to ensure that all of the live closures in from-space are evacuated and scavenged before to-space is filled up. Therefore, at each allocation request, the mutator checks if there is a garbage collection in progress: if there is, it arranges for k closures (on the collector queue) to be scavenged. It must also arrange for the stacks (and other pointers outside the heap) to be scavenged incrementally in similar fashion.

The main idea is a cheap implementation of the read-barrier invariant (Section 2.1), which states that the mutator sees only to-space pointers. The STG machine spends most of its time executing code generated from Haskell function definitions. This code executes in an immediate environment, or *activation* consisting of (a) registers, (b) locations in the function's *stack frame*, and (c) the fields of the *currently-active closure* (CAC). This is quite conventional. The CAC is necessary to support first-class functions and thunks: it contains the environment captured when the function or thunk was allocated. (Object-oriented programmers call the CAC the *this* pointer.) The STG machine captures a separate flat (i.e. non-nested) environment in each such closure, a design decision that turns out to make incremental garbage collection much easier.

Since the mutator only accesses the current activation, we can maintain the read-barrier invariant thus:

[1] Note that some values are bigger than the closures that built them. In these cases the closure is replaced with a pointer to the value (an indirection).

Ensure that all the fields of the current activation have been scavenged; that is, they point into to-space

In operational terms, there are two major elements to implementing this requirement.

- Before entering the code for a closure, we must first scavenge the closure (Section 3.1).
- When a function returns to its caller, we must first scavenge the caller's stack frame (Section 3.2).

3.1 Entering a Closure

The first thing that happens to a closure when it is entered during a garbage collection is that it is evacuated and placed on the queue of closures in to-space waiting to be scavenged. At some subsequent time, the garbage collector scavenges the closure. If the closure is entered between being evacuated and being scavenged, there is a danger of from-space references being propagated into to-space. We avoid this by arranging for the closure to be scavenged if it is entered while it is on the collector queue.

When a closure is evacuated, its info pointer is replaced by a pointer to code that scavenges the closure before entering it. In our prototype implementation the original info pointer is remembered in an extra header word — Word −1 — in the evacuated copy of the closure itself, as illustrated in Figure 2.

Note that there are several ways of avoiding this info pointer copy. We could instead have used part of the original copy of the closure in from-space to store the info pointer, but this would require all objects to have a minimum size of three words, which is not guaranteed in v4.01 of GHC. An alternative is to make each info table contain both entry code and a copy of the self-scavenging code, both with associated layout information. The info-pointer of a closure can then be switched between the entry code and the self-scavenging code transparently from the point of view of the mutator. This is an attractive option but requires a substantial increase in the size of each static info table. From the point of view of a prototype, the info pointer copy also turns out to be useful should a closure be updated before being scavenged: the scavenger can use the original info pointer to determine the size of the original closure and hence how far to skip after the closure has been scavenged (see Section 3.3). So, although maintaining the copy is not ideal in the longer term it does have the short-term advantage of simplifying the prototype implementation. Various improvements to the current prototype, including the removal of the extra word, are discussed in Section 7.

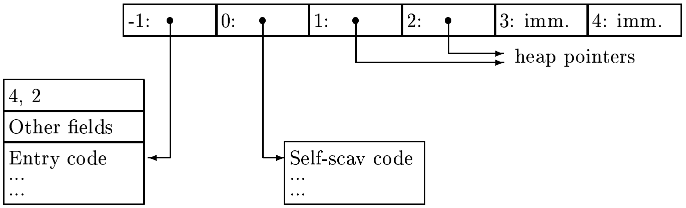

Figure 2: The layout of a closure with two pointers and two immediate values after it has been evacuated under the incremental regime.

After evacuation, there are two possibilities: the closure will either be entered by the mutator or scavenged by the garbage collector.

- If the closure is entered by the mutator first, the mutator executes Self-scav code. This code uses the layout info accessed via Word −1 to scavenge the closure, restores the original info pointer to Word 0, and then enters the closure as it originally expected. The garbage collector will eventually reach the closure, but as Word 0 no longer points to Self-scav code, it knows that the closure has already been scavenged, so it does nothing.

- If the closure is reached by the garbage collector first, the garbage collector scavenges the closure, again using the layout information accessed via Word −1. It also copies the original info pointer back to Word 0, so that if the closure is eventually entered, entry occurs in the normal way.

In effect, the two segments of scavenging code co-operate to guarantee that the closure is scavenged exactly once *before* it is entered.

