The New GHC/Hugs Runtime System

Simon Marlow   Simon Peyton Jones
University of Glasgow

October 9, 1998

Abstract

This paper describes the new runtime system being developed for the Glasgow Haskell Compiler. The goal is to provide support for mixed interpreted/compiled execution of Haskell programs, with Hugs as the interpreter. In the process, we've taken the opportunity to fix some of the deficiencies of the old system (such as garbage collection of CAFs), add some new features (cost-centre stack profiling, and a more flexible storage manager), and improve performance (one stack instead of two, better register usage on register-challenged architectures).

We also took the opportunity to design a clean, simple API to the runtime system that would allow it to be used in a variety of applications, from standalone Haskell binaries and interactive interpreters to encapsulated COM/CORBA objects.

1 Introduction

Compilers get the headlines, but behind every great compiler lies a great runtime system. A surprising amount of work goes into runtime systems, but disproportionately few papers are written about them. This paper describes the design of the runtime system we have built to support mixed compiled/interpreted execution of Haskell.

Runtime systems start simple; after all, the compiler translates all the programmer-written source into executable code, so what is there to do? Initially not much, perhaps, but serious language implementations rapidly grow features that require run-time support. To be specific, the runtime system we describe in this paper supports the following features:

Mixed compiled/interpreted execution was our prime goal (Section 6.4). The programmer sees the Hugs interpreter, but can load modules compiled by the Glasgow Haskell compiler (GHC). Execution switches between compiled and interpreted code at a rather fine-grain level.

The storage manager includes a generational garbage collector (Section 4). The storage manager is made more complicated by the need to support a number of language extensions, namely:

- *Mutable variables and arrays* support the state monad [LJ95].

- *Weak pointers* allow the programmer to build self-cleaning memo-tables, among other things [DBE93].
- *Stable pointers* are immutable names for heap objects; they allow pointers to Haskell objects to be held outside the heap.

From the programmer's point of view, a big improvement is that the maximum heap size no longer has to be specified; instead, the heap simply grows (and shrinks) as required.

Concurrency. The run-time system supports Concurrent Haskell by default [PJGF96], and hence has to manage multiple Haskell threads.

Profiling is now recognised as even more important for functional languages than for imperative ones. We support both space and time profiling, using Jarvis's *cost-centre stacks* [Jar96], a development of our earlier work on cost centres [SPJ95] (Section 7).

The foreign language interface allows C or COM/CORBA to call a Haskell function, or be called by a Haskell function.

2 Overview

The runtime system has three main components (see Figure 1):

- A scheduler, which manages the various threads in the system. The scheduler sits at the centre of the action: every thread returns to the scheduler before the next thread is run, and the garbage collector is normally called from the scheduler level. The scheduler is described in detail in Section 6.
- The storage manager consists of a low-level block allocator that requests storage on demand from the operating system, and a garbage collector. The storage manager is described in Section 4.
- The interpreter executes bytecodes. These may be read from a file, or dumped directly in the heap by an external bytecode compiler.

The entire runtime system is a library that can be embedded in some larger program (the *client application*). The ones we have in mind are:

- The Hugs interpreter;
- a standalone GHC-compiled binary;
- a library callable from some enclosing C++ or Java program;
- a COM or CORBA object.

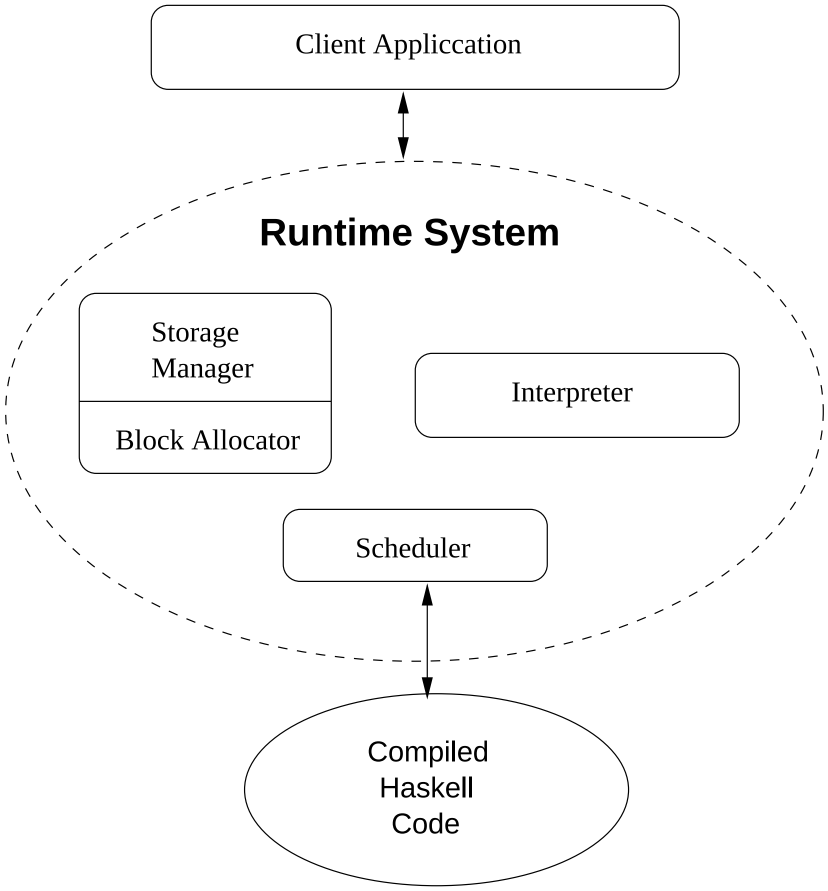

Figure 1: Runtime System Overview

3 Heap objects

A central aspect of the design is the shape of heap-allocated objects. In particular, since an object might be allocated by Hugs, and then used by GHC-compiled code, or vice versa, Hugs and GHC must agree precisely on the layout of heap objects.

