Exploring the Barrier to Entry - Incremental Generational Garbage Collection for Haskell

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

S. Marlow & S.L. Peyton Jones
Microsoft Research, Cambridge
{simonmmar,simonpj}@microsoft.com

R.L. While
The University of Western Australia, Perth
lyndon@csse.uwa.edu.au

ABSTRACT

We document the design and implementation of a "production" incremental garbage collector for GHC 6.02. It builds on our earlier work (Non-stop Haskell) that exploited GHC's dynamic dispatch mechanism to hijack object code pointers so that objects in to-space automatically scavenge themselves when the mutator attempts to "enter" them. This paper details various optimisations based on code specialisation that remove the dynamic space, and associated time, overheads that accompanied our earlier scheme. We detail important implementation issues and provide a detailed evaluation of a range of design alternatives in comparison with Non-stop Haskell and GHC's current generational collector. We also show how the same code specialisation techniques can be used to eliminate the write barrier in a generational collector.

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

General Terms: Algorithms, Design, Experimentation, Measurement, Performance, Languages

Keywords: Incremental garbage collection,
Non-stop Haskell

1. INTRODUCTION

Generational garbage collection [24] is a well-established technique for reclaiming heap objects no longer required by a program. Short-lived objects are reclaimed quickly and efficiently, and long-lived objects are promoted to regions of the heap which are subject to relatively infrequent collections. It is therefore able to manage large heap spaces with generally short pause times, these predominantly reflecting the time to perform minor collections.

Eventually, however, the region(s) containing long-lived objects (the "older" generation(s)) will fill up and it will be necessary to perform a so-called *major* collection.

Major collections are typically expensive operations because the older generations are usually much larger than the young one. Furthermore, collecting an old generation requires the collection of all younger generations so, regardless of the actual number of generations, the entire heap will eventually require collection. Thus, although generational collection ensures a relatively small mean pause time, the pause time distribution has a "heavy tail" due to the infrequent, but expensive, major collections. This renders the technique unsuitable for applications that have real-time response requirements, for example certain interactive or real-time control systems.

The traditional way to reduce the variance of the pause times is to perform the garbage collection incrementally: rather than collect the whole heap at once, a small amount of collection is performed periodically. In this way the activities of the executing program (referred to as the *mutator*) and garbage collector are interleaved.

One way to achieve incremental collection in the context of *copying* garbage collectors [8] is to use a *read barrier* to prevent the mutator from accessing live objects that have yet to be copied. This is the basis of Baker's algorithm [2][1].

In our earlier paper [7] we described a low-cost mechanism for supporting the read barrier in the GHC implementation of Haskell with a single-generation heap. This exploits the dynamic dispatch mechanism of GHC and works by intercepting calls to object evaluation code (in GHC this is called the object's *entry* code) in the case where the garbage collector is on and where the object has been copied but not yet scavenged. This 'hijacking' mechanism directs the call to code that automatically scavenges the object ("self-scavenging" code) prior to executing its entry code; this eliminates the need for an explicit read barrier.

In implementing this "barrierless" scheme we had to modify the behaviour of objects copied during garbage collection so that an attempt to enter the object lands in the self-scavenging code. At the same time, we had to devise a mechanism for invoking the object's normal entry code after

ISMM'04, October 24–25, 2004, Vancouver, British Columbia, Canada.

[1] An alternative is to use replication [13] which replaces the read barrier with a write barrier, but at the expense of additional work in maintaining a log of outstanding writes. This paper focuses on Baker's approach, although the replication approach is the subject of current work.

execution of the self-scavenging code. We chose to do this by augmenting copied objects with an extra word which retains a reference to the original object entry code after the entry code pointer has been hijacked. This extra word comes at a price: in addition to the overhead of managing it, it also influences the way memory is used. For example, in a fixed-size heap a space overhead reduces the amount of memory available for new object allocation which can reduce the average time between garbage collections and increase the total number of collections.

An alternative approach is to build, at compile time, *specialised* entry code for each object type that comes in two flavours: one that executes normal object entry code and one that first scavenges the object and then executes its entry code. This eliminates the need for the extra word as we can simply flip from one to the other when the object is copied during garbage collection. The cost is an increase in the size of the static program code (code bloat).

The main objective of this paper is to detail how this code specialisation can be made to work in practice and to evaluate the effect of removing the dynamic space overhead on both the mutator and garbage collector. Although the idea is simple in principle it interacts in various subtle ways with GHC.

Specialisation also facilitates other optimisations, for example the elimination of the write barrier associated with generational collectors. We show how the basic scheme can be extended to do this, although in the end we chose not to implement the extension as the average performance gain is shown to be small in practice. However, for collectors with a more expensive write barrier, or write-intensive applications, one might consider the optimisation worthwhile.

Experimental evaluation shows that a modest increase in static code size (25% over and above stop-and-copy and an additional 15% over our earlier implementation of Non-stop Haskell) buys us an incremental generational collector that increases average execution time by a modest 4.5% and reduces it by 3.5% compared with stop-and-copy and Non-stop Haskell respectively, for our chosen benchmarks. Our solution is a compromise that exploits specialisation to remove the expensive read barrier but retains the write barrier to yield an acceptable overhead in the static code size.

The paper makes the following contributions:

- We describe how to eliminate the read barrier in GHC using object specialisation and compare it to our previous scheme which pays a space overhead on all copied heap objects (Section 3).

- We present performance results for various incremental generational schemes, focusing in particular on execution-time overheads, pause time distribution and mutator progress (Section 4).

- We report and explain unusual pathological behaviour observed in some benchmarks whilst running an incremental collector and show that execution times can sometimes be *reduced* by a substantial factor when compared with stop-and-copy. We believe this is the first time such behaviour has been reported in respect of incremental collection (Section 4.3).

- We describe how to use specialisation to eliminate the write barrier associated with generational collection and compare the approach with that proposed by Rojemo in [16] (Section 5).

2. BACKGROUND

We assume that the reader is familiar with Baker's incremental collection algorithm [2] and the fundamentals of generational garbage collection [24]. However, we review some notation.

In this paper we assume that all collection is performed by copying live objects from a *from-space* to a *to-space*. In Baker's algorithm copying is synonymous with *evacuation*. Evacuated objects are scavenged incrementally and the set of objects that have been evacuated but not scavenged constitute the *collector queue*.

A *read barrier* is required on each object access to ensure that objects in from-space are copied to to-space before the mutator is allowed to access them. A significant component of this barrier code is a test that first determines whether the garbage collector is on (GC-ON) before applying the from-space test.

In generational collection we shall assume in the discussions that follow that there are just two generations: the *young* generation and the *old* generation, although our implementations can be configured to handle an arbitrary number of generations. We assume that objects are aged in accordance with the number of collections they survive. We distinguish object *tenuring*, the process of ageing an object within a generation, from object *promotion*, whereby an object is deemed sufficiently old for it to be relocated to the next oldest generation.

The remainder of this section focuses on the implementation platform that is the subject of this paper.

2.1 GHC and the STG-machine

GHC (the Glasgow Haskell Compiler) implements the Spineless Tagless G-machine (STG) [21, 18, 19], which is a model for the compilation of lazy functional languages.

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 and scavenging the closure.

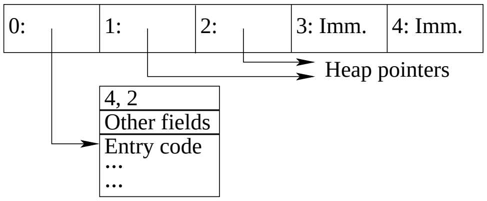


Figure 1: Closure layout of four fields – two pointers followed by two immediate values.

Some closures represent *functions*; their entry code is simply the code to execute when the function is called. Others represent *constructors*, such as list 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 unevaluated expressions; these are known as *thunks* (or in some parlances as "suspensions" or "futures") and are the principal mechanism by which lazy evaluation is supported. When the value of a thunk is required, the mutator simply *enters* the thunk by jumping to its entry-code without testing whether it has already been evaluated. When the thunk has been evaluated it is overwritten with an indirection to its value. If the mutator tries to re-evaluate the thunk, the indirection is entered, landing the mutator in code that returns the value immediately.