Note that Word −1 plays no role outside the garbage collection: new objects are allocated in a different region of memory to from-space and do not require the extra word. The space overhead therefore depends on the proportion of closures that are alive at a garbage collection, known as the *residency* of the system. However, this overhead does not require the heap to be any larger. For the same heap size, it means that garbage collection is performed more often; this is automatically factored into the timings.

3.2 Returning to a Closure

The second thing we must ensure is that the active stack frame is completely scavenged. One way to achieve this is to completely scavenge the stack at a GC flip, but this leads to an unbounded pause, especially since GHC supports concurrency, so there may be many stacks.

An alternative is to view the stack as a sequence of closures. The stack grows toward lower addresses, so each return address sits immediately below (in address terms) the stack frame to which it corresponds. In fact GHC makes each return address look exactly like an info pointer, so stack frames really do have the same layout as heap objects.

Returning to the previous stack frame is very like entering a closure, and we could use the same technique to scavenge the stack frame incrementally as we do for closures. However, stack frames are contiguous, so the "Word –1" trick does not work so well. Instead, we adopted a compromise. Interspersed among the regular stack frames are *update frames*, whose role is to update a thunk with the value being returned. These update frames have a fixed return address. The compromise is this: we scavenge all the stack frames between one update frame and the one below, replacing the return adddress in the latter with a self-scavenging return address. This self-scavenging code scavenges the next group of frames, before jumping to the normal, fixed, update code. Since update frames can, in principle, be far apart, pause times can be long; but this case is rare. The basic idea is illustrated in Figure 3.

At the start of a garbage collection, we scavenge each stack down to the topmost update frame, whose return address we replace with a self-scavenging code pointer. After that, the stack is scavenged incrementally, either by the background collector, or when the stack retreats to an un-scavenged update frame. The mutator and the collector maintain a pointer to the highest unscavenged stack frame so that each knows how far the scavenging has progressed. Thus the mutator and the collector co-operate in the scavenging of the stack, without the possibility of the collector encountering a frame that has already been scavenged by the mutator.

3.3 Updates

When garbage collection is performed incrementally, it is possible for a thunk to be updated between being evacuated and being reached by the linear scavenging process. This could cause a problem if the updated closure is smaller than the original: when the linear scan eventually encounters the closure and attempts to skip over it to the next closure it will not skip far enough and will interpret the dead space at the end of the previous closure as the start of the next. However, the collector can determine that the closure has already been scavenged because its info pointer no longer points at Self-scav code.

The problem is therefore avoided by using the original layout information which has been saved at Word −1.

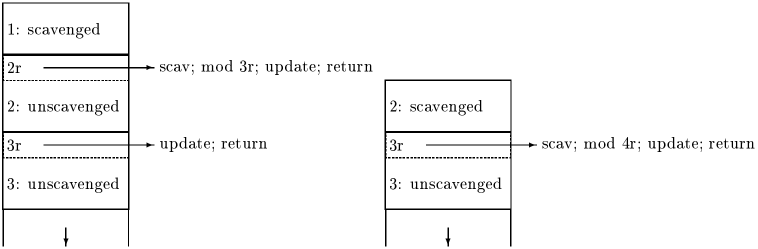

Figure 3: Before and after returning to the caller of group 2. Group 2 has been scavenged and the code-pointer in group 3 has been modified.

4 Implementation

We have constructed a prototype implementation of the scheme on top of GHC v4.01. The run-time system is quite complex so we shall not attempt to give details. Some points are worth noting, however:

- GHC v4.01 uses a two-level storage allocation policy [18]. The central storage manager maintains a free-list of 4KB-blocks of memory. When the mutator encounters an allocation request it obtains a block from the storage manager and chains it onto the heap. The mutator then allocates individual closures within that block until it is exhausted, at which time it obtains a new block. This continues up to some specified maximum number of blocks. The advantages of using a block allocated scheme are that areas of memory used by the system need not be contiguous; blocks that are freed during a collection become available for immediate reuse. Also, new objects are allocated to a separate area (the *nursery*) from those being evacuated, so the scavenger never encounters them.

- The block-allocation scheme makes it easy for the storage manager to vary the size of to-space at run-time.

The manager tries to keep the residency (how much heap is reachable, sampled at the previous garbage collection) at about 33%. It gets more memory from the operating system to achieve this, up to a settable maximum. However, it has a default minimum heap size of 256KB, so for small live data sets the actual residency can be very small.

