Under consideration for publication in J. Functional Programming

The GHC Runtime System

fill in

Abstract

(This paper describes the implementation of the GHC runtime system)

Contents

1 Introduction

The GHC runtime system is perhaps one of the most unusual language runtimes in wide use today. Much of its implementation was directly motivated by the unusual (by popular

programming language standards!) features in Haskell which the runtime needs to support. Some of these features, such as lazy evaluation and ubiquitous concurrency, complicate the design of the system and require us to do a lot more work to give programmers the high-level behavior they desire. Other features, such as an emphasis on pure computation without mutation, simplify the design components such as garbage collection and software transactional memory.

Complete, flesh out, rewrite

SM: This is a good start. I think it will help to tell the reader up front that we intend to focus on the main novel aspects of the runtime how concurrency and parallelism, the garbage collector, and lazy evaluation work. Everything else we say will be stuff that you need to know in order to understand these three main topics, and the order of the paper will reflect this - talking about the essential aspects of the evaluation model and data representations first, because you need that to understand the rest.

SM: Please feel free to label sections that you would like me to write.

2 Storage

An essential component of a runtime system for any high-level programming language is the garbage collector, which is responsible identifying and reclaiming memory from objects which are no longer in use by the program. When it comes to a garbage collector, *efficiency* is the order of the day: the speed of the garbage collector affects the performance of all programs running on the runtime, and thus the GHC runtime devotes a substantial portion of its complexity budget to a fast garbage collector. What do we need for a fast garbage collector?

2.1 Blocks

The very first consideration is a low-level one: "Where is the memory coming from?" The runtime can request memory from the operating system via `malloc`, but how much should it request, and how should it be used? A simple design is to request a large, contiguous block of memory and use it as the heap, letting the garbage collector manage objects allocated within it. However, this scheme is fairly inflexible: when one of these heaps runs out of memory, we need to double the size of the heap and copy all of the old data into the new memory. Picking the initial sizes of heaps can be an exercise in fiddling with black magic tuning parameters.

A more flexible scheme is a *block-structured heap* (Steele, 1977; Dybvig *et al.*, 1994; Marlow *et al.*, 2008). The basic idea is to divide the heap into fixed-size B-byte blocks, where B is a power of two: blocks are then linked together in order to provide memory for the heap.[1] These blocks need not be contiguous: thus, if your heap runs out of space, instead of having to double the size of your heap, you can simply chain a few more blocks onto it. There are some other benefits as well:

[1] GHC uses 4kb blocks, but this is an easily adjustable constant.

1. Large objects (e.g. close to block size) do not have to be copied from one region to another; instead, the block they reside in can be relinked from one region to another.[2]
2. Blocks make it easy to provide heap memory in contexts where it is not possible to perform garbage collection. As an example, consider the GMP arbitrary-precision arithmetic library. This C code requires the ability to allocate memory while performing internal computation. However, if the heap runs out of memory, what can you do? If the heap were contiguous, you would now need to carry out a GC to free up some memory (or get a larger heap); but this would require us to halt the C code (arbitrarily deep in some internal computation) while simultaneously being able to identify all pointers to the heap that it may be holding. Whereas in a block-structured heap, we can simply grab a new block and defer the GC until later.
3. Free memory can be recycled quickly, since a free block can be quickly reused somewhere else.

One reason why this scheme works so well is that most objects on the heap are much smaller than the block size; handling these cases is very simple. When an object is larger than a block size, it needs to be placed into a *block group* of contiguous blocks—which in turn need to be handled with some care to avoid fragmentation. The blocks themselves are provided by the operating system in large units called *megablocks*.[3]

Finally, each block is associated with a *block descriptor*, which contains information about the block such as what generation it belongs to, how full it is, what block group it is part of, etc. An obvious place to put the block descriptor is at the beginning of a block, but this runs into problems when the block is the member of a block group (the memory must be contiguous!) Thus, the descriptors of blocks of a megablock are instead organized together in the first block of a megablock; some care is taken to ensure that the runtime can efficiently compute the block descriptor of any given block.

2.2 Memory layout

Before we can discuss the garbage collector proper, we have to describe the layout of the data that is to be garbage collector. GHC has a uniform representation for objects on the heap with a *header*, which indicates what kind of object the data is, and a *payload*, which contains the actual data for an object (e.g. free variables for function values and thunks, or fields for data values). The header points to an *info table*, which provides more information about what kind of object the closure is, what code is associated with the object and what the layout of the payload is (e.g. what fields are pointers.)

The presence of info tables makes it easy for the garbage collector to determine what other closures on the heap an object may reference, as it says which fields in the payload are pointers. Essentially everything that the GC touches has an info table, including the stack frames (each block of compiled code receives its own info table.) In particular, this makes calculating the *roots* (base objects which are always considered reachable) of the

[2] Of course, this requires *only* one object to live in a block, which can result in fragmentation. However, empirically this does not seem to have caused much of a problem.

[3] 1Mb in size, in the current implementation.

application much simpler: at the beginning of any block of code, the info table and registers (which known and saved by the code itself) constitute all of the pointers in use, allowing us to accurately perform GC.