Every heap object looks like Figure 2. The first word of every object is called its *info pointer*, and points to the static *entry code* of that object. The entry code is the code that evaluates the object to weak head normal form; it is the most-used field during execution (as opposed to garbage collection).

Immediately before the entry code is the object's *info table*. The info table has fields that say what kind of object this is, and hence what its layout is; this information is required by the garbage collector. More specifically, the fields of an info table are as follows:

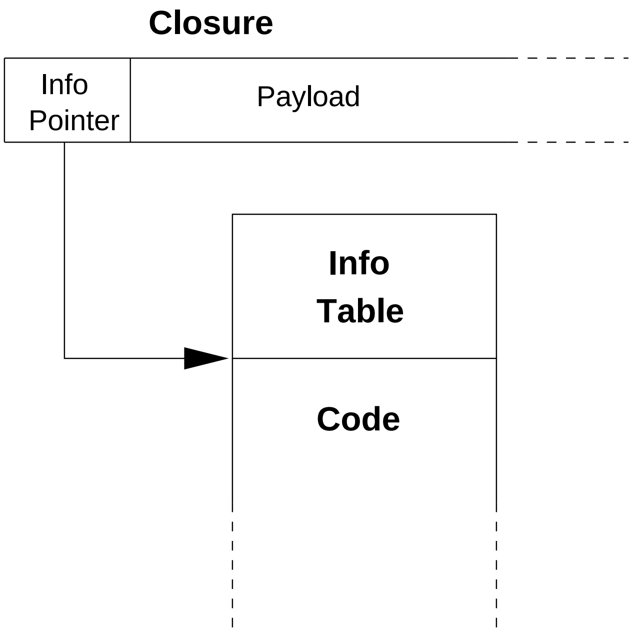

Figure 2: A Closure

- Closure type: a constant indicating which kind of closure this is (eg. function, thunk, constructor etc.)
- Layout information for the garbage collector: either a pair (pointers, non-pointers) or a bitmap.
- A pointer to the SRT for the object (see Section 4.6)
- Closure flags: several on-off flags for quick access to certain properties of the closure. (eg. is static, is pointed etc.)
- Various optional profiling, parallel and debugging fields.

A simpler[1] alternative would be to make the info pointer point to the info table, and point to the entry code from an entry in the info table. Having the info pointer point *directly* to the entry code, instead, looks attractive because it requires one fewer indirections. However, it pollutes the instruction cache with never-executed info tables, and may not be a win on modern architectures. We intend to experiment with the more conventional layout.

The runtime system supports quite a wide variety of heap objects. They fall into three main groups:

[1]Simpler, not only because it is more conventional, but also because one can only generate this layout if one generates assembly code directly. If one goes via C, then one has to post-process the output code to achieve the same effect.

Pointed objects represent values of a pointed type; that is, a type that includes bottom. All pointed objects may be evaluated (or *entered*). This is done by putting the address of the object in a register and jumping to the object's entry code. Pointed objects include:

- Thunks (unevaluated closures).
- Data constructors.
- Functions.
- Partial applications — these arise from an update of a function-valued thunk.
- Indirections, which are used to overwrite a thunk with its value.
- Black holes, which overwrite a thunk when its evaluation is begun, but before its value has been computed (Section 5.1).

Unpointed objects represent values of unpointed type; that is, a type that does not include bottom. A pointer to an unpointed heap object never points to a thunk or an indirection; it always points to the object itself. Examples include:

- (Immutable) arrays.
- Mutable arrays and variables.
- MVars — required for Concurrent Haskell.
- Weak Pairs — for supporting memo tables.

Administrative objects are allocated by the runtime system itself, and never seen by the programmer. They include:

- Thread state objects (TSOs). These contain the state of a given Haskell thread, including the thread's stack.
- Byte code objects (BCOs). These contain byte codes for the interpreter.
- The stable pointer table, which maps the stable name of an object to its heap address.

4 The Storage Manager

The runtime system includes a completely rewritten storage manager providing several enhancements both for compiled and interpreted execution. Hugs in particular will benefit from an industrial-strength garbage collector.

To summarise the main improvements provided by the new storage manager and garbage collector:

- Dynamic resizing for heap and thread stacks: better performance for programs with low memory requirements, and no more heap-size settings. The storage manager will also be able to adjust its memory requirements to optimise the trade-off between time and space. For example, it could reduce the size of the heap if the system is paging excessively.

- The garbage collector will support multiple generations, and will even be able to adjust the number of generations during execution, depending on the storage requirements of the program. It will also allow objects to be aged within a generation, by dividing the generation into *steps*; this prevents objects being promoted to the next generation when they are still very young.
- Top-level expressions (CAFs) will be garbage collected. Previously, the storage allocated by a CAF would be retained throughout the lifetime of the program. The new storage manager will be able to traverse the code tree and discover which CAFs are still accessible, allowing garbage CAFs to be collected. More about this in Section 4.6.
- Large objects, such as arrays and thread stacks, can be treated independently by the storage manager, allowing them to remain at static locations in memory.
- Weak pointers. Using weak pointers (or weak pairs) we can provide support for hashed memo tables without space leaks.