Note that there is an inherent delay between a closure being entered for the first time and the closure being updated with its value. In a single-thread model, an attempt to enter a closure that has already been entered but not yet updated results in an infinite loop. To trap this, objects are marked as *black holes* when they are entered.

In GHC 6.02 closures representing function applications actually work in a slightly different way to thunks. Instead of blindly entering the function's entry code an explicit "apply" function first evaluates the function (recall that Haskell is higher-order) and then inspects the result to determine the function's arity. At this point it determines whether an exact call to the function can be made or whether a partial application must be constructed.

This mechanism is different to previous implementations of GHC and has important implications for the garbage collector, as we discuss in Section 3.3. Full details of the "eval/apply" mechanism can be found in [11].

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. We remark that this type of representation of heap objects is quite typical in object-oriented systems, except that the header word of an object typically points to a method *table* rather than to a single, distinguished method (our "entry code").

2.1.1 Stacks and Update Frames

In describing the various implementations it is important to understand the way GHC stacks operate. Each stack *frame* adopts the same layout convention as a closure, with its own info table pointer and entry code. The most frequently occurring frame is the activation record of a function and its associated return address. However, the stack also contains *update frames* which are pushed onto the stack when a closure is entered. The update frame contains a pointer to the closure. When the closure has been evaluated its corresponding update frame will be at the top of the stack. This is popped and the closure is updated with an indirection to its value. Control then passes to the stack frame underneath the update frame on the stack by entering it. All updateable thunks push an update frame as a result of executing their entry code – something we shall exploit later on.

2.1.2 Generational Collection in GHC

Generational garbage collectors have been studied extensively in the context of lazy functional languages [17, 22, 16]; the most recent releases of GHC use a generational garbage collector. The key point is that new references from the old to young generation (by 'new' we mean references that appear after an object is promoted) can come about only when a thunk is updated with its value – recall that there is no explicit assignment in Haskell! Thus, the write barrier is implemented in software by planting test code around each update.

There have been numerous studies on tenuring/promotion strategies, generation sizing and number [1, 10, 23, 9, 26, 27, 17, 16]. The conclusion is that in practice only a few generations and distinct tenure ages (steps) are necessary. GHC defaults to two generations and two tenuring steps, although these can be changed.

2.2 Non-stop Haskell

We presented in [7] an incremental collector for Haskell that cheaply implements the read-barrier invariant: *all pointers accessible by the mutator point to to-space.* It works by arranging for a closure to be scavenged if it is entered while it is on the collector queue by "hijacking" its info pointer: when a closure is evacuated, its info pointer is replaced by a pointer to code that first scavenges the closure before entering it. We referred to this as "self-scavenging" code. 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.

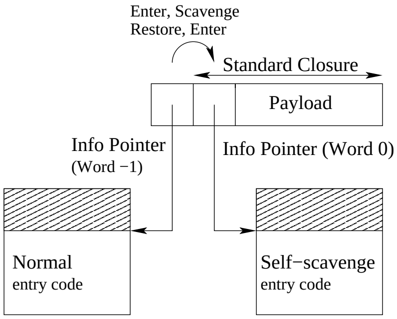


Figure 2: An evacuated closure in Non-stop Haskell.

After evacuation, 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 the self-scavenging 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-scavenging code, it knows that the closure has already been scavenged and so 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 code is executed 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.

The key benefit of modifying the object's behaviour depending on its status, rather than explicitly querying the object's status, is that execution proceeds normally, i.e. with no overhead, when the garbage collector is off.

2.2.1 Invoking the Scavenger

GHC has a block-allocated memory system with a default block size of 4KB. We explored two levels of granularity at which to perform incremental scavenging during garbage collection. The first invokes the scavenger at each *object* allocation – the object allocation code is inlined with additional code which first tests whether the collector is on and then, if it is, invokes the scavenger before allocating space for the object. The second plants the same code in the runtime system at the point where a new memory *block* is allocated. By scavenging at each block allocation the mutator does more work between pauses at the expense of increased mean pause times (the scavenger also has to do more work to keep up). In this paper we again evaluate both options.

2.2.2 Incremental Stack Scavenging

At the start of a garbage collection cycle all objects in the root set are evacuated so that all immediately accessible objects are in to-space (the *read-barrier* or *to-space invariant*). The simplest mechanism that enforces this invariant is to scavenge the entire stack at the start of the collection cycle, but this leads to an unbounded pause. The alternative, is to place a read barrier on stack pop operations, which adds further significant overhead. We devised a cheap means of scavenging the stack incrementally, this time hijacking the return addresses in update frames which are interspersed among the regular frames on the stack. These update frames have a fixed return address. We scavenge all the stack frames between one update frame and the one below, replacing the return address in the latter with a self-scavenging return address. This self-scavenging code scavenges the next group of frames, before jumping to the normal, generic, update code.

Since update frames could, in principle, be far apart, pause times could be long, although this case is rare in practice. Later, we show how moving to the eval/apply model of function application in GHC 6.02 enables the same hijacking trick to be applied between *adjacent* stack frames, rather than between arbitrarily-spaced update frames.

3. OBJECT SPECIALISATION

The principal disadvantage of our Non-stop Haskell collector is that there is a one-word overhead on all objects that survive at least one garbage collection (all objects are either copied to the young generation to-space or are promoted). This extra word must be allocated and managed during garbage collection which carries an overhead in its own right. Furthermore, it can have an indirect effect on the behaviour of the collector. For instance, in a fixed-size heap the overhead reduces the amount of space available for new allocations which can reduce the mean time between garbage collections and hence increase the total number of garbage collections. On the other hand, with a variable heap sizing policy, where the amount of free space is a function of the space occupied by live data, the overhead can work the other way. We discuss the various trade-offs on garbage collection in detail in Section 4. For now we focus on removing the extra word to eliminate the management costs.

3.1 Removing Dynamic Overhead

We now show how the dynamic space overhead can be eliminated by object specialisation. This gives us the opportunity to evaluate the effect that dynamic space overheads have on program execution time in general.

The idea is simple: instead of employing generic code to perform actions like self-scavenging, and using additional space in the heap to remember the object's original entry code, we instead modify the compiler so that for each closure type, in addition to generating its usual info table and entry code, we build one or more "specialised" versions that modify the object's usual behaviour when entered.

For each closure type we generate one additional variant. This inlines generic "self-scavenging" code with a duplicate copy of the standard entry code (but see Section 3.2 below). Info tables are extended with an extra field to contain a reference to their partner, i.e. the standard info table contains a reference to the self-scavenging info table and vice versa. The scheme is identical to the original, but now instead of copying the standard info pointer to Word –1 on evacuation, we simply replace it with its partner's (i.e. self-scavenging) info pointer. This obviates the need for Word –1. On scavenging, we do the reverse, thus restoring the standard info pointer. Notice that the static data that sits above the object's entry code must carry a pointer to the object's partner in order to facilitate this flipping of code pointers.

This new scheme is pictured in Figure 3. This scheme has replaced the runtime space overhead with a slight increase in compilation time and code space; both info table and entry code are stored in the text segment.

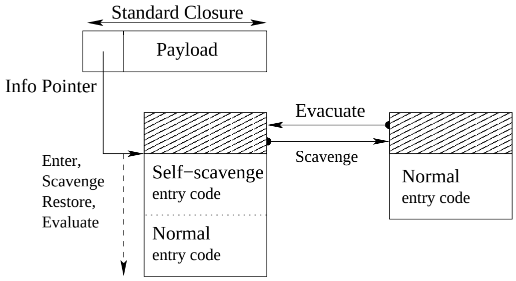


Figure 3: Object Specialisation

3.2 Entry Code Duplication vs Inlining

Note that in the diagram, each partner is shown with an identical *inlined* copy of the object's original entry code. Of course, we could replace one of these by a direct jump to the other, thus reducing the space overhead at the expense of additional instruction execution (JUMP). Although we do not expect much difference in execution time we evaluate both variants of this mechanism in Section 4.