There are many possible methods by which objects on the heap can be represented (for example, in place of info tables, pointer tagging can be used to distinguish non-pointers from pointers). However, in order to support lazy evaluation, headers have another important function, which is that they double as pointers to the *entry code* responsible for evaluating a thunk. The ability to *replace* one header with another and have the behavior of a thunk change correspondingly is tremendously useful. The issues are discussed in Section **??**.

2.3 Generational garbage collection

The next question you might ask is, "What kind of garbage collector should I use?" By default, GHC uses a generational copying collector.

A *generational collector* divides the heap into *generations*, where generations are numbered with the zero being the youngest. Objects are allocated into the youngest generation, which has garbage collection performed on it whenever it runs out of memory. Surviving objects are *promoted* to the next generation, which is collected less frequently, and so forth. In a *copying collector*, this promotion occurs by simply copying the object into the *to-space*, a process called *evacuation*. Evacuated objects are subsequently *scavenged* to find what objects they refer to, so they can be evacuated as well.

The efficacy of generational collection hinges on the "generational hypothesis", which states that data that has been recently allocated is the most likely to die and become unreachable. This tends to be particularly true of functional programs, which encourage the use of short-lived intermediate data structures to help structure computation. In fact, functional programs allocate so much memory that it makes sense not to immediately promote data, since objects may not have had sufficient chance to die by the time of the first GC. Thus, GHC further implements an *aging* scheme, where reachable objects in generation 0 are not immediately promoted to generation 1; instead, they are aged and promoted the next GC cycle.[4]

Use of a copying collector has other benefits for allocating heavy workloads. In particular, copying collection ensures that free memory is contiguous, which allows for extremely efficient memory allocation using a *bump allocator*—so named because a heap allocation simply involves bumping up the free space pointer. Additionally, while copying collectors are often criticized for wasting half of their allocated memory to maintain the two spaces for copying, a block structured heap can immediately reuse blocks in the from-space as soon as they are fully evacuated.

[4] When objects are only aged once, an equivalent way of stating this scheme is that generation 0 is split into two generations, but we never garbage collect just the younger generation, we always collect both on a minor collection. This is in fact how GHC implements aging.

2.3.1 Mutability in the GC

The primary complication when implementing a generational garbage collector is the treatment of mutable references. When a heap is immutable, pointers in the young generation can only ever point into older generations; thus, to discover all reachable objects when collecting an old generation, it suffices to simply collect all younger generations when performing an (infrequent) collection of an older generation. However, if objects in the old generation are mutable, they may point back into the young generation, in which case we need to know that those objects are reachable even when the only references to them are in the old generation (which we would like to avoid collecting).

The solution to this problem is to apply a *GC write barrier* (sometimes confusingly referred to as a *write barrier*) to memory writes, adding the mutated object to a *remembered set* which is also considered a root for garbage collection. Now the GC cannot accidentally conclude an object pointed to by a mutable reference in an old generation is dead: it will discover its reachability through the remembered set. However, this scheme is costly in two ways: first, all mutation must pay the overhead of adding the object to the remembered set, and second, as the remembered set increases in size, the amount of heap that must be traversed during a minor collection also increases.

In the first case, GHC keeps track of mutation per object, spending a single memory write to add a mutated object to a mutable list. This design lies in a continuum of precision: one could increase the precision of the remembered set by only adding mutable fields (rather than objects), or one could decrease the precision by only tracking *cards* (i.e. portions of the heap) at a larger granularity. Increased precision increases the overhead of the mutable list but reduces the amount of extra work the GC needs to perform, while reduced precision makes mutation more efficient but leads to slower minor collections. We think that mutation per object is a good balance: mutation is not prevalant enough in functional code that coarse-grained card making buys much, and most mutable objects in Haskell are quite small, with only one or two fields.[5]

In the second case, GHC can take advantage of an interesting property of lazy functional programs: thunks are only ever mutated once, in order to update them with their fully evaluated values—they are immutable afterwards. Thus, we can immediately eliminate an updated thunk from the mutable list by *eagerly promoting* the data the updated thunk points to into the same generation as the thunk itself. Since the thunk is immutable, this data is guaranteed not to be GC'd until the thunk itself is GC'd as well. This leads to an interesting constraint on how garbage collection proceeds: we must collect older generations first, so that objects we may want to promote have not been evacuated yet. Because an already evacuated object may have forwarding pointers pointing to it, it cannot be evacuated again within the same GC.

[5] However, this assumption has caused the GHC runtime some grief, e.g. in the case of mutable arrays of pointers, which we used to scan the entirety. Today, we have a card-marking scheme to permit mutable arrays to be efficiently GC'd.

2.4 Parallel garbage collection

A generational garbage collector offers quite good performance, but there is still the question, "Is it fast enough?" One avenue for speeding up the garbage collector when multiple cores are available is to perform *parallel garbage collection*, having multiple threads traverse the heap in parallel. Note the distinction from *concurrent collection*, where the GC runs concurrently with user code which is mutating values on the heap. GHC implements parallel collection (Marlow *et al.*, 2008) but not concurrent collection: concurrent collection requires synchronization between the GC and the mutator and consequently is more complex. However, we have experimented with a form of concurrent collection in which individual cores have local heaps that can be collected independently of activity on the other cores (**?**).

There are two primary technical challenges that accompany building a parallel garbage collector. The first is how to divide the GC work among the threads, the second is how to synchronize when two GC threads attempt to evacuate the same object.