- Some of GHC's object types are always evaluated at construction, and thus are never evaluated lazily; they are never updated or entered. These are called *unpointed objects*, and include primitive arrays and mutable variables. The lack of entry code in unpointed objects means that we cannot use the self-scavenging scheme to collect them incrementally. They are therefore scavenged immediately after evacuation (*eager* scavenging). This can introduce longer pause times. However, a large object, such as a big array, gets a storage block to itself, so evacuation does not require it to be copied, and hence takes constant time. An alternative would be to implement a Baker-style read-barrier in all the access functions associated with unpointed objects.

- The fact that *all* dispatching must be dynamic in order for the scheme to work means that some compiler optimisations have to be turned off. For example, when entering a dynamic function closure which has a known info table we can jump straight to the entry code rather than indirect via the info pointer in the closure. To exploit our incremental garbage collection scheme these short-cuts must be forbidden for otherwise a scavenging code pointer planted in the object during garbage collection might be missed. This constitutes an additional "cost", although for GHC 4.01 the optimisation actually yields very small improvements (less than 2% on the benchmarks considered below).

The nature of the block allocator in GHC enables the incremental scheme to be supported at two alternative levels. In the traditional Baker scheme, a scavenging phase occurs at each new heap allocation. We can do the same here, although we remark that a single allocation in GHC typically delivers a 'chunk' of memory that can satisfy multiple lower-level allocations. We refer to this as EA (scavenge at Every Allocation). The alternative, which turns out to be beneficial for a number of reasons, is to scavenge at each 4KB block allocation. This increases the grain of each scavenge phase so that the impact of keeping the scavenger going during mutation is reduced. Also, we can tune the block size if necessary in order to reduce, or increase, the pause time. This alternative scheme is called EB (scavenge at Every Block allocation).

5 Experiments

In order to evaluate the new scheme, and to compare it with stop-and-copy and Baker, we have tested it with six applications taken from the nofib test suite [20] and an additional contrived benchmark called `ref` designed to explore some of the extreme cases of heap usage.

We remark that none of the nofib benchmarks requires real-time response. The purpose of the experiments was to get a feel for both the average and extreme performance overheads incurred by the new scheme in relation to stop-and-copy for a representative cross section of applications. The bencharks we selected were chosen arbitrarily with no preconception about the applications' expected behaviour using either scheme. The only criterion was that the application should be able to run to completion without invoking a garbage collection for a suitably chosen set of parameters and initial heap allocation. This was to enable the actual garbage collection and mean pause times to be determined accurately when the garbage collector is turned on.

We ran each application in two configurations: one with a heap large enough for the application to complete without causing a garbage collection (typically around 110Mb), and one with a smaller initial heap (256kB). It is worth noting that the execution time of the application must be controlled such that the total allocation does not exceed the larger heap value (which would result in a garbage collection). The value is constrained to just below the amount of memory in the benchmark machine. With profiling turned off the difference between the two execution times tells us the total cost of garbage collection. For the new scheme the application was first executed with profiling turned on[2] in order to determine the total number of mutator pauses and the heap allocation rate in bytes/s. The timings taken from runs without profiling were used to determine the mean pause time. All experiments were run on a Pentium-II/350 with 128MB RAM running RedHat Linux 6.0.

We ran each application with four garbage collection algorithms: stop-and-copy (with all optimisations turned on), a traditional implementation of Baker with a software read-barrier, and the two versions of the new scheme, EA and EB. The Baker implementation was set to scavenge at each block allocation, as per EB. In each case the mark-ratio was 2 for the runs of EA and 2048 for the runs of EB. Note that the smallest initial heap size is, by default, 256KB.

5.1 Overheads

In keeping with the basic Baker algorithm, the current implementation of the allocator tests whether there is a garbage collection in progress in order to decide whether it needs to do some scavenging. This involves testing a run-time flag called `gc_on` each time an area of free space is requested by the mutator. (Note that it is possible to avoid this test by arranging for the garbage collector to be permanently switched *on*, as described in Section 7, but we retain it here to enable a more direct comparison with the traditional Baker scheme.) The `gc_on` test introduces an overhead on each allocation, the cost of which depends on the granularity of the garbage collector, i.e. how often it performs the test.