3.3 Implications of Eval/Apply

Recall from Section 2.1 that GHC 6.02 uses an eval/apply model of function application. Because "apply" is a generic operation, it must explicitly inspect the type tag of the closure and its arity. From the point of view of the garbage collector it is at this point that we must ensure the closure has been scavenged, in order to maintain the to-space invariant. The key difference between GHC's previous "push/enter" model, and eval/apply is that in the latter functions are *not* actually "self"-scavenging in the above sense since the scavenging is performed by the apply function based on the object's type tag. For sure, we could make them so and simply arrange for the apply function to enter self-scavenging code, but there is nothing to be gained.

3.4 Stack Scavenging

One of the consequences of the eval/apply model is that the stack now consists solely of stack frames. In the push/enter model the stack is also littered with pending argument sections (arguments pushed in preparation for a function call). Using eval/apply it is therefore possible to "walk" the stack *cheaply* one frame at a time, something that is not possible in the push/enter scheme.

In our implementation of Non-stop Haskell we could not scavenge the stack one frame at a time; the only stack object whose return address we could safely hijack was the next update frame on the stack (refer back to Section 2.2.2). Although updates in GHC are quite common, the amount of work required to scavenge between adjacent update frames (a "unit" of stack scavenging work) was impossible to bound. In the eval/apply scheme the maximum amount of work is determined by the largest allowable stack frame, which is fixed in GHC at 1KB. This provides a hard bound on the time taken to perform one "unit" of stack scavenging work.

Given that the stack really is just like a collection of closures do we need to specialise the entry code for these stack frames in order to make the stack scavenging incremental? No, because the stack is accessed *linearly* from the top down. To hijack the return address of the frame below we just need a single extra word (e.g. a register) to remember the original info pointer of the frame that is the latest to be hijacked. We cannot do the same in the heap as the heap objects are subject to random access.

3.5 Slop Objects

When a closure is updated it is (always) replaced by an indirection to the closure's value. Because the indirection may be smaller than the original closure, the scavenger has to be able to cope with the remaining "slop", i.e. the now unused memory within the original closure. In short, the scavenger must know how to skip this dead space in order to arrive at the next valid object in the collector queue.

Note, that for the stop-and-copy collector this issue of slop does not exist – the scavenger does not scavenge from-space where slop exists. Furthermore, the collection cycle is instantaneous with respect to mutator execution, and so it is not possible for an evacuated object to be updated before it is processed by the scavenger.

If we use the extra word, the scavenger can always determine the size of the original closure since its info table will be accessible from the object's original info pointer at Word –1. With object specialisation, this information is lost: when the scavenger has processed the updated version of the object it has no idea whether there is any dead space following the object, let alone how much. We solve this problem by creating a special "dummy" object – a *slop object* – which looks to the scavenger like a normal object whose size is that of the dead space created by the update and whose payload contains no pointer fields. When the scavenger encounters a slop object the effect is to skip immediately to the next object in the collector queue. This is illustrated in Figure 4.

We do not want to build slop objects on the fly, so to make "slopping" fast for the vast majority of cases we predefine eight special slop objects, one for each slop size from 1 to 8. A generic slop object is constructed dynamically for all other cases whose payload contains the slop size. This is more expensive but it very rarely required (we did not observe it at all in the benchmarks we used for evaluation). Note that slop objects will never be entered – only the layout information is ever used – so the entry code for each is null. Figure 4 shows the situation after the object's update code has overwritten the original object with an indirection and the remaining dead space with a pointer to one of the eight predefined slop objects – the seventh in this case as the dead space comprises seven words.

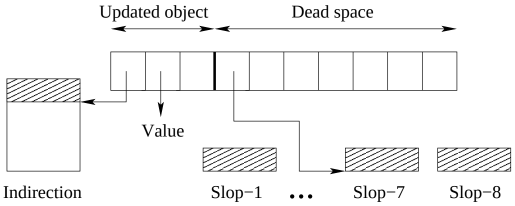


Figure 4: Slopping in to-space.

Note also, that indirections contain a third word that is used to chain indirections into a list in each old generation. This forms a generation's *remembered set* – a list of old to younger generation references. This spare slot is also present in indirections in the new generation, but is never used. In short, we do *not* have to arrange for special indirections in the young and old generations.

Recall that slop objects are only required in to-space, since this is the only place the scavenger visits. We can therefore avoid slopping altogether in from-space by suitably specialising the entry code for objects after they have been evacuated. However, because slopping in our implementation is very fast – a single word assignment in almost all cases – the performance benefits of such specialisation turns out to be negligible (see Section 4).

3.6 Fast Entry

The crucial property on which our scheme depends is that every object is entered before it is used. However, GHC sometimes short-circuits this behaviour. Consider this function:

```
f x = let { g y = x+y } in (g 3, g 4)
```

The function `g` will be represented by a dynamically-allocated function closure, capturing its free variable `x`. At the call sites, GHC knows statically what code will be executed, so instead of *entering* the closure for `g`, it simply loads a pointer to `g` into a register, and the argument (3 or 4) into another, and *jumps directly* to `g`'s code.

Notice that this only applies for dynamically-allocated function closures. For example, the function `f` also has a closure, but it is allocated statically, and captures no free variables. Hence it does not need to be scavenged, so calls to `f` can still be optimised into direct jumps.

In non-stop Haskell, the optimisation fails, because we must be sure to scavenge `g`'s closure before using it. The simple solution, which we adopt, is to turn off the optimisation. It turns out (as we show in Section 4 below) that this has a performance cost of less than 2%.

4. EVALUATION

To evaluate the various schemes we have implemented each of them in the current version of GHC (6.02). We selected 36 benchmark applications from the "nofib" suite [14]

– see below for an explanation of the ones selected – and conducted a series of experiments using both the complete set and, for more detailed analysis, a selection of eight of the applications that exhibit a range of behaviours. Benchmarks for the incremental collectors were executed on a 400MHz Celeron with 1GB RAM, a 128K level-2 cache, a 16KB instruction cache and a 16KB, 4-way set-associative level-1 data cache with 32-byte lines. The write barrier experiments were run on a 500MHz Pentium III processor with a 512K level-2 cache and identical level-1 cache, associtivity and line size specifications. The systems ran SUSE Linux 9.0 with 2.6.4 kernels in single-user mode. We deliberately chose slow machines in order to increase the execution times and reduce their variance. All results reported are averages taken over five runs and are expressed as overheads with respect to reference data. At the bottom of each table we show the minimum, maximum and (geometric) mean overhead taken over all 36 benchmarks.

In evaluating the various schemes we focus particular attention on the following questions:

1. What is the overhead on the mutator execution time (Section 4.1) and code bloat (Section 4.1.1) of each of the various schemes?

2. How do the various collectors perform in a single-generation heap? In particular, how does the dynamic one-word overhead on each copied object in our original Non-stop Haskell scheme affect the behaviour of the mutator and garbage collector (Section 4.2)?

3. In moving to a generational scheme how much can be gained by the various optimisations that accrue from specialisation (Section 4.4)? Supplementary to this, is it worth trying to eliminate the write barrier (Section 4.6)?

4. How does incremental collection change the distribution of the pause times in a generational collector (Section 4.5)? In answering this question we also wish to consider minimum mutator utilisation [5] which is a measure of the amount of useful work the mutator performs in a given "window" of time.

4.1 Baseline Overheads

We begin by analysing the overheads imposed on the mutator by the various schemes. To do this we ran each of the benchmarks in the nofib suite and selected a subset of 36 that, appropriately parameterised, satisfied the following two conditions: 1. They terminated without garbage collection using a 1GB fixed-size heap on the reference platform, 2. They executed for a sufficiently long time (at least 5 seconds, the majority much longer) to enable reasonably consistent and accurate timings to be made.

The results are shown in Table 1 for GHC's single-generation stop-and-copy collector (REF), the same with fast entry points turned off (REF*) (see Section 3.6), the traditional Baker algorithm with an explicit read barrier (BAK), our previous 'Non-stop Haskell' collector (NSH), the new variant of NSH with object specialisation and shared entry code (SPS) and the same but with inlined entry code (SPI). For the last four, we report results where incremental scavenging takes place on every mutator object allocation (-A) and every new memory block allocation (-B) – see Section 2.2.1. We list individual results for the eight benchmarks selected, in addition to averages taken over the whole benchmark suite. We report average overheads, but not average execution time. A positive overhead represents a slow-down.