GHC overcomes the first challenge by utilizing the block structure of the heap. In particular, a block in the to-space of a garbage collection constitutes a unit of work: a thread can either claim the block to scavenge for itself, or the block can be transferred to another thread to process. Once blocks are chosen as the basic unit of work, there are a variety of mechanisms by which work can be shared: blocks can be statically assigned to GC threads with no runtime load balancing, blocks can be taken from a global queue which provides blocks to all threads, or a hybrid solution can have threads have local queues, but permit other threads to steal work from other queues when they are idle via a *work-stealing queue structure*. (Arora *et al.*, 1998) GHC originally implemented a single global queue, but we have since switched work-stealing queues because they have much better data locality, as processors prefer to take work from their local queues before stealing work from others. In fact, we don't want to do any load-balancing on minor collections, because it ruins locality by shipping work off to another core when the data is likely *already* in the cache of the original core. (Marlow *et al.*, 2009)

The second challenge reflects the primary cost of parallel GC, which is the extra synchronization overhead any parallel scheme will impose. In particular, we must prevent the GC from duplicating mutable objects when multiple threads attempt to evacuate the same object by synchronizing the object (by either locking it pessimistically or compare-and-swapping optimistically). This synchronization is expensive; fortunately, there is a wonderful benefit for immutable objects: they require no synchronization, because it is safe to have multiple copies of an immutable data structure! Eliminating the locks in these cases accounts for a 20-30% speedup, which is nothing to sneeze at.

One important parameter which must be set properly is the size of the young generation, i.e. the nursery. If the nursery is too small, then we will need to perform minor garbage collections too frequently. But if it is too large, then cores will generate a lot of memory traffic getting data that is not in their cache. In general, memory bandwidth is the bottleneck when multiple cores are allocating quickly; thus, having the nursery be the size of the cache is generally the best setting. But perhaps see the discussion at

`http://donsbot.wordpress.com/2010/07/05/ghc-gc-tune-tuning-haskell-gc-settings-for-fun-and-`

2.5 Summary

SM: Not sure whether going into this much detail is helpful to the average reader. I'm prepared to be persuaded otherwise.

We conclude by sketching the overall operation of GHC's parallel, generational, block-structured garbage collector, with all of the features that we have described thus far.

The main garbage collection function `GarbageCollect` in `rts/sm/GC.c` proceeds (after all user execution is halted) as follows:

1. Prepare all of the collected generations by moving their blocks into the from-space and throwing out their mutable lists (recall the remembered set is only necessary when the generation is not being collected.) The blocks themselves indicate what generation live objects in them should be promoted to.
2. Wakeup the GC threads, initializing them with eager promotion enabled.
3. *Evacuate* the roots of application (including the mutable lists of all older generations), giving work to the main GC thread to do.
4. In a loop, each thread:
   (a) Look for local blocks to *scavenge* (e.g. if the thread recently evacuated some objects which it hasn't scavenged), starting with blocks from the oldest generation.
   (b) Try to steal blocks from another thread (at the very beginning of a GC, idle GC threads are likely to steal work from the main thread, if they didn't have any work to begin with).
   (c) Halt execution, but while there are still GC threads running, poll to see whether or not there is any work to do.
5. Cleanup after the GC, which includes running finalizers, returning memory to the operating system, resurrecting threads XXX, etc.

The *evacuate* function `evacuate` in `rts/sm/Evac.c` takes a pointer to an object and attempts to copy it into a destination generation to-space, as specified by the block it resides in (or the generation that it needs to be promoted to, if eager promotion is enabled). Before doing so, it performs the following checks:

1. Is the object heap allocated? (If not, it is handled specially.)
2. Was the object already evacuated (e.g. the pointer already points to a to-space, or the object is a forwarding pointer)? If it was, and to a generation which is younger than the intended target, then it reports the evacuation as failed (so scavenge can add a mutable reference pointing to the object to a mutable list, etc.)

After the copy, the original is overwritten with a forwarding pointer. If the object in question is mutable, this is done atomically with a compare-and-swap to avoid races between two threads evacuating the same object.

The *scavenge* function `scavenge_block` in `rts/sm/Scav.c` walks a pointer down the provided block (filled in previously by evacuate), reading the info table in order to determine what kind of object it is. It evacuates the fields of the object, temporarily turning off eager promotion if the object is mutable. If evacuation is unsuccessful for the field of a mutable object, it must be added back to the mutable list. When the block is finished being scavenged, it gets pushed to the list of completed blocks. The block that is scavenged

can be thought of as the "pending work queue"; this optimization was first suggested as part of Cheney's algorithm and avoids the need for an explicit queue of pending objects to scavenge.

2.6 Further reading

While we have discussed many of the most important features of GHC's garbage collector, there remain many other features we have not discussed here. These include:

- an implementation of a compacting collector, no docs about this!
- support for weak pointers and finalizers, (Peyton Jones *et al.*, 2000) We might actually want to talk about this, it is probably one of the more voodoo-y parts of the system and
- garbage collection of static objects.