The driving technical decision is to allow the heap to consist of many dis-contiguous areas, rather than a single large continguous address space. In fact, we go further, dividing the heap into *blocks* of uniform size (4k bytes at present). The heap can then be partitioned between generations, and age steps with each generation, in a flexible and dynamic way. The blocks making up a partition do not need to be contiguous, so it is easy to use any free block for any purpose.

Many other sophisticated storage managers have adopted a similar design [DEB94, Rep94]. It makes the storage manager more complicated, and a little less efficient, but it is much more flexible and robust, as we discuss below.

4.1 Structure

The storage manager is divided into two layers:

- The Block Allocator. This is the low level machine dependent layer in the storage manager. It provides memory in units of *blocks* to the upper levels.
- The rest of the storage manager. This consists of the garbage collector and two memory allocation interfaces: one for compiled code which provides a collection of blocks for fast sequential allocation, and a separate interface for the interpreter and other external use.

A *block* is a power-of-2 sized chunk of memory. Blocks may be lumped together to form larger contiguous chunks of memory called *groups*. Each block has an associated *block descriptor*, which stores information about the block:

- The address of the block itself,
- the first byte of free memory in the block,

- for the first block in a group, the number of blocks in the group,
- which generation/step the block belongs to,
- a link to another block descriptor.

Blocks (and groups) can be linked together using the link field to form chains. Chains are used for example to link together all the blocks in a single generation, and to maintain the list of free blocks in the block allocator.

The flexibility of a block-structured storage manager provides several advantages:

- If memory runs short, the runtime system can request more from the operating system, without requiring it to be contiguous with the existing area. The requirement that the programmer must often specify a maximum heap size is a major annoyance of the current GHC implementation.
- Individual generations and allocation areas can be resized according to the memory requirements of the program, by chaining blocks together. There is no need for the memory in a generation to be contiguous.
- Large objects can be allocated and managed independently, allowing the garbage collector to simply chain the objects onto new lists instead of copying the data itself during garbage collection.
- Memory can be recycled quickly: as a block becomes free it can be immediately linked onto a new list and reused. Consider a two-space collector running at 50% residency: if the amount of live data is n bytes, then the total memory use is $4n$. With a block-structured garbage collector we can do this in $3n$ bytes because the allocation area is re-used after each GC.

The block allocator provides an interface whereby blocks and groups may be allocated to the upper levels, and later freed again. The block allocator maintains a pool of free blocks and requests memory from the operating system as necessary. Memory is requested from the operating system in units of *megablocks*, a multiple of the block size. Each megablock is divided into blocks with a small amount of space reserved for block descriptors at the start of the megablock. The block descriptor for the block containing any address is given by a simple function of the address.

The storage manager is largely independent of the actual size of blocks and megablocks that are chosen, with the only requirements that both are powers of 2 in size and a megablock is a multiple of the block size. Currently we use 4k blocks and 1M megablocks.

The storage manager is self-contained with a simple API, consisting of just 7 functions. In the following sections we will describe these functions.

4.2 Fast Sequential Allocation

Three functions provide access to contiguous chunks of memory which compiled code can fill with objects sequentially. The storage manager provides a number of blocks collectively known as the nursery (where objects are born!). Each nursery block will be filled in turn, until the nursery is full when a garbage collection is needed. The garbage collector will provide a new set of nursery blocks. The size of the new nursery can be adjusted to maximise performance or reduce memory use.

```
OpenNursery(hp,hplim)
ExtendNursery(hp,hplim)
CloseNursery(hp,hplim)
```

Compiled code uses two registers for allocation purposes: the heap pointer `Hp` and the heap limit `HpLim`. On entering compiled code from the scheduler, a call to `OpenNursery` is performed. This call initialises the `Hp` and `HpLim` registers, usually with `Hp` pointing to the start of the next free nursery block, and `HpLim` pointing to the end.

Each basic block performs a *heap check*, which simultaneously increases `Hp` while checking that it has not exceeded the value of `HpLim`. The code following the heap check then fills in the memory with new objects. If the check fails, then `ExtendNursery` is called. This function either initialises `Hp` and `HpLim` to the next free nursery block, or fails indicating the nursery block pool is exhausted and a garbage collection is required.

On exiting compiled code, `CloseNursery` is called which saves away the next free location in the current nursery block in the block descriptor ready for the next `OpenNursery`.

The upshot is that code can allocate away happily by incrementing `Hp` until `HpLim` is reached, when either a new block will be chained on or the code must return to the scheduler for a garbage collection. The calls to `ExtendNursery` are performed out of line (the compiled code jumps to some common heap-check failure code), but are nevertheless very cheap. All the nursery functions are implemented as short macros.

One advantage with this allocation scheme is that we have the opportunity to perform some extra checks out of line in the "minor" heap check failure code that we wouldn't want to do at every heap check. One example is checking the rescheduling timer; we can't allow the timer to interrupt a thread in mid-execution since it must leave everything in a tidy state so that garbage collection can be performed. On a heap-check failure, however, the thread is already prepared for a possible garbage collection so we can easily yield if the rescheduling timer has expired.

4.3 Out of band allocation

The following interface can be used when the sequential allocation scheme isn't appropriate (we'll give some examples below):

```
StgPtr allocate(int words)
Bool   doYouWantToGC(void)
```

The `allocate` function provides a chunk of memory of size `words`. It *always succeeds*. It seems that we are fooling ourselves and pretending that we've got an infinite supply of memory, but the idea is that `allocate` is only used in short bounded sections of (C or Haskell) code where garbage collection isn't convenient. At an appropriate point the application can call `doYouWantToGC` and ask the storage manager if it thinks a garbage collection is called for; if so the application should save away all live pointers on the thread stack and call the garbage collector.