For the two variants of Baker's read barrier scheme, the overheads come from:

- The need to test whether the collector is on (GC-ON) at each allocation and at each object reference (Section 2).

- The test to see whether an object has been evacuated and/or scavenged at each object reference (Section 2).

- The fact that fast entry code must be turned off (Section 3.6)

Clearly, the overhead of the second of these will be higher if incremental scavenging occurs at each object allocation, rather than at each allocation of a new memory block (4K words in the current implementation). For the remaining collectors the overhead comes from the GC-ON test at each allocation and from turning off fast-entry points.

Taken over all benchmarks, turning off fast entry points costs on average just under 2% in execution time. The GC-ON test in NSH introduces a negligible additional overhead (less than 0.3%) when performed at each block allocation; when performed at each object allocation the overhead is more significant (around 2.6%). With no garbage collection the only difference between the code executed by the NSH, SPS and SPI variants is in respect of slop filling (Section 3.5). However, this introduces a very small overhead as the vast majority of useful slop objects (covering object sizes 1–8) have been predefined (Figure 4).

Interestingly, although the code executed with the collector off is the same for both SPS and SPI, the measured times for SPI are slightly larger. We do not have a plausible explanation for this – experiments with Valgrind [25] do not show any significant change in the cache behaviour.

The individual benchmarks show significant variation in their behaviour. symalg is particularly unusual: this computes $\sqrt(3)$ to 65000 significant figures and is almost totally dominated by the construction of new data representing the result, none of which is "revisited" by the program. The execution time is practically identical over all schemes. The variations in the table are largely attributable to measurement noise.

4.1.1 Code Bloat

The binary sizes are listed in Table 2. In our implementation of Baker's algorithm extra code is inlined around all object accesses to implement the read barrier. Code is also planted around each object allocation in the case of BAK-A that invokes the incremental scavenger during garbage collection. The same overhead is paid in NSH-A, SPS-A and SPI-A. For BAK-B (likewise the other -B variants) this code is attached to the block allocator in the run-time system so the additional code overheads are negligible.

For Non-stop Haskell the code bloat comes primarily from additional static code in the run-time system, including the code to perform incremental stack scavenging (1.8MB vs 280KB approximately). For larger binaries the overhead is proportionally smaller. For NSH-A the additional overhead comes from the additional code around each object allocation.

Application	REF (s)	REF* (%)	BAK-A (%)	BAK-B (%)	NSH-A (%)	NSH-B (%)	SPS-A (%)	SPS-B (%)	SPI-A (%)	SPI-B (%)
circsim	16.29	+1.04	+38.31	+37.51	+4.85	+1.47	+6.38	+3.19	+11.42	+6.32
constraints	18.33	+3.38	+53.96	+50.52	+5.54	+2.45	+9.98	+7.36	10.69	+7.58
lambda	26.01	+0.88	+36.91	+35.22	+2.61	+1.73	+4.15	+4.27	+5.42	+1.31
lcss	19.40	–1.60	+31.13	+31.34	–0.62	–3.40	+1.34	–1.29	–0.46	–0.82
scs	23.17	+3.88	+32.50	+28.96	+6.34	+4.96	+5.83	+4.88	+8.42	+5.70
symalg	33.86	–0.15	–0.24	–0.03	+0.18	–0.27	+0.41	+0.21	+0.00	–0.15
wave4main	51.95	+1.12	+58.79	+56.13	+2.69	+2.00	+6.16	+2.93	+4.47	+2.16
x2n1	22.56	+0.09	+55.41	+53.86	+4.21	–0.18	+5.76	+1.60	+5.27	+1.73
Min 36		–1.71	–0.24	–0.03	–0.62	–3.40	–3.10	–1.29	–0.46	–1.13
Max 36		+20.45	+84.64	+82.63	+26.10	+21.91	+31.08	+25.17	+43.63	+25.03
ALL 36		+1.99	+39.20	+37.22	+4.61	+2.28	+6.54	+3.84	+7.10	+3.78

Table 1: Mutator overheads reported as a percentage overhead on GHC 6.02 execution times (REF) with no garbage collection

For the specialised code versions the overheads are primarily due to the additional entry code variants associated with each object type. Once again, the code bloat in the -A versions is higher than that in the -B versions because of the additional object allocation code, as we would expect.

4.2 Dynamic Space Overhead

Having two implementations of the barrierless Baker scheme, with and without a one-word space overhead on copied objects, gives us the opportunity to investigate the effects of dynamic space overheads. Recall that newly-allocated objects do not carry the overhead. We now explore the effects in practice.

We work with our original non-stop Haskell (NSH) scheme with the dynamic space overhead on copied (to-space) objects. However, we did not want the incremental nature of the NSH collector to interfere with the experiment – some extremely subtle effects can be observed when switching from stop-and-copy to incremental as we shall see shortly. We therefore removed the "return" instruction from the incremental scavenger so that the garbage collector actually completes in one go, essentially making the collector non-incremental. This enables us to focus on the effect of the space overhead independently of the issue of incrementality.

One further point must be understood before we proceed. All of GHC's collectors use a dynamic *sizing policy* which allocates free space for newly-allocated objects (the *nursery*) in accordance with the amount of live data at the end of the previous collector cycle. In other words, GHC does *not* work with a fixed-size heap. The default is to allocate twice as much nursery space as there is live data, with a view to keeping the residency at around 33% on average. However, the *minimum* nursery size is 256KB so the average residency may be smaller for some applications.

If each copied object carries a space overhead we create the illusion that the residency is larger than it really is, or rather, otherwise would be. Thus, the collector will allocate *more* nursery space than it would otherwise. This increases the mean time between garbage collections and hence reduces the average total number of garbage collections. This effect is countered somewhat by the 256KB minimum nursery size as we explain below.

Note that the increased memory usage per object can either be thought of as increasing the GC overhead per unit memory, or increasing the memory demand per unit runtime.

To explore the effect that the overhead has in practice, we therefore built an *adjusted* variant of NSH (NSH-1) that subtracts the overhead from the total amount of live data *before* sizing the nursery. The number of garbage collections performed by this variant should now be the same as that of a baseline stop-and-copy collector – we were able to verify this as we had made the NSH-1 variant non-incremental for the experiment.

What about a fixed-size heap, i.e. where memory is constrained? This would essentially *reduce* the nursery size compared with stop-and-copy. We could rework the GHC runtime system to do this. However we instead simulate the effect qualitatively in the context of dynamic nursery sizing by building a second variant of NSH that subtracts *twice* the overhead from the total amount of live data before sizing the nursery.

The results are summarised in Table 3. The average object size for our benchmarks is around 3.3 words so the space overhead is around 30% on all copied objects. This suggests that NSH (NSH-2) should reduce (increase) the number of garbage collections by around 30% when compared to NHS-1. However, the 256KB minimum nursery size reduces the effect somewhat. If there is less than 128KB of live data all three variants will allocate the same size nursery (256KB). We would therefore expect the average reduction/increase in garbage collections to be rather less than 30%, indeed the observed averages are -13.57% and +19.19%. For applications whose working set is less than 128KB all three variants will behave identically; we see this on a number of the smaller benchmarks.

Application	NSH-1	NSH	NSH-2
circsim	69	57 (–17.39%)	88 (+27.54%)
constraints	50	40 (–20.00%)	70 (+40.00%)
lambda	94	65 (–30.85%)	145 (+54.25%)
lcss	450	207 (–54.00%)	350 (–22.22%)
scs	561	560 (–0.18%)	648 (+15.51%)
symalg	3764	3764 (+0.00%)	3764 (+0.00%)
wave4main	298	295 (–1.01%)	306 (+2.68%)
x2n1	226	187 (–17.26%)	286 (+26.55%)

Table 3: Number of garbage collections for NSH, NSH-1 and NSH-2

4.3 Incremental Collector Performance

We now consider raw performance of the various incremental collectors in comparison with the baseline stop-and-copy collector, for a single generation. Note in particular that the NSH implementation is the unadjusted version – we do not correct for the dynamic space overhead when sizing the nursery.