We have also omitted many details about the features we have discussed. For a good account of the block-structured parallel garbage collector, please see (Marlow *et al.*, 2008); however, since the paper was published the default locking and load balancing schemes for the parallel GC have changed, and we have implemented the improvement described in Section 7.1. Additionally, the GHC Commentary (Marlow, 2013) has good articles for technically inclined GHC hackers on a variety of issues we have discussed here, including eager promotion, remembered sets ete etc

3 Concurrency and parallelism

We now turn our attention to the implementation of concurrency (Peyton Jones *et al.*, 1996) and parallelism (Harris *et al.*, 2005) in the GHC runtime. It is well worth noting the difference between concurrency and parallelism: a parallel program uses multiple processing cores in order to speed up computation, while a concurrent program simply involves multiple threads of control, which notionally execute "at the same time" but may be implemented merely by interleaving their execution.

GHC is both concurrent and parallel, but many of the features we describe are applicable in non-parallel but concurrent systems (e.g. systems which employ cooperative threading on one core): indeed, some were developed before GHC had a shared memory multiprocessor implementation. Thus, in the first section, we consider how to implement *threads* without relying on hardware support. We then describe a number of inter-thread communication mechanisms which are useful in concurrent programs (and say how to synchronize them). Finally describe what needs to be done to make *compiled* code thread-safe.

3.1 Threads

Concurrent Haskell (Peyton Jones *et al.*, 1996) exposes the abstraction of a *Haskell thread* to a programmer. As the operating system also provides native threads, one may wonder if there any difference between a Haskell thread and an OS thread.

Many languages with multithreading support simply expose OS threads: a "thread" is one and the same as an OS thread. While simple, this approach has costs. In particular, users

of these languages must be economical in their use of threads, as most operating systems cannot support thousands of simultaneous OS threads. This is a shame, because in many applications the most natural way to *structure* a program involves thousands of logical threads: consider a web server, for example, which logically has a thread of execution per request. Furthermore, when a program cannot be made thread-safe and must be run in a single OS thread, no concurrency is available; it must be implemented in userspace, as is the case with many asynchronous IO libraries.

To support cheap threads, we must *multiplex* multiple Haskell threads onto a single OS thread. A Haskell thread runs until it is *preempted*, at which point we suspend its execution and switch to running another thread, an operation handled by the *scheduler*. How is this preemption implemented? True preemption is difficult to implement, because it implies we can interrupt the execution of code any any point, even if it would be leaving some data structures in an intermediate state. So instead, compiled Haskell code yields cooperatively[6] at various safe points, where we know that the heap is in order and our internal state is saved (to be restored on resumption). These points are in fact when Haskell code performs a *heap check* to find out if there is enough free memory on the heap to perform an allocation. These checks automatically are safe, because Haskell code already needs to be able to yield to the garbage collector, in case we have run out of memory. This check is extended to also yield when a thread has been preempted.[7]

Once we have a way of suspending and resuming threads, the scheduler loop is quite simple. Maintain a *run queue* of threads, and repeatedly:

1. Pop the next thread off of the run queue,
2. Run the thread until it yields,
3. Check why the thread exited:
   - If it ran out of heap, call the GC and then run the thread again;
   - If it ran out of stack, enlarge the stack and then run the thread again;
   - If it was preempted, push the thread to the end of the queue;
   - If the thread exited, continue.

3.2 Foreign function interface

The benefit of lightweight concurrency is that it offers a way of producing threads of control which are indistinguishable from OS threads, but are dramatically cheaper. However, there are some places when this illusion does not hold: of particular note is the *foreign function interface* (FFI) (Marlow & Jones, 2004), which permits Haskell code to call out to foreign code not compiled by GHC, and vice versa. What happens if an FFI call blocks? As our concurrency is cooperative, if an FFI call refuses to return to the scheduler, the execution of all other threads will grind to a halt. There isn't really any way around this

[6] Cooperative in the sense that the compiled code has explicit yield points, not in the sense that the code a programmer has to write contains yield points.

[7] In practice, preemption is handled by way of a timer signal, which, when fired, sets the heap limit to zero, triggering a faux "heap overflow" which the scheduler can then identify as a preempt and not a true request for garbage collection. Thus, the heap check and preemption check is a single conditional branch.

problem without introducing true OS threads to the mix: what we'd like to do is arrange for the blocking FFI call and the scheduler to run on different OS threads concurrently.

As it turns out, it is simpler to move the scheduler to a different OS thread than the FFI. Thus, we decompose OS threads into two parts: the OS thread itself (called a *task* in GHC terminology), and the Haskell execution context (called a *capability*). The Haskell execution context contains the scheduler loop and is responsible for the contents of the run queue: when executing, it is owned exclusively by the particular task (OS thread) which is running it. A single-threaded Haskell program has one capability: the capability is a *global lock* on the Haskell runtime. Now, before a blocking FFI call is made, the task releases the capability: if there is another idle *worker thread*, it can acquire the now free capability and continue running Haskell code.

When an FFI call returns, we'd like to return the capability to the original OS thread. Thus, we have to modify the scheduler loop as follows:

1. Check if we need to yield the capability to some other OS thread, e.g. if an FFI call has just finished,
2. *Run as before.*

Another place where Haskell threads differ from OS threads is thread local state. As capabilities are passed around OS threads, we make no guarantee that any given FFI call will be performed on the same OS thread as the previous FFI call. To accomodate Haskell threads which rely on thread-local state, Haskell introduces a *bound thread*, which binds a Haskell thread to a fresh OS thread.[8]

How can we support bound threads? A simple scheme is to give each bound thread its own OS thread. However, if we have only one capability, we need to coordinate these new OS threads so that only one is running Haskell code at a time. We can do this by, once again, passing the capability to whichever OS thread truly needs to run the bound thread:

1. *Check if we need to yield the capability to some other OS thread, e.g. if an FFI call has just finished,*
2. *Pop the next thread off of the run queue,*
3. Check if the thread is bound:
   - If the thread is bound but is already scheduled on the OS thread, proceed.
   - If the thread is bound but on the wrong OS thread, give the capability to the correct task.
   - If the thread is not bound but this OS thread is bound, give up the capability, so that any capability that truly needs this OS thread will be able to get it.
4. *Run as before.*

While the movement of capabilities from task to task is somewhat intricate, it imposes no overhead when bound threads are not used.