A prime example is the GNU multi-precision integer library that we use to implement the Integer type in Haskell. The library allows the application to provide its own allocation interface, but we can't garbage collect while in the allocator because we don't know what live pointers the library function has hold of at any given time. It's possible to guess how much memory is needed beforehand, but this is a risky business and requires knowledge of the internals of the library. The solution is to use the `allocate` interface, and call `doYouWantToGC` on returning from the library call.

The `allocate` interface can also allocate objects larger than the block size, which is impossible with the sequential allocation scheme because the difference between `Hp` and `HpLim` is never larger than a block. So functions which need large chunks of memory, such as for an array or a thread stack, must use `allocate` to get their memory. Large allocations are fulfilled by asking the block allocator for a block group (a contiguous sequence of blocks) of the correct size, and then keeping track of the group on a list separate from the nursery. Large objects are never moved in memory; their block groups are simply chained onto a new list during garbage collection.

4.4 Garbage Collection

The garbage collector is normally called from the scheduler, but may be called by external C code provided the calling thread has saved away its state on the thread stack.

```
void        GarbageCollect(void (*get_roots)(void))
StgClosure *MarkRoot(StgClosure *p)
```

The garbage collector is invoked by calling `GarbageCollect` passing a pointer to a function which will be called back to find all the possible live roots. The reason for the callback is that the application, and not the storage manager, knows where all the possible roots are. When called from the scheduler, for example, the roots will normally consist of the various thread queues plus things like the stable pointer table.

The application-specific `get_roots` function calls `MarkRoot` for each possible root, which returns the new location of the object. The garbage collector will traverse the reachable graph from each root to find the closure of live data.

4.5 Support for generational GC

To support generational garbage collection the storage manager needs to know the location of objects than can refer to objects in newer generations. This can happen in two ways:

- An update overwrites a thunk in an older generation with an indirection to a new object, or
- a destructive write is performed to a mutable object in an older generation.

We provide two functions to indicate to the storage manager when one of these situations has occurred:

```
void UpdateWithIndirection(StgPtr p1, StgPtr p2)
void RecordMutable(StgPtr p)
```

When an object is to be updated, the actual update is performed by `UpdateWithIndirection`, which checks whether the object to be updated is in an old generation, and if so it must be marked for traversal during garbage collection. `UpdateWithIndirection` is implemented as a macro for speed.

There are two ways to cope with destructive writes in a generational garbage collector: either trap the write itself and mark the target object for later traversal, or keep track of all mutable objects in older generations and traverse them all during GC. We opt for the latter approach since the number of mutable objects is usually low, and we would like to keep the write operations fast and inline.

Only certain objects are explicitly mutable, and they are indicated as such by their object types. Hence the garbage collector can easily keep a record of all live mutable objects in the heap, except when an object is changed from being immutable to mutable. The `RecordMutable` operation is provided for this eventuality, which only occurs in one place in GHC: the `thawArray` function (the opposite of `freezeArray` which makes a mutable array immutable). An object that is recorded as mutable using this interface and subsequently becomes immutable will be automatically removed from the mutable list by the garbage collector.

4.6 Garbage Collecting CAFs

A CAF (constant applicative form) is a top-level expression with no arguments. The expression may need a large, even unbounded amount of storage when it is fully evaluated.

CAFs are represented by closures in static memory that are updated with indirections to objects in the heap once the expression is evaluated. Previous versions of GHC maintained a list of all evaluated CAFs and traversed them all during GC, the result being that the storage allocated by a CAF would reside in the heap until the program ended.

The problem with garbage collecting CAFs is that they can't be reached by following the live roots of the running program: we don't store pointers to static objects in dynamic memory because we already know at compile time where the object lives so there's no need to store pointers to it in the heap. The problem with this approach is that we can't know whether a CAF expression will ever be referenced again just by looking at the contents of the heap and stack, we have to look at the code itself.

The solution we adopted is to generate a *static reference table* (SRT) for each code segment in the program, containing pointers to all of the top-level functions and CAFs that it directly references. During garbage collection, we traverse all the objects referred to by the reachable SRTs, recursively to identify the reachable CAFs.

For example, each thunk in the heap will have some associated code which evaluates the thunk. This code will also have an SRT which records the top level functions and CAFs referenced by the code for the thunk (any non-top-level objects will be pointed to by the thunk in the heap). By recursively following the pointers in the SRT we can identify all the CAFs that can possibly be reached directly or indirectly by the thunk. Similarly for stack frames; however constructors are fully evaluated and hence have no SRTs. Each static object has a spare field which we use to link together all the reachable static objects and also stop the traversal from looping.

The scheme can be optimised by noting that many code sequences don't reference any static objects at all, and hence have an empty SRT. A top-level function with an empty SRT need not be referenced from other SRTs, possibly resulting in more empty SRTs, and so on. We have found by using this optimisation that most functions get away with an empty SRT, and non-empty SRTs generally have no more than a few entries, especially if the information is passed across module boundaries (which we do). The increase in code size has so far been minimal. There is a penalty at garbage collection time for following the SRTs, but it is offset by the saving in heap residency when CAFs can be garbage collected.