Application	REF (KB)	REF* (%)	BAK-A (%)	BAK-B (%)	NSH-A (%)	NSH-B (%)	SPS-A (%)	SPS-B (%)	SPI-A (%)	SPI-B (%)
circsim	299	+0.00	+12.95	+8.46	+13.56	+9.09	+30.56	+26.06	+43.46	+38.33
lcss	249	+0.03	+12.72	+8.46	+15.16	+10.72	+30.18	+25.92	+41.28	+36.42
symalg	444	+0.00	+12.99	+7.92	+11.10	+6.04	+27.94	+22.82	+40.59	+34.70
wave4main	209	+0.04	+13.11	+9.23	+16.84	+7.10	+29.28	+25.45	+39.20	+34.71
x2n1	328	+0.00	+12.74	+8.21	+12.76	+8.24	+28.30	+23.77	+39.03	+33.95
Min 36		–0.02	+11.82	+7.24	+8.60	+2.77	+27.14	+22.08	+37.85	+32.17
Max 36		+0.14	+14.36	+9.28	+19.16	+15.60	+30.56	+26.79	+47.49	+40.35
ALL 36		+0.02	+12.99	+8.55	+14.21	+9.76	+29.14	+24.68	+40.90	+35.74

Table 2: Code bloat reported as a percentage overhead on the stop-and-copy collector

For the two Baker variants, BAK-A and BAK-B, we report overheads for collectors *without* incremental stack scavengers. The stacks are scavenged in their entirety at the GC flip (see Section 2.2.2) – the addition of incremental stack scavengers would further degrade the performance of the collectors that already incur the highest overhead. All other collectors employ incremental stack scavengers. The results are presented in Table 4.

It is clear from the table that something quite unusual is going on. Averaging over all 36 benchmarks the incremental-B schemes are actually running *faster* than stop-and-copy. Closer inspection of the figures shows that some applications (bernoulli, gamteb, pic and symalg) exhibit substantial speed-*ups* (35-79%) across *all* incremental variants. There are also significant outliers in the other direction (paraffins and scs, for example). Curiously, wave4main exhibits extreme speed-*ups* across all -B variants and extreme slow-*downs* across all -A variants!

Because the effects are seen across the board it suggests that it is an artefact of the incremental nature of the garbage collector, rather than the idiosyncrasies of any particular implementation. Examination of the Valgrind cache traces reveals no obvious trend that might explain the differences.

It turns out that the extreme behaviours are attributable to two different aspects of incremental collector behaviour. The extreme speed-ups are an artefact of long-lived data and the fact that stop-and-copy collection happens instantaneously with respect to mutator execution. During incremental collection, data that must be retained by the stop-and-copy collector is given the chance to die – specifically if the mutator updates the root of the data *before* the scavenger evacuates it. The greatest speed-ups are observed when 1. incremental collection cycles are long, 2. the lifetime of the data is slightly shorter than the incremental collection cycle, 3. the data is not in the *immediately* active portion of the mutator's working set and 4. death occurs before the scavenger evacuates it.

The extreme slow-downs are an artefact of incremental collection cycles that are *forced to completion.* The stop-and-copy collector sizes the nursery based on the amount of live data at the end of the *current* cycle, while the incremental collector can only use that of the previous cycle. If the sizing policy does not size the nursery large enough for the collection of from-space to complete before the nursery is again exhausted, then the current cycle is forced to completion and a new cycle is started. This can result in the incremental collector doing significantly more work than the stop-and-copy collector.

One way to remedy this is to adjust the amount of work done at each allocation on the basis of forced completions in previous collector cycles. However, as GHC has a block-based allocator, the most reliable solution is to allocate one or more extra blocks to the nursery as needed. This is a straightforward modification and has the added benefit that it enables us to provide hard bounds on the pause times (but see Section 6.1 below), although we have not had a chance to evaluate it in practice.

Intriguingly wave4main exhibits both of these traits. The incremental cycles are long and span 98% of the total execution time. When scavenging occurs at every block allocation, the scavenger is invoked much less frequently than at every allocation and the mutator evacuates objects in reference order – the immediately active portion of the working set. A large number of long-lived objects outside this portion die before the scavenger evacuates them. However, the scavenger that operates at every object allocation evacuates data in closure layout order and is invoked much more frequently. This results in the scavenger encountering the data that previously died off in this cycle and its evacuation. A high rate of allocation and increasing amount of live data compound the effect, resulting in a significant number of forced completion cycles. This significantly increases the amount of data copied during garbage collection.

Note that NSH performs well because the sizing policy incorporates the extra word in the live data calculations. In practice, the collector would never be chosen over an SPS or SPI collector. Instead, the sizing factor, a configurable parameter to the collector, would be adjusted to provide equivalent performance.

4.4 Incremental Generational Collectors

We now investigate the raw performance of the various collectors with two generations and two steps per generation (GHC's current default configuration). This is of interest as it indicates the bottom line on performance that we would expect to see in a "production" environment. The results are shown in Table 5. The performance of each variant is expressed as an overhead on the execution time observed using GHC's baseline generational collector. Once again the results suggest that there is no advantage at all to inlining entry code in each variant. Indeed the mutator overhead that accompanies the code bloat often leads to worse performance when compared to SPS that shares a single copy of each closures's original entry code.

An interesting observation is that the extreme behaviour we observed in the single-generation collectors is much diminished. This is because the nursery in the generational collector is of fixed size (256KB) so that the collection cycles are relatively small (point 1. above). This all but eliminates the pathological speed-*ups*. Furthermore, the long-lived data is promoted to the older generation more quickly where garbage collection is less frequent; the extreme effects we saw earlier are altogether much less apparent.

Application	REF (s)	REF* (%)	BAK-A (%)	BAK-B (%)	NSH-A (%)	NSH-B (%)	SPS-A (%)	SPS-B (%)	SPI-A (%)	SPI-B (%)
circsim	45.42	–1.25	+65.06	+57.24	+13.25	+6.80	+22.04	+13.67	+22.85	+14.70
constraints	50.09	+3.29	+82.87	+63.19	+8.17	+3.09	+22.80	+10.32	+23.12	+11.08
lambda	52.03	+3.50	+93.47	+57.89	+15.47	+2.08	+20.51	+4.36	+21.72	+3.77
lcss	51.41	–1.24	+53.69	+38.16	+11.26	+7.62	+11.79	+9.04	+11.57	+8.69
scs	43.44	+3.15	+99.88	+93.62	+86.26	+69.66	+58.40	+50.41	+63.61	+57.14
symalg	76.19	+0.67	–65.99	–65.99	–66.23	–66.06	–66.32	–66.22	–66.09	–66.20
wave4main	509.24	–0.14	+92.85	–36.53	+75.18	–64.73	+49.43	–60.84	+50.57	–61.20
x2n1	45.56	–0.81	+48.9	+31.74	+8.58	–9.20	+6.01	+0.88	+6.19	–1.25
Min 36		–1.25	–71.73	–72.96	–78.29	–74.73	–77.55	–78.57	–77.90	–78.82
Max 36		+11.05	+248.48	+177.00	+86.26	+69.66	+62.01	+54.83	+68.51	+52.72
ALL 36		+1.53	+41.98	+26.37	+3.45	–5.58	+4.73	–6.03	+4.65	–7.68

Table 4: Single-generation incremental collector performance as an overhead on baseline stop-and-copy (REF)