[8] GHC also calls these *in-calls*, due to the fact that external code which calls into Haskell must be bound: if it makes the Haskell code calls out via the FFI again, the inner and outer C code may rely on the same thread local state.

3.3 Load balancing

Assuming that the compiled Haskell code is thread safe (see Section 4.2), it is now very simple to parallelize execution: allocate multiple capabilities! Each OS thread in possession of a capability runs the scheduler loop, and everything works the way you'd expect.

There is one primary design choice: should each capability have its own run queue, or should there be a single global run queue? A global run queue avoids the need for any sort of load balancing, but requires synchronization and makes it difficult to keep Haskell threads running on the same core, destroying data locality. With separate run queues, however, threads must be load balanced: one capability could accumulate too many threads while the other capabilities idle.

The very simple load balancing scheme GHC uses is as follows: when a capability runs out of threads to run, it suspends itself (releasing its lock) and waits on a condition variable. When a capability has too many threads to run (it checks each iteration of its schedule loop), it takes out locks on as many idle capabilities as it can and pushes its excess threads onto their run queues. Finally, it releases the locks and signals on each idle capabilities that they can start running. The benefit of this scheme is that the run queues are kept completely unsynchronized, but a plausible alternative is to use work-stealing queues.

3.4 Sparks

Sparks (Marlow *et al.*, 2009) are a mechanism for speculative parallel computation. When a program is not utilizing all of its CPUs, the other CPUs can be reallocated to evaluate thunks that the programmer indicated are likely to be needed in the future. These units of work are called sparks, and they offer a mechanism for cheap, deterministic parallelism—in contrast to Haskell threads, which are more expensive and nondeterministic.

Sparks take advantage of Haskell's lazy evaluation (Section **??**) to provide a source of units of work for sparking. The ability to speculatively evaluate thunks—a spark is not guaranteed to be evaluated—comes from the fact that thunks encapsulate pure code and have no side effects. Furthermore, as thunk update is thread safe (Section 4.2), they provide a natural mechanism of communicating the result of a computation back to the thread that requested it.

Given thread-safe thunks, the implementation of sparks is quite simple: when a computation is sparked, it is stored in a *spark pool* associated with a capability. When a capability has no work to do (e.g. its run queue is empty), it creates a *spark thread*, which repeatedly attempts to retrieve a spark from the capability's spark pool and evaluate it. Thunk update ensures the results get propagated back to the main thread. If the capability receives any real work, it immediately terminates the spark thread.

Whereas threads rarely need to be load balanced, sparks frequently need to be migrated, as the capability that is generating sparks is likely to be the one that is also doing real work. Sparks are balanced using bounded work-stealing queues (Arora *et al.*, 1998; Hendler *et al.*, 2005), where a spark thread goes and steals sparks from other threads when it has none to execute.

One important optimization for sparks is removing sparks which will not contribute any useful parallelism. For example, if the spark is evaluated in the main thread before a spark

thread gets around to it, the spark is *fizzled* and should be removed from the spark pool. Additionally, if a spark's thunk has no references to it (i.e. is dead), then the result cannot possibly have an impact on program execution and it should also be pruned. It is relatively easy to check for both of these conditions in the garbage collector, by traversing the spark pool and checking if the spark points to a thunk that was successfully evacuated.[9]

3.5 MVars

Haskell offers a variety of ways for Haskell threads to interact with each other. We now describe how to implement MVars, the simplest form of synchronized communication available to Haskell threads. An MVar is a mutable location that may be empty. There are two operations which operate on an MVar: `takeMVar`, which blocks until the location is non-empty, then reads and returns the value, leaving the location empty, and `putMVar`, which dually blocks until the location is empty, then writes its value into location. An MVar can be thought of as a lock when its contents are ignored.

The blocking behavior is the most interesting aspect of MVars: ordinarily, one would have to implement this functionality using a condition variable. However, because our Haskell threads are not operating system threads, we can do something much more lightweight: when a thread realizes it needs to block, it simply adds itself to a *blocked threads queue* corresponding to the MVar. When another thread fills in the MVar, it can check if there is a thread on the blocked list and wake it up immediately.

This scheme has a number of good properties. First, it allows us to implement efficient *single wake-up* on MVars, where only one of the blocking threads is permitted to proceed. Second, using a FIFO queue, we can offer a fairness guarantee, which is that no thread remains blocked on an MVar indefinitely unless another thread holds the MVar indefinitely. Finally, because threads are garbage collected objects, if the MVar a thread is blocking on becomes unreachable, *so does the thread.* Thus, in some cases, we can tell when a blocked thread is deadlocked and terminate it.