There are three time overheads on garbage collection itself: (i) the cost of adjusting the info pointer of each closure during evacuation and scavenging (ii) the cost of adjusting the code pointer in each stack frame that is active during the garbage collection, and (iii) the cost of counting closures as

[2] We found that profiling code significantly affected the execution times in the incremental scheme.

```
contrived
  = if length x == 0
    then [0]
    else if f [ 1..run ] == 0
         then x
         else []
  where
    run  = ...
    work = ...
    copy = ...
    x = take copy [ 1.. ]
    f []         = 1
    f ( x : xs ) = if g x work
                   then 1
                   else f xs
    g a b = if b == 0
            then False
            else g a ( b-1 )
```

Figure 4: The contrived benchmark code

they are scavenged (this overhead can be eliminated if the mark-ratio is hard-coded, e.g. to 1).

6 Results

The performance results reported are based upon the average of a number of identical executions of each application. This was typically between 5 and 10 runs, sufficient at least to keep the 90% confidence interval for the mean execution time within 1% of the mean.

6.1 Contrived Benchmarks

To begin with it is instructive to discuss the results for a contrived benchmark called `ref` as we can control three of the key parameters which affect application behaviour. The code for `ref` is shown in Figure 4. It contains a top-level function which first computes a list `x` of length `copy` and then recurses over a second list of length `run` whilst holding on to the first (by virtue of the final result being dependent on `x`). In each step of the recursion a tail-recursive function that performs no heap allocation is called `work` times. `copy` thus controls the amount of live heap data which has to be copied at each garbage collection; `work` controls the amount of work done between allocations and hence controls the allocation rate (in bytes/s) and `run` controls the length of the run and hence the number of garbage collections that need to be performed. The results reported here were undertaken with the EB scheme, i.e. the self-scavenging scheme with routine scavenging at each block allocation, although similar results are observed with EA.

By setting `work=copy=1` we obtain a program with a very high allocation rate (around 110MB/s) and a very small residency (around 0.1% of the available heap). This means that most of the overhead seen in the new scheme is incurred by the `gc_on` test. We find that the overhead is around 21% in this case. To investigate this further, the benchmark was re-run with a 110MB heap which for `loop` $= 3 \times 10^6$ is sufficient to avoid a garbage collection. We were then able to safely turn off the `gc_on` test in the new scheme. We found

that this reduced the overhead to just over 2%. Indeed, a similar experiment in which the `gc_on` test was redundantly added to the stop/copy scheme with the garbage collector turned on again brought the two execution times to within 2% of one another.

We next increased `copy` in order to increase the heap residency and so measure the effect of managing the info-pointer copies in to-space. The effect is quite subtle: as `copy` increases the allocation rate drops and the residency increases. Proportionally less time is now spent allocating fresh heap space, so the overall effect of the `gc_on` test is reduced. If we set `copy=50,000` the residency remains at around the target value of 33%; the allocation rate drops to around 90MB/s and as a result the overhead reduces to 5.6%. Note that if `copy` is substantially increased the maximum heap size is approached resulting in the residency creeping towards 100%. At this point the extra to-space word has the effect of reducing the time between garbage collections, compared with stop-and-copy, and we find that the overhead swings the other way: as `copy` approaches the maximum value that can be handled by the memory manager the overheads increase by virtue of an increase in the number of garbage collections.

Finally, as `work` is increased, more work is performed between allocations and so the allocation rate is reduced. With `copy=50,000` as above, increasing `work` further reduces the overhead, as is expected. For `work=100` the allocation rate drops to around 16MB/s and the overhead falls to around 1.9%. Note that the residency is unaffected by the value of `work`.

These experiments provide some useful insights into the behaviour of the various collectors in extreme cases and also to the observed performance of some of the `nofib` benchmarks which follow. However, the interactions between the application and the block allocation scheme results in many subtle performance anomalies in both directions: the observations cannot be reliably used as a model of garbage collection performance but they can suggest the correct qualitative trend in many cases.

6.2 Experiments with nofib

The results obtained from our experiments with six of the `nofib` benchmarks are summarised in Figures 1 – 6. Recall that the parameter(s) in each case were chosen to allow the application to complete without garbage collection using a 110MB heap.

The runs of EB for these problem sizes show less than a 6% average overhead relative to the standard stop-and-copy algorithm, but with substantially smaller pause times, typically much less than 1ms. The mean pause times are, on average, approximately 360 times higher for stop-and-copy than for EB. Technically, we are interested in the maximum pause time (rather than the average), but this is too small to make reasonable measurements with the profiling tools that were available. This is discussed in Section 7.

The runs for EA show significantly shorter pause times, of the order of 1μs (approximately 18000 times shorter than stop-and-copy), but with larger overheads (an average of 15%), due primarily to the cost of the `gc_on` test at each allocation.