Application	REF (s)	REF* (%)	BAK-A (%)	BAK-B (%)	NSH-A (%)	NSH-B (%)	SPS-A (%)	SPS-B (%)	SPI-A (%)	SPI-B (%)
circsim	40.15	–2.19	+24.46	+22.76	+16.06	+13.05	+7.95	+4.48	+9.94	+7.10
constraints	58.56	+1.42	+19.02	+16.96	+10.86	+9.90	+6.11	+3.48	+7.67	+4.80
lambda	46.92	+2.39	+30.18	+25.47	+10.40	+7.10	+9.10	+3.30	+11.15	+0.79
lcss	56.28	+3.86	+42.36	+33.26	+20.84	+16.12	+14.23	+7.11	+15.05	+7.43
scs	31.75	+1.98	+30.27	+26.65	+10.16	+8.85	+8.15	+5.01	+9.34	+6.46
symalg	26.28	+0.08	+7.08	+0.46	+9.13	+0.76	+8.30	+0.61	+8.53	+1.56
wave4main	286.45	–0.76	+10.56	+10.88	+4.53	+5.55	+4.50	+5.85	+4.41	+6.04
x2n1	210.01	–2.87	+6.33	+4.96	+10.59	+7.63	+8.82	+9.76	+8.99	+10.66
Min 36		–3.33	–0.11	–0.08	–4.04	–3.40	–3.10	–2.54	–3.21	–2.31
Max 36		+8.33	+79.14	+76.57	+26.02	+27.76	+23.49	+20.45	+17.88	+16.06
ALL 36		+0.44	+24.62	+22.59	+9.18	+7.70	+6.78	+4.42	+8.27	+4.76

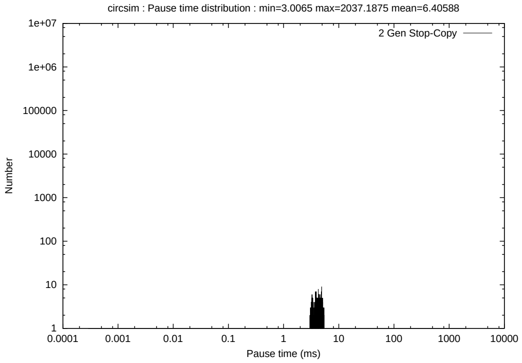
Table 5: The Bottom Line: Incremental generational collector performance as an overhead on execution time compared to GHC's current generational collector (REF)

4.5 Pause Times and Mutator Progress

Our previous paper showed the mean pause times for both per-object and per-block (-A and -B) schemes to be very favourable over a range of benchmarks. In this paper we focus on the pause time distribution and mutator progress for a small subset of the benchmarks. To gather these we used PAPI [15] to get access to the hardware performance counters and produced a trace of the *wall clock* time for each pause. It is important to understand the measurements are wall-clock timings; in some rare cases the pause times include context switches. Also, some very small pauses are subject to measurement granularity. We are forced to use wall clock time, as the time quantum PAPI is able to multiplex for per-process "user" time is generally larger than the incremental collector pause times.

We show here results for circsim, x2n1 and wave4main. The pause time distribution graphs for GHC's generational collector and that of SPS-A and SPS-B are included. In the stop-and-copy generational collector the mean pause time was 6.4ms; for SPS-A this was around 7μs. The longest pauses were 2.04 seconds (a major collection), and 738ms respectively. The figure for SPS-A is unusually large considering the granularity of incremental scavenging – this is an artefact of a forced completion. This highlights the need to eliminate forced completions in order to bound the pause times. The results for x2n1 and wave4main can be seen to be similar.

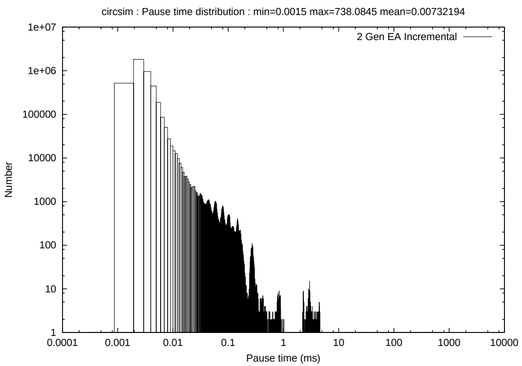
The minimum mutator utilisation (MMU) graphs for circsim, x2n1 and wave4main are included and overlay the baseline (stop-copy) generational collector and both the SPS-A and SPS-B schemes. These MMU graphs show the lowest average mutator utilisation seen over the entire program run, for varying intervals of time. Notice that the utilisation is greatest in SPS-A and least for the baseline collector. However, recall that whilst pause times are shorter, the execution time is extended – this is as we would expect.

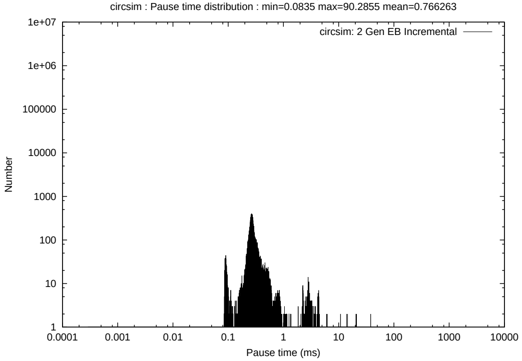


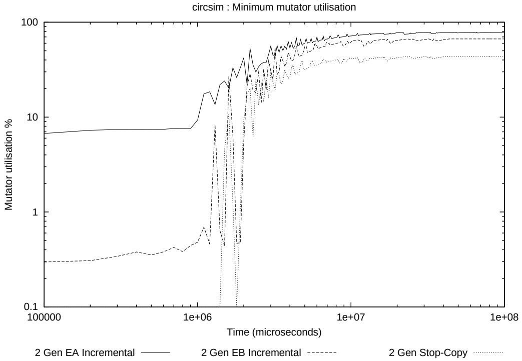
4.6 The Write Barrier

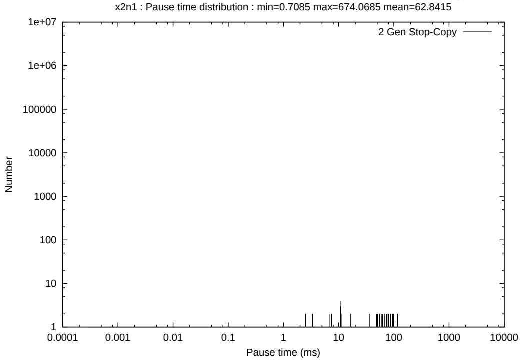
Each of the collectors above avoids the explicit write barrier using the techniques described in Section 5. Historically, the write barrier has been found to have a small overhead on execution time – see for example [17]. The issue is potentially more interesting in the current GHC because of its block-allocated memory. To implement the write barrier the object's address must first be mapped to a block identifier and the block header then examined to determine in which generation it sits. The cost of the write barrier is 15 instructions on a Pentium III, compared with just two instructions for a contiguous heap. Does this significantly affect overall execution time?

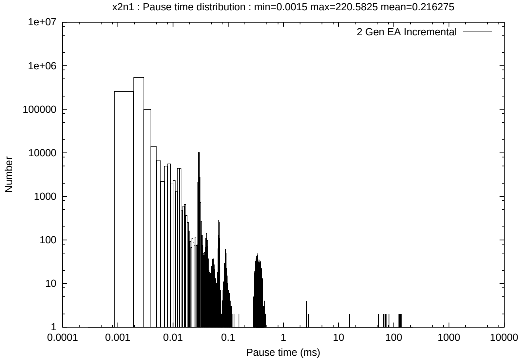
To find out we ran the benchmarks with a 1GB heap so that no garbage collection, and hence no promotion, takes place. We used the baseline generational collector (i.e. with a write barrier) and compared it with our barrierless col-

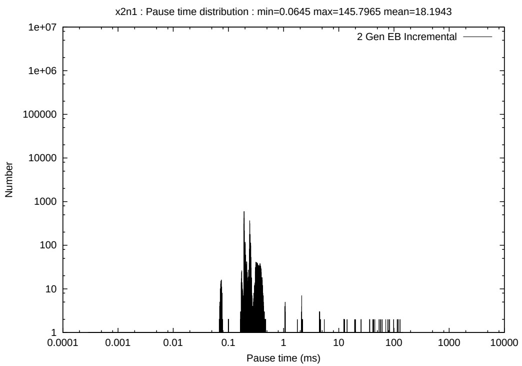

circsim : Pause time distribution : min=0.0015 max=738.0845 mean=0.00732194
2 Gen EA Incremental
Number
1e+07
1e+06
100000
10000
1000
100
10
1
0.0001
0.001
0.01
0.1
1
10
100
1000
10000
Pause time (ms)