5 Execution model changes

The basic execution model that the runtime system supports is the STG machine [Jon92b]. However, the need to support mixed compiled/interpreted execution led us to make a number of changes to the original STG execution model. Specifically:

One stack. The original STG machine used two stacks, one for pointers and one for non-pointers. Segregating the pointers makes garbage collection easier, but it makes everything else more complicated. The compiler has to model two stacks; two registers are consumed as stack pointers; copying chunks of stack into separate heap objects is more complex (this happens when updating a function-valued thunk); and so on.

Our new model has a single stack, containing both pointers and non-pointers. The stack grows downwards, towards decreasing addresses. This makes it easy to think of an activation record as a kind of stack-allocated heap object: the return address (at the lowest address of the activation record) is like an info pointer, and it is furnished with an info table that describes the layout of the activation record. The garbage collector uses this information to identify the pointers and non-pointers in the activation record. The idea of attaching layout information to return addresses is not new, but we have found that thinking of an activation record as a stack-allocated, but otherwise normal, heap object, is quite helpful.

When calling an unknown function, it turns out to be necessary to tag each non-pointer explicitly ([Jon92b] gives an example that shows the problem). We do this by allocating an extra tag word on the stack for each such non-pointer. Fortunately, calls to unknown functions with unboxed arguments are pretty rare.

The single stack model is a bit more efficient than the two-stack model, because there is now no need to "stub" dead pointers — that is, overwrite them with a null pointer — in the pointer stack. Previously this was necessary to avoid space leaks, but now the descriptor in the info table can simply describe the dead stack slot as a non-pointer, and hence not to be treated as a root by the garbage collector.

Uniform return in heap convention for boxed values. The original STG machine had the nifty feature that a function returning (say) a pair would return the two components in registers. If the caller of the function was a `case` expression that took the pair apart, then the returned pair might never be allocated at all. This "return in registers" convention contrasts with the simpler "return in heap" approach of allocating the pair and returning a pointer to it. We have now reverted to this simpler approach.

The return-in-registers scheme looks superficially very attractive, but it has a number of undesirable consequences:

- Since garbage collection may strike, or an update may need to be performed, an info pointer has to be returned along with the components of the pair. This uses another instruction, and another register, and is only needed some of the time.
- The `case` expression might not use all of the components of the returned value — selectors have this property. This matters when an *existing* value is returned, because then the components are loaded into registers only to be discarded.
- Matters get complicated when switching back to interpreted execution; the latter certainly uses the simple return-in-heap convention, and impedance-matching is complicated.

Return-in-registers looks a particularly poor choice for register-impoverished architectures, such as the x86, an architecture for which we would particularly like good performance.

Using the return-in-heap convention means that we can no longer update thunks in place (except by copying the value, which is clearly a bad idea), because the returned value is already allocated. Necessity is the mother of invention: we have designed (but not implemented) a scheme that recovers update in place by returning unboxed tuples (only) in registers.

Better register usage. The STG also has a register-based calling convention, whereby calling a known function with the right number of arguments is done by loading the arguments into registers and jumping to the function's entry point. Registers were previously used for the first 8 arguments, which is a pessimisation when real machine registers are not available and these "pseudo-registers" are mapped to memory locations. In the new system, we adjust the calling convention based on the number of machine registers available; for instance on the x86 only the first argument to a function is passed in a register, the rest are passed on the stack.

Polymorphic case. Haskell 1.4 allows any value (whether a data value or a function value) to be evaluated to weak head normal form, using the `seq` operator. The execution model supports this by using a special sort of update frame.

5.1 Space leaks and black holes

A program exhibits a *space leak* if it retains storage that is sure not to be used again. Space leaks are becoming increasingly common in imperative programs that `malloc` storage and fail subsequently to `free` it. They are, however, also common in garbage-collected systems, especially where lazy evaluation is used. [Wad87, RW93]

Quite a bit of experience has now accumulated suggesting that implementors must be very conscientious about avoiding gratuitous space leaks — that is, ones which are an accidental artefact of some implementation technique[App92]. The update mechanism is a case in point, as [Jon92a] points out. Consider a thunk for the expression

```
let xs = [1..1000] in last xs
```

where `last` is a function that returns the last element of its argument list. When the thunk is entered it will call `last`, which will consume `xs` until it finds the last element. Since the list `[1..1000]` is produced lazily one might reasonably expect the expression to evaluate in constant space. But *until the moment of update, the thunk itself still retains a pointer to the beginning of the list* `xs`. So, until the update takes place the whole list will be retained!

Of course, this is completely gratuitous. The pointer to `xs` in the thunk will never be used again. In [Jon92b] the solution to this problem that we advocated was to overwrite a thunk's info with a fixed "black hole" info pointer, *at the moment of entry.* The storage management information attached to a black-hole info pointer tells the garbage collector that the closure contains no pointers, thereby plugging the space leak.

5.1.1 Lazy black-holing

Black-holing is a well-known idea. The trouble is that it is gallingly expensive. To avoid the occasional space leak, for every single thunk entry we have to load a full-word literal constant into a register (often two instructions) and then store that register into a memory location.

Fortunately, this cost can easily be avoided. The idea is simple: *instead of black-holing every thunk on entry, wait until the garbage collector is called, and then black-hole all (and only) the thunks whose evaluation is in progress at that moment.* There is no benefit in black-holing a thunk that is updated before garbage collection strikes! In effect, the idea is to perform the black-holing operation lazily, only when it is needed. This dramatically cuts down the number of black-holing operations.

How can we find all the thunks whose evaluation is in progress? They are precisely the ones for which update frames are on the stack. So all we need do is find all the update frames (via the `Su` chain) and black-hole their thunks right at the start of garbage collection. Notice that it is not enough to refrain from treating update frames as roots: firstly because the thunks to which they point may need to be moved in a copying collector, but more importantly because the thunk might be accessible via some other route.