The highest EB overhead (22%) is for primes 1500. The resident (live) data is quite small in this case and the nursery never exceeds the 256KB minimum allocation. The residency is around 24% and the allocation rate is approximately 40MB/s. In an effort to explain the high overhead we ran `ref` with parameters which matched quite closely the residency and allocation rate of `primes 1500` (`copy`=3000, `work`=25). With these parameters `ref` showed a 21% overhead for EB when compared with stop-and-copy. This is very similar to that observed in `primes 1500`. As the problem size increases the amount of live data and the residency (as a %) both increase. Our earlier experiments with `ref` suggest that the overhead should drop. Indeed, for `primes 6000` (see below) the live data averages around 220KB and the residency increases to around 30% with the measured overhead for EB reduced to just under 2%. By setting `work` and `copy` to approximate this behaviour the measured overhead for `ref` also drops significantly (to around 5%). Qualitatively the relative performance moves in the right direction but we stress that making accurate quantitative predictions in general proves to be very hard.

Remark: In these benchmarks the stop-and-copy collector actually performs very well in terms of mean pause time; the longest mean pause time recorded in these experiments was just 117ms (`pic`). However, it is very easy to construct applications with substantially longer pause times. For example with `work=1, copy=2.5M` in `ref` the mean pause time is around 1.6 seconds.

6.2.1 Longer Runs

To conclude, we executed each of the `nofib` applications with larger problem sizes than could be accommodated in the above experiments. These are summarised in Figure 7 which also shows the arithmetic and geometric means of the overheads of each scheme compared to stop-and-copy. Compared with the earlier experiments we find that the overhead for two of the applications increases slightly with the problem size (`pic` and `circ`), three decrease (`anna`, `primes` and `wave4main`) and one stays approximately the same (`queens`).

We can understand some of these trends from experience with `ref`. For example, an analysis of `circ` shows that with significantly larger problem sizes, the maximum heap size is being reached. This is analogous to the situation earlier when we tried to increase `copy` in `ref` which lead to an increase in overhead. On the other hand, in `wave4main`, for example, the amount of live data increases more slowly and the allocation rate drops compared to the earlier problem size; this accounts for the slight drop in overhead. The results for `anna` are somewhat anomalous for Baker (308% overhead for `preludelist`). The allocation rate for this program is relatively low (an average of just under 3MB/s) with between 5 and 10MB allocated between each garbage collection. Existing heap objects are therefore being accessed at a very high rate relative to the allocation of new ones, which explains the very poor performance of the conventional read barrier implementation. In the new scheme, there is *no* read barrier overhead when the garbage collector is off. This conversely explains the very small overhead seen with both the EA and EB schemes.

	Stop-and-copy	Baker	Self-scav (EA)	Self-scav (EB)
Running time 110Mb (s)	2.03	3.20	2.31	2.14
Running time 256KB (s)	4.26	5.33 (+25%)	4.59 (+8%)	4.19 (-2%)
GC time (s)	2.23	2.13	2.28	2.05
Number of pauses	19	—	1121409	5900
Average pause time (ms)	117	—	0.00203	0.347

Table 1: Results for pic 1500, a particle simulator.

	Stop-and-copy	Baker	Self-scav (EA)	Self-scav (EB)
Running time 110Mb (s)	4.37	7.41	4.59	4.36
Running time 256KB (s)	6.97	12.75 (+83%)	7.86 (+13%)	7.06 (+1%)
GC time (s)	2.60	5.34	3.27	2.70
Number of pauses	213	—	3182084	118556
Average pause time (ms)	12.2	—	0.00103	0.0228

Table 2: Results for anna ap_ListofList, a frontier-based strictness analyser.

	Stop-and-copy	Baker	Self-scav (EA)	Self-scav (EB)
Running time 110Mb (s)	1.28	2.15	1.45	1.27
Running time 256KB (s)	3.89	5.51 (+42%)	4.61 (+19%)	4.16 (+7%)
GC time (s)	2.61	3.36	3.16	2.89
Number of pauses	61	—	1896616	23425
Average pause time (ms)	42.8	—	0.00167	0.123

Table 3: Results for circ 8 125, a simple circuit simulator.

	Stop-and-copy	Baker	Self-scav (EA)	Self-scav (EB)
Running time 110Mb (s)	1.65	2.59	1.66	1.64
Running time 256KB (s)	2.31	3.74 (+62%)	3.08 (+33%)	2.81 (+22%)
GC time (s)	0.66	1.15	1.42	1.17
Number of pauses	353	—	1214107	420491
Average pause time (ms)	1.87	—	0.00117	0.00278

Table 4: Results for primes 1500. primes k returns the k^{th} prime number using the Sieve of Eratosthenes.