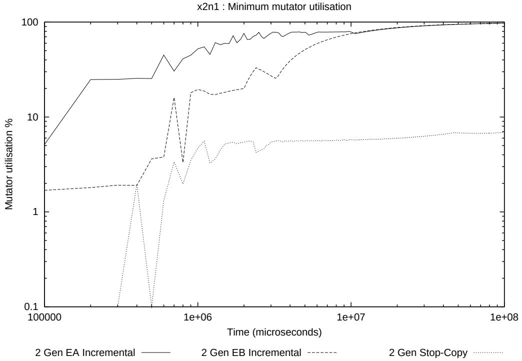

circsim : Pause time distribution : min=0.0835 max=90.2855 mean=0.766263
circsim: 2 Gen EB Incremental
Number
1e+07
1e+06
100000
10000
1000
100
10
1
0.0001
0.001
0.01
0.1
1
10
100
1000
10000
Pause time (ms)

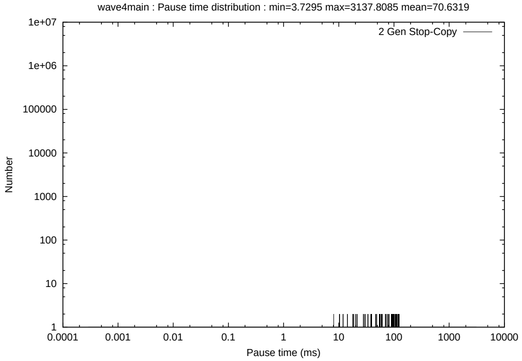

circsim : Minimum mutator utilisation
Mutator utilisation %
100
10
1
0.1
100000
1e+06
1e+07
1e+08
Time (microseconds)
2 Gen EA Incremental
2 Gen EB Incremental
2 Gen Stop-Copy

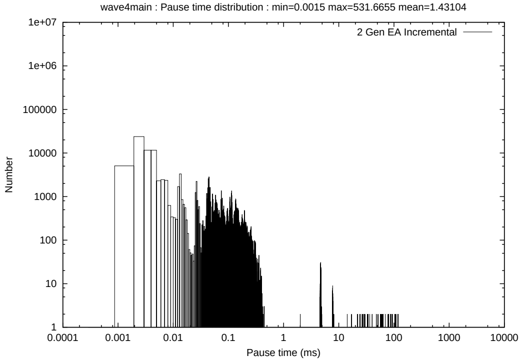

x2n1 : Pause time distribution : min=0.7085 max=674.0685 mean=62.8415
2 Gen Stop-Copy
Number
1e+07
1e+06
100000
10000
1000
100
10
1
0.0001
0.001
0.01
0.1
1
10
100
1000
10000
Pause time (ms)

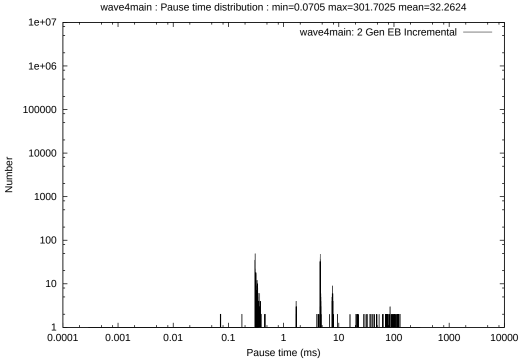

x2n1 : Pause time distribution : min=0.0015 max=220.5825 mean=0.216275
2 Gen EA Incremental
Number
1e+07
1e+06
100000
10000
1000
100
10
1
0.0001
0.001
0.01
0.1
1
10
100
1000
10000
Pause time (ms)

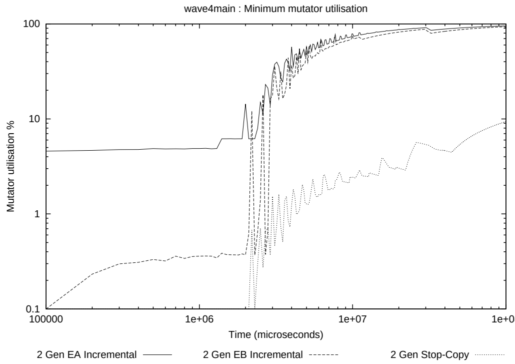

x2n1 : Pause time distribution : min=0.0645 max=145.7965 mean=18.1943
2 Gen EB Incremental
Number
1e+07
1e+06
100000
10000
1000
100
10
1
0.0001
0.001
0.01
0.1
1
10
100
1000
10000
Pause time (ms)


x2n1 : Minimum mutator utilisation
Mutator utilisation %
100
10
1
0.1
100000
1e+06
1e+07
1e+08
Time (microseconds)
2 Gen EA Incremental
2 Gen EB Incremental
2 Gen Stop-Copy


wave4main : Pause time distribution : min=3.7295 max=3137.8085 mean=70.6319
2 Gen Stop-Copy
Number
1e+07
1e+06
100000
10000
1000
100
10
1
0.0001
0.001
0.01
0.1
1
10
100
1000
10000
Pause time (ms)


wave4main : Pause time distribution : min=0.0015 max=531.6655 mean=1.43104
2 Gen EA Incremental
Number
1e+07
1e+06
100000
10000
1000
100
10
1
0.0001
0.001
0.01
0.1
1
10
100
1000
10000
Pause time (ms)


wave4main : Pause time distribution : min=0.0705 max=301.7025 mean=32.2624
wave4main: 2 Gen EB Incremental
Number
1e+07
1e+06
100000
10000
1000
100
10
1
0.0001
0.001
0.01
0.1
1
10
100
1000
10000
Pause time (ms)



lector with specialised thunk code, but running in a non-incremental mode (SPI-SC). Actually, only the young generation variant of the thunk code will be executed as no objects are ever promoted. Since neither collector executes, the difference is in the mutator code to implement the barrier. We then re-ran the benchmarks with the same collectors, but using the standard heap sizing policy – two generations and two tenuring steps – so that garbage collection and object promotion now occur. The results are shown in Table 6. Note, the REF times differ to those of Table 4 because different machines were used for the execution of the benchmarks (see Section 4).

Running in a 1GB heap the SPI-SC figures indicate the maximum benefit that can be expected from removing the write barrier – if all objects sit in the young generation then the write barrier is wholly superfluous. The figures show that removing the barrier altogether would only gain around 1.7% in performance averaged over all benchmarks. The figures for the standard configuration show the benefit in more typical situations. The conclusion is that, despite the relatively expensive write barrier in GHC, the barrier overheads are actually very small.

5. BRIDGING THE GENERATION GAP

Although for many applications removal of the write barrier doesn't pay, it is possible in principle to eliminate it using the same specialisation trick that we used to optimise our Non-stop Haskell collector. For the generational collector of "Eager Haskell" [12], or some write-intensive applications, the optimisation may be worthwhile, and so we outline here how it can be done.

	1GB Heap		Standard Heap	
Application	REF (s)	SPI-SC (%)	REF (s)	SPI-SC (%)
circsim	14.06	–0.07	38.68	+0.34
constraints	13.79	+1.96	53.96	+4.78
lambda	22.26	+0.04	44.27	+5.35
lcss	17.77	–4.11	53.95	+1.72
scs	18.86	+2.01	27.19	–0.59
symalg	35.31	–0.04	28.04	+0.11
x2n1	11.90	–3.87	71.70	–3.38
ALL 36		-1.67		-0.51

Table 6: Costing the write barrier

We want to achieve two effects:

1. We would like there to be *no* write barrier when updating thunks in the young generation as these updates cannot generate inter-generational pointers.
2. When updating an object in the old generation we would like the thunk to be added to the remembered set *automatically* either when it is entered or when it is eventually updated.

The bottom line is that we want the update code to be different for the young and old generations. The trick is to hijack the info table pointer associated with a thunk at the point where the garbage collector promotes it so that it behaves differently depending on the generation in which it sits.