5.1.2 Detecting loops

Black-holing has a second minor advantage: evaluation of a thunk whose value depends on itself will cause a black hole closure to be entered, which can cause a suitable error message to be displayed. For example, consider the definition

```
x = 1+x
```

The code to evaluate `x`'s right hand side will evaluate `x`. In the absence of black-holing, the result will be a stack overflow, as the evaluator repeatedly pushes a return address and enters `x`. If thunks are black-holed on entry, then this infinite loop can be caught almost instantly.

With our new method of lazy black-holing, a self-referential program might cause either stack overflow or a black-hole error message, depending on exactly when garbage collection strikes. It is quite easy to conceal these differences, however. If stack overflow occurs, all we need do is examine the update frames on the stack to see if more than one refers to the same thunk. If so, there is a loop that would have been detected by eager black-holing.

5.1.3 Lazy locking

In a parallel implementation, it is necessary somehow to "lock" a thunk that is under evaluation, so that other parallel evaluators cannot simultaneously evaluate it and thereby duplicate work. Instead, an evaluator that enters a locked thunk should be blocked, and made runnable again when the thunk is updated.

This locking is readily arranged in the same way as black-holing, by overwriting the thunk's info pointer with a special "locked" info pointer, at the moment of entry. If another evaluator enters the thunk before it has been updated, it will land in the entry code for the "locked" info pointer, which blocks the evaluator and queues it on the locked thunk.

The details are given by [THM+96]. However, the close similarity between locking and black holing suggests the following question: can locking be done lazily too? The answer is that it can, except that locking can be postponed only until the next *context switch*, rather than the next *garbage collection*. We are assuming here that the parallel implementation does not use shared memory to allow two processors to access the same closure. If such access is permitted then every thunk entry requires a hardware lock, and becomes much too expensive.

Is lazy locking worth while, given that it requires extra work every context switch? We believe it is, because contexts switches are relatively infrequent, and thousands of thunk-entries typically take place between each.

The scheme can be further optimised by only locking each thunk once: if, while traversing a thread's stack locking the thunks referred to by each update frame, we encounter a thunk which is already locked, we can safely assume that all thunks further down the stack have already been locked. This reduces the cost of lazy locking if context switches occur frequently.

6 The Scheduler

The scheduler manages the concurrent execution of multiple Haskell threads, and is the heart of the runtime system.

6.1 Concurrency model

A Haskell thread is meant to be an extremely lightweight execution engine, and there may be hundreds or thousands of them in a Concurrent Haskell program [PJGF96]. Furthermore, they all share and mutate a common heap. If each Haskell thread were an operating system thread, then every access to a heap object would have to be synchronised using relatively expensive OS support.

Accordingly, the scheduler runs as a single OS thread, and multiplexes this single execution thread among the runnable Haskell threads. Once started, a Haskell thread is never pre-empted between two arbitrary instructions. Instead, it is pre-empted only when it makes the next call to `ExtendNursery`; the `ExtendNursery` procedure inspects a global pre-emption flag. This may delay thread switching, but it does make everything very simple, and doesn't increase code size since the call to `ExtendNursery` is out-of-line in a generic heap-check failure fragment. Instruction sequences are therefore atomic by default, so no special support is required when mutating `MVars`. Requesting pre-emption is just a matter of setting a global variable. Threads are suspended only at a moment when they have tidied up their activation record, and left a suitable info pointer to describe the activation record on the stack.

A Haskell thread is represented by a heap-allocated *Thread State Object* (TSO). The TSO has room to save the register state at thread suspension, and also contains the thread's stack. If the stack overflows, we simply allocate a new TSO of double the size, and copy the old TSO into the new one.

6.2 The scheduler state

The scheduler state, together with any state maintained by the application (eg. Hugs), gives the entire state of the Haskell program. In particular, this state gives all garbage collection roots. It consists of:

- A queue of runnable threads.
- A queue of threads waiting for a timeout or blocked I/O operations.
- A queue of threads currently engaged in external C calls which called back to Haskell (see Section 6.5).
- A pointer to the stable pointer table.
- Roots from the application. For instance, Hugs maintains a symbol table which records the locations of the byte code objects for each loaded function.

6.3 The scheduler API

To facilitate calling Haskell functions from a variety of clients, the scheduler provides an API that lets a client create and execute Haskell expressions. Here it is:

```
void    startupHaskell  ( int argc, char *argv[] );
void    shutdownHaskell ( void );

void    getHaskellFlags ( int *argc, char **argv[] );
void    setHaskellFlags ( int  argc, char  *argv[] );

StgClosure *eval           ( StgClosure *closure );
StgClosure *evalIO         ( StgClosure *closure );

StgClosure *applyClosure   ( StgClosure *c, StgClosure *a );
StgClosure *applyInt       ( StgClosure *c, StgInt i );
StgClosure *applyFloat     ( StgClosure *c, StgFloat f );
StgClosure *applyAddr      ( StgClosure *c, StgAddr a );
...

StgInt      resultInt      ( StgClosure *c );
StgFloat    resultFloat    ( StgClosure *c );
StgAddr     resultAddr     ( StgClosure *c );
...
```

The whole Haskell system is started by invoking `startupHaskell`, with some optional command-line style arguments. This function initialises the scheduler and storage manager and prepares for execution. The flags allow control of various aspects of the system, such as the maximum heap size and allocation area used by the storage manager. The `getHaskellFlags` and `setHaskellFlags` allow the flag settings to be altered on the fly — this is useful in an interactive environment such as Hugs.