3.6 Asynchronous exceptions

MVars are a cooperative form of communication, where a thread must explicitly opt-in to receive messages. Asynchronous exceptions (Marlow *et al.*, 2001), on the other hand, permit threads to induce an exception in another thread without its cooperation. Asynchronous exceptions are much more difficult to program with than their synchronous brethren: as a signal can occur at any point in a program's execution, the program must be careful to register handlers which ensure that any resources are released and invariants are preserved. In *pure* functional programs, this requirement is easier to fulfill, as pure code can always be safely aborted. Asynchronous exceptions are quite useful in a variety of situations, including timing out long running computation, aborting speculative computation and handling user interrupts.

[9] An alternate design is to have sparks be GC roots, so that an outstanding spark keeps its data alive. While this is convenient for the implementation of *parallel strategies*, it can result in space leaks, and GHC no longer uses this strategy.

Triggering an asynchronous exception is relatively simple with preemptive scheduling: force the target thread back to the scheduler, at which point the scheduler can introduce the exception and walk up the stack, looking for exception handlers. In case a thread is operating in a sensitive region, an exception masking flag can be set, which defers the delivery of the exception (it is saved to a list of waiting exceptions on the thread itself).

There are two primary differences between how asynchronous exceptions and normal exceptions are handled. The first is that a thread which is messaged may have been blocking on some other thread (i.e. on an MVar); thus, when an asynchronous exception is received, the thread must remove itself from the blocked list of threads.[10]

This explanation probably still needs a little work

The second difference is how lazy evaluation is handled. When pure code raises an exception, referential transparency demands that any other execution of that code will result in the same exception. Thus, while we are walking up the stack, when we see an *update frame*, which is a continuation responsible for taking a value and saving it to the thunk (overwriting it), we go ahead and instead overwrite the thunk with a new thunk that always throws the exception. However, in the case of an asynchronous exception, the code could have simply been unlucky: when someone else asks for the same computation, we should simply resume where we left off. Thus, we instead *freeze* the state of evaluation by saving the current stack into the thunk. (Reid, 1999) This involves walking up the stack and performing the following operations when we encounter an update frame:

1. Allocate a new closure (called an `AP_STACK` closure, for "apply stack") which contains the contents of the stack above the frame, and overwriting the old thunk[11] with a pointer to this closure,
2. Truncate the stack up to and including the update frame, and
3. Push a pointer to the new `AP_STACK` closure onto the stack.

The result is a chain of `AP_STACK` closures, where the top of each frozen stack links to the next frozen stack. When another thread evaluates an `AP_STACK` closure (intending to evaluate the thunk), it pushes the frozen stack onto the current stack, thus resuming the computation.

3.7 STM

Software Transactional Memory, or STM, is an abstraction for concurrent communication which emphasizes transactions as the basic unit of communication. The big benefit of STM over MVars is that they are composable: while programs involving multiple MVars must be very careful to avoid deadlock, programs using STM can be composed together effortlessly.

Before discussing what is needed to support STM in the runtime system, it is worth mentioning what we do not have to do. In many languages, an STM implementation must

[10] If your queues are singly linked, you will need some cleverness to entries. GHC does this by stubbing out an entry with an indirection, the very same that is used when a thunk is replaced with its true value, and modifying queue handling code to skip over indirections; because blocking queues live on the heap, the garbage collector will clean it up for us in the end.

[11] Possibly a black hole.

also manage all side-effects that any code in a transaction may perform. In an impure language, there may be many of these side-effects (even if they are local), and the runtime must make them transactional at great cost. In Haskell, however, the type system enforces that code running in an STM transaction will only ever perform pure computation or explicit manipulation of shared state. This eliminates a large inefficiency that plagues many other STM implementation.

How is it implemented

Maybe move this below messages and white holes

3.8 Messages and white holes

In the descriptions above, we said very little about the synchronization that is necessary to implement them in a multiprocessor environment. Under the hood, the GHC runtime has two primary methods of synchronization: *messages* and *white holes* (effectively a spin-lock). The runtime makes very sparing use of OS level condition variables and mutexes, since they tend to be expensive.

GHC uses a very simple message passing architecture to pass messages between capabilities. A capability sends a message by:

1. Allocating a message object on the heap;
2. Taking out a lock on the message inbox of the destination capability;
3. Appending the message onto the inbox;
4. Interrupting the capability, using the same mechanism as the context switch timer (setting the heap limit to zero); and
5. Releasing the lock.

This allows the message to be handled by the destination capability at its convenience, i.e. after the running Haskell code yields and we return to the scheduling loop. In general, the benefit of message passing systems is that they remove the need for synchronizing any of the non-local state that another capability *might* want to modify: instead, the capability just sends a message asking for the state to be modified.

When sending a message is not appropriate, e.g. in the case of synchronized access to closures which are not owned by any capability in particular, GHC instead uses a very simple spinlock on the closure *header*, replacing the header with a *white hole* header that indicates the object is locked. If another thread enters the closure, they will spinlock until the original header is restored. A spinlock is used as the critical regions they protect tend to be very short, and it would be expensive to allocate a mutex for every closure that needed one.

We can now describe how MVars and asynchronous exceptions are synchronized. An MVar uses a white hole on the MVar itself to protect manipulations of the blocked thread queue; additionally, when it needs to wakeup a thread, it may need to send a message to the capability which owns the unblocked thread. An asynchronous exception is even simpler: it is simply a message to the capability which owns the thread.