	Stop-and-copy	Baker	Self-scav (EA)	Self-scav (EB)
Running time 110Mb (s)	1.94	3.97	2.13	2.02
Running time 256KB (s)	2.09	4.17 (+100%)	2.26 (+8%)	2.17 (+4%)
GC time (s)	0.15	0.20	0.13	0.15
Number of pauses	403	—	21106	403
Average pause time (ms)	0.372	—	0.00616	0.372

Table 5: Results for nqueens 11.

	Stop-and-copy	Baker	Self-scav (EA)	Self-scav (EB)
Running time 110Mb (s)	2.94	4.98	3.10	3.03
Running time 256KB (s)	5.62	8.24 (+47%)	6.12 (+9%)	5.74 (+2%)
GC time (s)	2.68	3.20	3.02	2.71
Number of pauses	139	—	1471723	37554
Average pause time (ms)	19.3	—	0.00205	0.0722

Table 6: Results for wave4main(2000), which predicts the tides in a rectangular estuary of the North Sea.

Application	Param(s)	Stop-and-copy	Baker	Self-scav (EA)	Self-scav (EB)
`pic`	8000	22.69	28.32 (+25%)	26.87 (+18%)	23.92 (+5.4%)
`anna`	preludeList	152.06	468.3 (+308%)	159.9 (+5.1%)	153.7 (+1%)
`circ`	8 500	15.26	22.13 (+33%)	18.60 (+12%)	16.59 (+8.7%)
`primes`	6000	71.58	100.31 (+40%)	79.82 (+12%)	72.89 (+1.8%)
`nqueens`	12	12.47	25.24 (+102%)	13.97 (+12%)	13.00 (+4.2%)
`wave4main`	6000	17.58	25.33 (+44%)	18.95 (+7.8%)	17.66 (+0.4%)
		Arith. mean	+92.0%	+11.2%	+3.6%
		Geom. mean	+59.8%	+10.36%	+2.2%

Table 7: Results for larger runs

7 Discussion and Future Work

The results we have produced are extremely favourable and the experiments we have performed show that the software read barrier can be implemented in systems like the STG-machine at very low cost. The overheads and mean pause times we have reported are significantly lower than for other published schemes. However, we should make it clear that the basis of the scheme is indirection and it is the fact that indirection is already inherent in the STG-machine that the read barrier can be achieved so cheaply. This has to be borne in mind when making direct comparisons.

7.1 Further Improvements

7.1.1 Pause time distribution

Although we have been able to determine accurately the average pause time in each application, we have not been able to determine the distribution because of the resolution of the clock used to profile the code. We propose to access the processor clock for profiling purposes in future implementations.

We expect the pause time to be almost constant in vast majority of cases since the mutator is resumed after a bounded amount of copying has been done. The two aspects of the current scheme for which there is no guaranteed upper bound on pause time is the copying of unpointed objects and stack scavenging.

For unpointed objects the evacuation cost is proportional to the size of the object unless it is a large object; in this case it will reside in its own block and can be evacuated in constant time. The main cost is in the scavenging of unpointed objects: an unpointed object may contain references to other objects, including other unpointed objects. The pause time in such cases may be very hard, or even impossible, to bound. An alternative approach is to implement an explicit read barrier for each unpointed object primitive. It will be interesting then to see whether we can obtain provable upper bounds on the pause time along the lines of [4], for example.

For reasons of simplicity incremental stack scavenging at present operates only between update frames. With some additional effort this can be adapted to work between arbitrary stack frames by hijacking *all* return addresses. This requires separate self-scavenging info tables to be available for all types of stack frame. Alternatively, we can build a generic self-scavenging info table provided we retain an additional copy of the return address in each stack frame. This is analogous to what happens in the current treatment of heap objects where we retain a copy of the old info table pointer in evacuated objects. These issues are currently being explored.

7.1.2 Continuous collection

As discussed earlier, each allocation in the current prototype of both the Baker and new schemes performs a `gc_on` test to determine whether a garbage collection is in progress. The overhead is most noticeable in programs with very high allocation rates and low residencies. The overhead of this test can be eliminated altogether by ensuring that the garbage collector is permanently switched on. This requires some care in "booting" the memory manager to avoid continuous flipping of the two heap spaces as the heap expands in the early stages of mutator execution. Currently, garbage collection terminates when there is nothing left to scavenge. Rather than flipping at this point the termination test can be modified so as to check additionally that a specified minimum number of blocks have been allocated before effecting the flip. Fine tuning of this extra parameter requires experimentation which we plan to undertake in the near future.