The simplest modification requires one additional variant of the thunk entry code (each thunk is specialised) that will be used only when the object has been promoted (designated a *thunk barrier*). The thunk barrier code adds the thunk to the remembered set immediately after it has been black-holed. Note that we cannot do this before black holing because the act of adding to the remembered set would corrupt the payload (recall that the remembered set is a list formed by chaining objects together – Section 3.5).
Technical detail: Note that this requires *eager black holing.* The alternative is to wait until the next garbage collection and do the black holing whilst scanning the update frames on the stack. This is called *lazy black holing* and is shown in [19] to be cheaper than black holing thunks at the point where they are entered. Note also that promoting a black hole requires the object to be added to the remembered set.

This *add-on-entry* scheme is actually not a new idea. Niklas Rojemo proposed the idea in [16] to enable tenuring policies to be modified with low overhead. The disadvantage is that the list will contain many objects in the "black hole" state for which no inter-generational pointers yet exist: these (may) appear when the object is eventually updated. This adds an overhead to minor collections because the remembered set that is scanned as part of the root set for the young generation can now be significantly longer.

A better approach is to arrange for the object to be added to the remembered set when it is *updated* – this may happen some time after the object has been entered, as explained earlier. This *add-on-update* scheme requires a second (generic) variant of the update frame for the old generation which additionally adds the *updated* thunk to the remembered set. The entry code for the old generation is the same as for the young generation except that it pushes the modified update frame. The code bloat for thunk and (single) update frame specialisation is on average 14.5% across all the benchmarks *before* combination with the incremental collector variants.

To complete the picture, self-scavenging variants of both the original entry code and thunk barrier entry code are needed as before: a total of four specialised thunk variants. Figure 5 shows a thunk and its four specialised variants. The example shows the info pointer set as it would be after promotion, before the thunk has been updated and after it has been scavenged during garbage collection. The picture would be the same for a promoted thunk when the garbage collector is off.

Although the code bloat as described would be substantial we can again choose to share the entry code that is common to each. Our previous evaluation shows that this gives almost identical performance whilst reducing the code bloat.

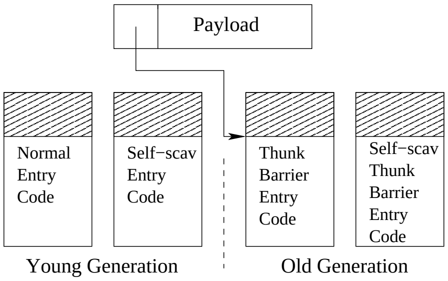


Figure 5: An evacuated closure in the specialised self-scavenging thunk barrier scheme.

6. CONCLUSIONS AND FUTURE WORK

Our experiments have shown that closure code specialisation can buy performance when it is used to remove dynamic space overheads. A 25% code bloat over stop-and-copy (an additional 15% over our previous Non-stop Haskell collector) buys us an incremental generational collector that runs only 4.5% slower than stop-and-copy when averaged over our chosen benchmarks and around 3.5% faster than our previous "Non-stop Haskell" collector (of course the benefits are greater when operating with limited memory, i.e. in the confines of a fixed-size heap).

Although specialisation opens up a number of additional optimisation opportunities, for example write barrier elimination, they appear to buy very little in practice. With respect to building a "production" garbage collector for GHC our preferred option is therefore to use specialisation to remove the dynamic space overheads and provide a fast implementation of Baker's algorithm that avoids the read barrier, and to pay the small additional cost of the write barrier to limit the code bloat.

6.1 Bounded Pause Times

Provided a program does not make use of large objects (mostly occurring as arrays allocated in separate blocks to the rest of the objects in the heap), we can, in theory, provide *hard* bounds on the pause time in keeping with the work of [4], for example. We did not have a handle on this in our previous work because we had no way of bounding the unit of work required by the stack scavenger – this has now been fixed. We would have to prevent forced completions, of course, which lead to an unbounded pause at the end of those collection cycles where the collector can't keep up with the mutator. We have suggested a way to do this that exploits GHC's block-allocated memory system, although we have not yet implemented it.

Large objects still present a problem. As it stands these have to be scavenged in one go although we could break these up into smaller fixed-sized structures [3] accessed by more expensive primitives with explicit read barriers. Alternatively a scheme which employs dynamic dispatch and our self-scavenging techniques could be used.

7. REFERENCES

[1] A. Appel. Simple Generational Garbage Collector and Fast Allocation. *Software Practice and Experience*, 19(2), pages 171–83, 1989.

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

[3] D.F. Bacon, P. Cheng, and V.T. Rajan. A Real-time Garbage Collector with Low Overhead and Consistent Utilization In *ACM SIGPLAN Symposium on Principles of Programming Languages* , pages 285–98, 2003.

[4] G. Blelloch and P. Cheng. On bounding time and space for multiprocessor garbage collection. In *ACM SIGPLAN Symposium on Programming Language Design and Implementation*, pages 104–17, 1999.

[5] P. Cheng and G. Blelloch. A parallel, real-time garbage collector. In *ACM SIGPLAN Symposium on Programming Language Design and Implementation*, pages 125–36, 2001.

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

[7] A.M. Cheadle, A.J. Field, S. Marlow, S.L. Peyton Jones, and R.L. While. Non-stop Haskell In *International Conference on Functional Programming*, pages 257–67, 2000.

[8] C. Cheney. A non-recursive list compacting algorithm. In *CACM 13(11)*, pages 677–8, 1970.

[9] J. DeTreville. Experience with conncurrent garbage collectors for Modula-2+. Technical Report 64, DEC Systems Research Center, Palo Alto, CA, August 1990.

[10] B. Hayes. Using key object opportunism to collect old objects. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications* , pages 33–46, 1991.

[11] S. Marlow and S. Peyton Jones Making a Fast Curry – Push/enter vs eval/apply for higher-order languages Submitted to *International Conference on Functional Programming*, 2004

[12] J.W. Maessen Hybrid Eager and Lazy Evaluation for the Efficient Compilation of Haskell PhD thesis, Massachusetts Institute of Technology, USA, 2002

[13] S.M. Nettles, J.W. O'Toole, D. Pierce and N. Haines. Replication-based incremental copying collection. In *Proceedings of the International Workshop on Memory Management, LNCS 637.* Springer Verlag, pages 357–64, 1992.

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

[15] PAPI: Performance Application Programming Interface http://icl.cs.utk.edu/papi/

[16] N. Rojemo. Generational Garbage Collection for Lazy Functional Languages Without Temporary Space Leaks Memory Management Workshop, pages 145–62, 1995. ftp://ftp.cs.chalmers.se/pub/users/rojemo/iwmm95.ps.gz

[17] P.M. Sansom and S.L. Peyton Jones. Generational garbage collection for Haskell In *Proceedings of the ACM conference on Functional Programming Languages and Computer Architecture*, pages 106–16, June 1993

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

[19] S. Peyton Jones. Implementing lazy functional languages on stock hardware: the Spineless Tagless g-machine. In *Journal of Functional Programming*, pages 127–202, 1992.

[20] S. Peyton Jones, S. Marlow, and A. Reid. The STG runtime system (revised). Draft paper, Microsoft Research Ltd, 1999.

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

[22] J. Seward. Generational Garbage Collection for Lazy Graph Reduction. In *International Workshop on Memory Management*, pages 200–17, 1992.

[23] R.A. Shaw. Improving garbage collector performance in virtual memory. *Technical Report CSL-TR-87-323, Stanford University*, March 1987.

[24] D.M. Ungar. Generation scavenging: A non-disruptive high performance storage reclamation algorithm, In ACM SIGSOFT/SIGPLAN *Software Engineering Symposium on Practical Software Development Environments*, pages 157–67, April 1984.

[25] Valgrind, an open-source memory debugger for x86-linux. http://valgrind.kde.org.

[26] B. Zorn. Comparative Performance Evaluation of Garbage Collection Algorithms. PhD thesis, University of California at Berkeley, 1989.

[27] B. Zorn. The measured cost of conservative garbage collection. In *Software Practice and Experience*, volume 23, pages 733–56, 1993.