New computations can be run using the `apply..`, `eval..`, and `result..` set of functions. The type `StgClosure *` is a pointer to an abstract Haskell object; top-level closures in compiled modules have a well-defined naming scheme, for example function `foo` in module `Bar` is called `Bar_foo_closure`.

To simply evaluate a closure `c`, the application calls `eval(c)`, which returns a pointer to the result closure. Results may be unpacked using the `result..` family of functions.

More complex expressions can be built by adding arguments to the closure, using the `apply..` functions. For example, to evaluate the expression `[1..10]`, we might do

```
e = applyInt (applyInt Prelude_enumFromToInt_closure 1) 10
eval(e);
```

(if the translation of [1..10] for Int is enumFromToInt in module Prelude).

Haskell IO computations can be evaluated using the evalIO interface, which keeps the representation of the IO type abstract.

The simplest runtime system client, a standalone compiled Haskell binary, can now be defined as follows:

```
int main(int argc, char *argv[])
{
   startupHaskell(argc,argv);
   evalIO(Main_main_closure);
   shutdownHaskell();
   exit(0);
}
```

6.4 Switching between compiled and interpreted execution

The interpreter executes byte codes, which are stored in Byte Code Objects (BCOs) in the heap. BCOs are stored and garbage collected along with the rest of the heap; for example when a module is re-loaded by the interpreter, the storage used by the code for the old copy will be collected at the next GC.

How does a thread switch between compiled and interpreted execution? Suppose it is executing interpreted code, and is about to evaluate a value (thunk or function). The interpreter looks at the heap object; it if is a Byte Code Object (BCO), the interpreter moves the byte-code program counter to the start of the object, and continues execution. Otherwise, it returns to the scheduler, requesting compiled execution. The scheduler fires up compiled execution by loading machine registers with the heap pointer, stack pointer, and so on, and branching to the code for the thunk or function.

Now suppose that compiled code enters the code for a function or thunk. If it is a compiled function or thunk, then execution proceeds normally. If it is a BCO, the BCO's entry code simply returns to the scheduler requesting interpreted execution. This return first saves the machine registers in the thread's TSO, of course.

In practice, we can optimise these returns to the scheduler, but logically that is what happens. Similar things happen when returning to a return address on top of the stack, if one thinks of the return address as the info pointer of a stack-allocated heap object, as discussed in Section 5.

6.4.1 Dynamic loading of compiled code

Hugs will need to dynamically load and unload compiled modules during a session. When a module is loaded, its external references need to be updated to point to other compiled modules, or into the interpreter itself when the reference is to a module that is currently being interpreted. This requires some special support in the compiler, as well as the runtime system.

It seems inevitable that we will have to compile modules into special dynamically-linkable objects, which will impose some overhead as most jumps will have to go through a jump table. There are several dynamic linking schemes in use, but most of these are specific to a particular architecture, operating system or binary format. Furthermore, these schemes aren't flexible enough to allow us to link binary modules into a running interpreter, because the linker's symbol table can't be modified. Therefore we intend to implement our own portable dynamic linking scheme.

6.5 Callouts and Callbacks

Integration with existing languages and tools is of the utmost importance if GHC is to be used in the real world. Hence we'd like to provide methods by which Haskell programs can call foreign language routines (a callout), and external systems can invoke Haskell functions. The scheduler API described above caters for the latter requirement, but how do we allow arbitrary foreign calls from Haskell?

As a first guess, we could just perform the call inline in a Haskell function. For most applications, this works fine, although the entire Haskell system, including all running threads, is temporarily suspended until the foreign call returns. However, this approach could get us into trouble if the foreign callee decided to use parts of the GHC RTS - perhaps to request a garbage collection, or even worse, to call the scheduler (a callback).

If the callee wanted to perform a garbage collection, the caller (the Haskell thread) must make sure it has left the world in a sane state, saving away registers on the stack and making sure the garbage collector has access to all the live roots. This means some extra bookkeeping around this kind of external call, so we provide a special construct for it (currently `_ccall_GC_` to indicate that the callee may call the garbage collector).

If the callee wants to call the scheduler, perhaps creating new threads first, then we have another problem: the scheduler is already running, and we have two options:

- We could have the Haskell thread return to the scheduler, requesting that the foreign call be performed from there, possibly in another operating system thread. This means implementing some interpretive foreign calling code in the scheduler, which is extremely hairy and architecture/language dependent.
- We could make the scheduler re-entrant. This is not as problematic as it sounds, but it does involve some careful considerations:
  - The calling thread must be prepared for possible GC. Hence we require that a foreign call which may call back be done with `_ccall_GC_`.
  - The scheduler must detect when it is being re-entered, and save the current thread on a queue. The calling thread is then blocked until the new scheduler instance returns (all other threads may continue to run, though).

- The scheduler also has to be careful not to assume that local variables containing pointers are constant across a call to Haskell land: the Haskell code may now call the garbage collector or reschedule under our feet.

We plumped for the second option, because it was easier to implement. However, it does have one drawback — the original Haskell thread that performs the external call could be blocked longer than necessary. This happens when several nested computations are running, and an earlier call completes while in the context of a nested call.