7.1.3 Space overheads

In the current implementation of the scheme an additional copy of the original info pointer is maintained when evacuating an object. This simplifies some aspects of the implementation but it carries a (transient) space overhead on all evacuated objects which can affect garbage collector performance, both in maintaining the extra copy and invoking more frequent garbage collections when the heap approaches saturation. With some additional effort in compilation this additional word can be eliminated, by changing the layout of the existing info tables and adding extra fields to store the entry code and self-scavenging code addresses. A separate self-scavenging info table is also required for each closure type. This increases the total space required to store the info tables but the overhead is static. An evaluation of the increase in compiled program size which results will therefore need to be undertaken.

With this modification in place, closure updates become more complicated, specifically if an object in to-space is updated with a value which is smaller than the original object. This can be overcome by adding a dummy info table pointer in the first word of the dead space at the end of the new object which appears to the scavenger to be an object of the

same size as the dead space, but containing no pointer fields. Its effect is to cause the scavenger to skip automatically to the next evacuated object in the collector queue when it encounters it. An implementation of this optimized scheme is now under way.

7.2 Generational Schemes

A number of incremental schemes have been developed based on generational storage structures in an attempt to avoid the read barrier. In a generational scheme, the heap is divided into two or more regions (generations), one holding freshly-allocated, but typically short-lived objects, and the remainder holding older objects copied from previous generations. Because the youngest generation typically contains a small amount of live data the cost of evacuating the data into an older region is usually small. These so-called *minor* collections consequently result in small average pause times.

The main challenge is in the incremental collection of objects in the older generations. As an example, the replicating collector of Nettles *et al* [21] allows the mutator to access from-space whilst garbage collection is in progress. Correctness is maintained by patching the mutator roots at the end of a garbage collection and maintaining a mutation log of all changes to from-space objects from the mutator. The scheme yields a slowdown of less than 20% for the benchmarks tested and less than 10% when restricted to major collections. A similar approach is used in [11] for ML which allows access to immutable objects in from-space.

Incremental schemes have also been developed for multi-threaded systems, such as [10] which is targeted at Concurrent Caml. Here immutable objects are allocated in private heaps and are copied to a shared heap during local garbage collection within the associated thread. The shared heap is collected by a separate thread operating Dijkstra's concurrent mark/sweep algrithm [9]. The scheme has the advantage that the threads can perform minor collections independently; however, mutable objects have to be allocated in the shared heap which incurs an additional expense. The average pause time for minor collections is favourable, but there is no guaranteed upper bound: the time is proportional to the amount of live data in the private heaps.

The scheme we have proposed here can be easily adapted to generational schemes and we are planning an implementation for Haskell in the near future. Unlike the existing schemes, however, the fact that the read barrier comes at such low cost means that *all* collections, including minor collections, can, in principal, be performed incrementally. Alternatively, minor collections could be performed on the conventional 'stop-the-world' basis with the incremental scheme being used for the older generations.

7.3 Object-oriented Systems

The dynamic dispatch exploited in the STG machine is something that is shared with almost all dynamic languages. For example, the representation of closures that GHC uses is strikingly similar to that used for objects in an implementation of a pure object oriented language such as Smalltalk. An object points to a (static) method table, which contains the addresses of each method associated with the object. Method invocation is then performed by an indexed jump through the table.

It would appear to be possible to adapt our incremental scavenging technique for an object oriented language, by temporarily replacing the method-table pointer in an object with one that consisted of a vector of self-scavenging methods. However, there is a price to pay: *all* methods would have to be invoked via jump tables, essentially eliminating any performance benefits that might accrue from knowing the exact type of an object, and therefore the exact method code pointer to jump to. (Work on virtual inlining of Java programs suggests that performance improvements of up +26% [8] can be achieved.) Direct field access would also be forbidden; all fields would have to be accessed via method calls. This imposes a significant one-off cost, but it is a cost that may have multiple potential pay-offs. For example, an object system that supports proxies, in which a local proxy stands in for a remote object, or perhaps simply logs calls to a local object, would need a similar restriction. Work is now under way to evaluate the performance trade-offs in the context of object-oriented systems.