3.9 Summary

Haskell *threads* are lightweight threads of execution which multiplex onto multiple CPU cores. Each core has a *Haskell execution context* which contains a scheduler for running these threads; in order to handle FFI calls execution contexts can migrate from core to core as necessary. Threads are load balanced across execution contexts by having execution contexts with work push threads to contexts which don't. Sparks are a simple way of utilizing idle cores when there is no other real work to do.

By in large, all inter-thread communication in Haskell is explicit, thus making it much easier to compile Haskell in a thread-safe way. *MVars*, *asynchronous exceptions* and *STM* are explicitly used by Haskell code and can be efficiently implemented by taking advantage of our representation of Haskell threads. The basic techniques by which these are synchronized are *messages* and *white holes* (spinlocks). We defer the issue of synchronizing lazy evaluation to the next section.

3.10 Further reading

We have said little about how to use the concurrency mechanisms described here. XXX

4 Lazy evaluation

Lazy evaluation is an evaluation strategy which delays the evaluation of an expression until its result is requested. Operationally, an expression does not result in a computation, but rather results in the allocation of a *thunk*, which can be forced in order to produce the true result of the expression. This result is then written over the thunk, so that future accesses to the thunk avoid repeated evaluation.

There are a number of different strategies for implementing lazy evaluation. Their primary difference lies in who is responsible for updating the thunk: is it the evaluator of the thunk, or the thunk itself? The former constitutes the *cell model*, where the call-sites of thunks check if the thunk is evaluated before calling in and write back the result on return; the latter constitutes the *self-updating model*, where call-sites unconditionally jump to (or *enter*) the thunk, leaving the thunk responsible for updating itself. GHC utilizes a self-updating model, partially a historical artifact from when *all* objects on the heap (including data constructors) were expected to be evaluated by entering them, and because it works well with dynamic pointer tagging (Section 4.1), which avoids performing the jump for evaluated thunks at the cost of a cheap register comparison (no memory access).

Most languages give very little thought to the efficient implementation of thunks, since they are not a core part of the language. However, Haskell is lazy by default, and thus the speed of thunks is critical to the overall performance of most Haskell programs. As a result, GHC has a very efficient implementation of thunks. This section discusses three important aspects of efficient implementation of thunks: how to efficiently determine if a thunk is already evaluated (Section 4.1), how to safely implement thunk update in a multithreaded setting (Section 4.2), and how to avoid duplicated work evaluating a thunk when two threads attempt to evaluate the same thunk at the same time (Section 4.3).

4.1 Dynamic pointer tagging

In the naïve self-updating model, we always perform an indirect jump to the entry code of a heap object before accessing its fields. If the object is already a data constructor, this jump returns immediately as a no-op. Unfortunately, these indirect jumps are poorly supported by modern branch prediction units on processors. Ideally, we would like to avoid making a jump at all when it is unnecessary.

Modern GHC implements a *dynamic pointer tagging* scheme (Marlow *et al.*, 2007) to provide information on whether or not a heap object is evaluated or not. This scheme works by using the lower order bits (two bits in a 32-bit machine, and three bits in a 64-bit machine) in order to record whether or not the contents of a pointer are already evaluated. If the lower order bits are zero, then the pointer is unevaluated and we need to enter the closure. Otherwise, these bits can be used to record which constructor the object is, e.g. for booleans, a tag of 1 would indicate `False` and tag of 2 would indicate `True`. Pointer tags are easily added to an object when it is initially allocated (the tag never changes, because data on the heap is immutable) and must be preserved by the garbage collector.

Pointer tagging is well worth considering even for non-lazy-by-default languages. Because case analysis on a tagged pointer can be done without any memory accesses, tagged pointers enable user defined data types to be nearly as efficient as built-in types (e.g. booleans), while at the same time allowing for fields in the constructors to store extra information. And of course, they are essential for thunks!

4.2 Synchronization

We now turn our attention to how thunks are updated. Updating a thunk with its new value constitutes mutation on shared state, thus, thunks update is an important obstacle on the way to thread-safety.[12] A naïve approach is to synchronize all of the updates. This is extremely costly: Haskell programs do a lot of thunk updates!

Once again, our saving grace is purity: as thunks represent pure computation, evaluating a thunk twice has no observable effect: both evaluations are guaranteed to come up with the same result. Thus, we should be able to keep updates to thunk unsynchronized, at the cost of occasional duplication of work when two threads race to evaluate the same thunk.

A race can still be harmful, however, if we need to update a thunk in multiple steps and the intermediate states are not valid. In the case of a thunk update, we need to both update the header and write down a pointer to the result; if we update the header first, then the intermediate state is not well-formed (the result field is empty); on the other hand, if we update the result field first, we might clobber important information in the thunk. Instead, we leave a blank word in every thunk where the result can be written in non-destructively, after which the header of the closure can be changed.[13]

```
word    step 1   step 2   step 3
  0      THUNK    THUNK      IND \ valid closure
```

[12] One might say it's the only obstacle, as pure code requires no synchronization and explicit interthread communication utilizes similarly explicit synchronization.