7 Profiling

We regard the profiler as an extremely important aspect of our runtime system. Though not yet properly implemented, we plan to fully support Jarvis's cost-centre stack profiling [Jar96], itself based on our earlier work on cost centres [SPJ95], for both space and time. The idea is briefly as follows:

- Costs are attributed to a *stack* of *cost centres*. This stack corresponds closely to the dynamic chain of a *call-by-value* language. For example

```
f x = scc "f" (
  let y = scc "y" (g x)
  in scc "fbody" (h y)
)
```

  The costs of evaluating `(g x)` will be attributed to the stack `<y,f>`, while those of evaluating `(h y)` will be attributed to `<fbody,f>` (excluding the costs of evaluating `y`, of course).

- The execution model is quite simple. A machine register identifies the *current cost centre stack*, to which are attributed all costs. In particular, timer interrupts increment a statistics counter in the stack. When a thunk is built, the current cost centre stack is stored in the thunk; when the thunk is entered, that stored cost centre stack is made the current one.

- To keep the number of distinct cost-centre stacks tractable, Jarvis's ingenious idea is to have the following invariant: *any cost centre stack includes a given cost centre at most once*. So, when pushing a new cost centre on top of a stack, we first delete any existing occurrence of that cost centre in the existing stack. This sounds as if it might be an expensive operation, but it turns out to be quite simple to attach a memo table to each stack, which maps a cost centre to the new stack obtained by pushing that cost centre onto that stack.

- Statistics are dumped to a *log file* during execution. After execution, the log file can be examined with one or more *browsers*. We have two browsers in mind: Jarvis's call-graph browser, and a space-profile browser. In both cases, the browser allows views at different levels of detail without re-running the program. Of course, this relies on storing rather a lot of information in the log file; we are experimenting with flexible and compact log-file formats to make this not too onerous [JMJW98]. Even so, the size of the log files is a worry.

We plan that the same general framework will support parallel profiling too.

- The cost-centre stack scheme has useful consequences for debugging, too. When a program encounters a fatal error, such as a pattern matching failure or an infinite loop detected by a black hole entry, the information provided to the programmer normally consists of no more than the function which contained the error. A stack trace is next to useless, because the lazy evaluation order confuses the issue. However, the current cost centre stack at the time of the error is much more useful, as it represents what the programmer would normally think of as the call stack.

All of this is equally accessible to a Haskell program interpreted by Hugs, or compiled by GHC, or a mixture.

We would like to support more elaborate space profilers, such as lag, drag, void, and retainers [RR96a, RR96b], but all of them require compiler and runtime system support that we have not yet studied carefully.

References

[App92] A. W. Appel. *Compiling with Continuations.* Cambridge University Press, 1992.

[DBE93] R. Kent Dybvig, Carl Bruggeman, and David Eby. Guardians in a generation-based garbage collector. In *Proceedings of the ACM SIGPLAN '93 Conference on Programming Language Design and Implementation*, pages 207–216, Albuquerque, New Mexico, June 23–25, 1993. *SIGPLAN Notices,* 28(6), June 1993.

[DEB94] R. Kent Dybvig, David Eby, and Carl Bruggeman. Don't stop the bibop: Flexible and efficient storage management for dynamically types languages. Technical Report 400, Indiana University, Computing Science Dept., March 1994.

[Jar96] S A Jarvis. *Profiling large-scale lazy functional programs.* PhD thesis, Department of Computer Science, University of Durham, 1996.

[JMJW98] Steven A. Jarvis, Simon Marlow, Simon L. Peyton Jones, and Eric Wilcox. A grammar for self-describing documents. Technical Report PRG-TR-4-98, Oxford University, July 1998.

[Jon92a] Richard Jones. Tail recursion without space leaks. *Journal of Functional Programming*, 2(1):73–79, January 1992.

[Jon92b] Simon L. Peyton Jones. Implementing lazy functional languages on stock hardware: the Spineless Tagless G-machine. *Journal of Functional Programming*, 2(2):127–202, April 1992.

[LJ95] J. Launchbury and S. L. Peyton Jones. State in haskell. In *Lisp and Symbolic Computation*, volume 8, pages 293–342, Dec 1995.

[PJGF96] Simon Peyton Jones, Andrew Gordon, and Sigbjorn Finne. Concurrent Haskell. In *Conference Record of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'96)*, pages 295–308, St. Petersburg, Florida, January 21–24, 1996. ACM Press.

[Rep94] J. H. Reppy. A high-performance garbage collector for standard ml. Technical report, AT&T Bell Labs, 1994.

[RR96a] Niklas Röjemo and Colin Runciman. Lag, drag, void and use—heap profiling and space-efficient compilation revisited. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming*, pages 34–41, Philadelphia, PA, May 24–26 1996. ACM Press.

[RR96b] Niklas Röjemo and Colin Runciman. New dimensions in heap profiling. *Journal of Functional Programming*, 6(4):587–620, September 1996.

[RW93] Colin Runciman and David Wakeling. Heap profiling of lazy functional programs. *Journal of Functional Programming*, 3(2):217–245, April 1993.

[SPJ95] Patrick M. Sansom and Simon L. Peyton Jones. Time and space profiling for non-strict, higher-order functional languages. In *Conference Record of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'95)*, pages 355–366, San Francisco, California, January 22–25, 1995. ACM Press.

[THM+96] P. W. Trinder, K. Hammond, J. S. Mattson, Jr., A. S. Partridge, and S. L. Peyton Jones. GUM: A portable parallel implementation of Haskell. In *Proceedings of the ACM SIGPLAN '96 Conference on Programming Language Design and Implementation*, pages 79–88, Philadelphia, Pennsylvania, May 22–24, 1996.

[Wad87] Philip Wadler. Fixing a space leak with a garbage collector. *Software - Practice and Experience*, 17(9):595–608, 1987.