8 Summary and Conclusions

We have described a scheme for incorporating incremental garbage collection into the STG-machine on stock hardware. The scheme is based on the manipulation of the code-pointers that are used to implement closure behaviour in the STG-machine. Each closure creates its own, 'personal' read-barrier by manipulating its code-pointer during execution and garbage collection. The technique derives its efficiency from the twin facts that these manipulations are very cheap and that they are performed on a per-closure basis, rather than a per-pointer-load basis, as in most implementations of the read-barrier. Experiments suggest that the new scheme carries a very low overhead, averaging less than 4% in execution time for the benchmarks tested with average pause time being sub-millisecond in all cases. This shows that in systems which implement dynamic dispatching exclusively, such as the STG-machine, a portable read barrier can be implemented extremely cheaply leading to very short mutator pauses with negligible overhead in execution time.

We have also found that the block-based memory allocator of the GHC run-time system provides an excellent level of granularity at which to perform incremental scavenging. By scavenging on each block allocation we can keep the scavenger going for longer, at the expense of an increase in pause time. This substantially reduces the overheads in the proposed scheme and provides an additional mechanism (the block size) by which to control the behaviour of the garbage collector.

References

[1] A. Appel, J. Ellis, and K. Li. Real-time concurrent collection on stock multiprocessors. In *Conference on Programming Language Design and Implementation*, pages 11–20, 1988.

[2] L. Augustsson. *Compiling Lazy Functional Languages, Part II.* PhD thesis, Chalmers University, Sweden, 1987.

[3] H. Baker. List-processing in real-time on a serial computer. In *CACM 21(4)*, pages 280–94, 1978.

[4] G. Blelloch and P. Cheng. One bounding time and space for multiprocessor garbage collection. In *ACM SIGPLAN Symposium on Programming Language Design and Implementation (PLDI)*, 1999.

[5] G. Burn, S. Peyton-Jones, and J. Robson. The spineless g-machine. In *Conference on Lisp and Functional Programming*, pages 244–58, 1988.

[6] C. Cheney. A non-recursive list compacting algorithm.

In *CACM 13(11)*, pages 677–8, 1970.

[7] R. Courts. Improving locality of reference in a garbage-collecting memory management system. In *CACM 31(9)*, pages 1128–38, 1988.

[8] D. Detlefs and O. Agesen. Inlining of virtual methods. In *ECOOP*, 1999.

[9] E. Dijkstra et al. On-the-fly garbage collection: An exercise in cooperation. In *CACM, 21, 11*, pages 966–975, 1978.

[10] D. Doligez and X. Leroy. A concurrent generational garbage collector for a multi-threaded implementation of ML. In *PoPL '93*, pages 113–123, 1993.

[11] L. Huelsbergen and J. Larus. A concurrent copying garbage collector for languages that distinguish (im)mutable data. In *4th ACM Symposium on Principles and Practice of Parallel Programming, Vol. 28(7) of ACM SIGPLAN Notices*, pages 73–82, 1993.

[12] D. Johnson. Trap architectures for lisp systems. In *Conference on Lisp and Functional Programming*, pages 79–86, 1990.

[13] D. Johnson. The case for a read barrier. In *ASPLOS-IV*, pages 279–87, 1991.

[14] T. Johnsson. *Compiling Lazy Functional Languages*. PhD thesis, Chalmers University, Sweden, 1987.

[15] R. Jones and R. Lins. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management.* Wiley, 1996.

[16] S. P. Jones. The Spineless Tagless g-machine: Second attempt. In *Workshop on the Parallel Implementation of Functional Languagesi*, volume CSTR 91-07, pages 147–91. University of Southampton, 1991.

[17] S. P. Jones. Implementing lazy functional languages on stock hardware: the Spineless Tagless g-machine. In *Journal of Functional Programming*, 1992.

[18] S. P. Jones, S. Marlow, and A. Reid. The STG runtime system (revised). draft paper, Microsoft Research Ltd, 1999.

[19] S. P. Jones and J. Salkild. The Spineless Tagless G-machine. In *Conference on Functional Programming Languages and Computer Architecture*, pages 184–201, 1989.

[20] W. Partain. *The nofib Benchmark Suite of Haskell Programs.* Dept. of Computer Science, University of Glasgow, 1993.

[21] e. a. S.M. Nettles. Replication-based incremental copying collection. In *Proceedings of the International Workshop on Memory Management, LNCS 637.* Springer Verlag, 1992.

[22] B. Zorn. *Comparative Performance Evaluation of Garbage Collection Algorithms.* PhD thesis, University

of California at Berkeley, 1989.