[13] Appropriate write barriers must be added to ensure the CPU does not reorder these instructions.

```
1         -   result   result /
2   payload  payload  payload <- payload is slop
```

4.3 Black holes

Some thunks take a long time to evaluate: we'd like to avoid duplicating their work. What we would like is for threads to notice when someone is working on a thunk, and wait for the result to become available.

The mechanism by which this is implemented is a *black hole*, which represents a thunk that is currently being evaluated. A thunk can be *claimed* by overwriting it with a black hole. Black holes were originally proposed as a solution for a space leak that occured in some cases of tail calls (Jones, 2008), but they have found their utility in a multithreaded setting. Recall that a thread wishing to evaluate a thunk jumps to the entry code; the entry code of a black hole places a thread on the blocked queue of the owner the black hole, to be woken up when the thunk has been evaluated.[14]

There are two times when a black hole can be written: it can be *eagerly* written as soon as a thunk is evaluated, or it can be *lazily* deferred for when a thread has gotten descheduled (and thus the thunk was taking a long time to evaluate.) If a black hole is written eagerly, it is on the fast path of thunk updates, and we cannot use synchronization. We call these *eager black holes* (also known as *grey holes*), which do not guarantee exclusivity. Lazy blackholing is done more infrequently, and thus we can afford to use a CAS to implement them.[15]

The upshot is that GHC is able to implement lazy evaluation without any synchronization for most thunk updates, applying synchronization *only* when it is specifically necessary. The cost of this scheme is low: a single extra field in thunk and a (rare) duplication of work upon a race.

5 Acknowledgements

Alexander Chernyakhovsky for helpful discussions.

References

Arora, Nimar S., Blumofe, Robert D., & Plaxton, C. Greg. (1998). Thread scheduling for multiprogrammed multiprocessors. *Pages 119–129 of: Proceedings of the tenth annual acm symposium on parallel algorithms and architectures*. SPAA '98. New York, NY, USA: ACM.

Dybvig, R. Kent, Eby, David, & Bruggeman, Carl. (1994). *Don't stop the BIBOP: Flexible and efficient storage management for dynamically-typed languages*. Tech. rept.

[14] It is somewhat difficult to put a blocked queue on the thunk itself (due to the lack of synchronization); instead, GHC uses a per-thread list of black hole blockers which is traversed every time a thread finishes updating a thunk.

[15] While multiple threads may have eagerly blackholed a thunk, we guarantee only one thread has lazily blackholed it. If a thunk *must not* be duplicated, it can achieve this by forcing all of its callers to perform lazy blackholing (`noDuplicate#`). `unsafePerformIO` uses precisely this mechanism in order to avoid duplication of IO effects.

Harris, Tim, Marlow, Simon, & Jones, Simon Peyton. (2005). Haskell on a shared-memory multiprocessor. *Pages 49–61 of: Proceedings of the 2005 acm sigplan workshop on haskell*. Haskell '05. New York, NY, USA: ACM.

Hendler, Danny, Lev, Yossi, Moir, Mark, & Shavit, Nir. (2005). *A dynamic-sized nonblocking work stealing deque*. Tech. rept.

Jones, Richard. (2008). Tail recursion without space leaks. *Journal of functional programming*, **2**(01), 73.

Marlow, Simon. (2013). *GHC commentary: The garbage collector*. Available online at `http://hackage.haskell.org/trac/ghc/wiki/Commentary/Rts/Storage/GC`.

Marlow, Simon, & Jones, Simon Peyton. (2004). Extending the haskell foreign function interface with concurrency. *Pages 57–68 of: In proceedings of the acm sigplan workshop on haskell*.

Marlow, Simon, Jones, Simon Peyton, Moran, Andrew, & Reppy, John. (2001). Asynchronous exceptions in haskell. *Pages 274–285 of: Proceedings of the acm sigplan 2001 conference on programming language design and implementation*. PLDI '01. New York, NY, USA: ACM.

Marlow, Simon, Yakushev, Alexey Rodriguez, & Jones, Simon Peyton. (2007). Faster laziness using dynamic pointer tagging. *Acm sigplan notices*, **42**(9), 277.

Marlow, Simon, Harris, Tim, James, Roshan P., & Peyton Jones, Simon. (2008). Parallel generational-copying garbage collection with a block-structured heap. *Pages 11–20 of: Proceedings of the 7th international symposium on memory management*. ISMM '08. New York, NY, USA: ACM.

Marlow, Simon, Peyton Jones, Simon, & Singh, Satnam. (2009). Runtime support for multicore Haskell. *Acm sigplan notices*, **44**(9), 65.

Peyton Jones, Simon, Gordon, Andrew, & Finne, Sigbjorn. (1996). Concurrent haskell. *Pages 295–308 of: Proceedings of the 23rd acm sigplan-sigact symposium on principles of programming languages*. POPL '96. New York, NY, USA: ACM.

Peyton Jones, Simon L., Marlow, Simon, & Elliott, Conal. (2000). Stretching the storage manager: Weak pointers and stable names in haskell. *Pages 37–58 of: Selected papers from the 11th international workshop on implementation of functional languages*. IFL '99. London, UK, UK: Springer-Verlag.

Reid, Alastair. (1999). Putting the spine back in the Spineless Tagless G-Machine: An implementation of resumable black-holes. *Implementation of functional languages*, 186–199.

Steele, Guy. (1977). *Data representations in PDP-10 MACLISP*. Tech. rept. MIT.