Haskell on a Shared-Memory Multiprocessor

Tim Harris Simon Marlow Simon Peyton Jones
Microsoft Research, Cambridge
{tharris,simonmar,simonpj}@microsoft.com

Abstract

Multi-core processors are coming, and we need ways to program them. The combination of purely-functional programming and explicit, monadic threads, communicating using transactional memory, looks like a particularly promising way to do so. This paper describes a full-scale implementation of shared-memory parallel Haskell, based on the Glasgow Haskell Compiler. Our main technical contribution is a lock-free mechanism for evaluating shared thunks that eliminates the major performance bottleneck in parallel evaluation of a lazy language. Our results are preliminary but promising: we can demonstrate wall-clock speedups of a serious application (GHC itself), even with only two processors, compared to the same application compiled for a uni-processor.

Categories and Subject Descriptors D.3.2 [*Language Classifications*]: Applicative (functional) languages

General Terms Languages, Performance

1. Introduction

For many years the easiest approach to getting software to go faster has been to sit around and save up for a new machine (and then preferably run the old software on it). It is becoming clear, however, that this free lunch is over [22]. Processor manufacturers have stopped struggling to push clock speeds much further, and are turning their attention to parallelism instead. Multi-core processors, with several symmetric processing cores on a single chip, will be the norm in consumer machines within the next 1-2 years. The software challenge is to take advantage of this extra processing power through parallelism.

The parallel functional programming community has been chasing this very goal for more than two decades, as we discuss in Section 8. However, the rules of the game have now changed. In the past, the free lunch meant that few people were prepared to invest *any* effort to parallelise their programs – and even functional programs take work to parallelise. Furthermore, an N-processor parallel machine used to cost more than N uni-processors. Hence (a) the market for shared-memory machines was small, and (b) the people who cared enough to buy such a machine were seeking performance above all else, and were willing to invest lots of programming effort to get it.

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

Proceedings of Haskell Workshop 2005 September 30, 2005, Tallinn, Estonia.
Copyright © 2005 ACM 1-59593-071-X/05/0009. . . $5.00.

The economics have now changed sharply. Soon every machine will be a parallel machine, whether we like it or not. Programmers will therefore be forced to do parallel programming, rather than relying on the free lunch; and the extra processors come for free[1], rather than costing extra. Parallel functional programming, which offers the hope of getting moderate parallelism in exchange for modest effort, suddenly looks more attractive.

Another reason to look at this problem now is the discovery of Software Transactional Memory (STM) [9], which for the first time offers programmers a real abstraction mechanism for building concurrent systems. Furthermore, STM has an efficient shared-memory implementation, which we will describe later in the paper.

Unlike most other work on parallel functional programming (see Section 8), we concentrate exclusively on *shared-memory* architectures, because that is the architecture of the upcoming multi-core processors. The specific contributions of this paper are as follows:

- Parallel evaluation in a shared heap requires rather intimate co-operation between processors, especially to avoid race conditions when evaluating and updating thunks. Normally such co-operation requires synchronisation instructions, such as compare-and-swap (CAS), but we give measurements that show that adding these instructions to thunk evaluation is unacceptably expensive: an average overhead of more than 50% across the `nofib` benchmark suite.
- Thus motivated, our main technical contribution is a novel technique for reducing this overhead, by completely eliminating locking instructions and memory barriers from thunk evaluation and update (Section 3). We get the average overhead down to less than 6%.
- The lock-free technique, by design, leaves a small possibility of semantically-harmless duplicate evaluation. We present techniques for making such duplication very unlikely, and for bounding the amount of duplication that can take place (Section 4).
- We describe a full-scale parallel implementation of Haskell, based on the Glasgow Haskell Compiler (GHC). All of GHC's runtime features are supported: Concurrent Haskell, I/O, the foreign function interface (FFI), exceptions, interrupts, and transactional memory (Section 5).
- We give measurements for the overhead of compiling a program with support for shared-memory parallelism: this is the baseline overhead, which you pay simply for compiling a program to work with multiple threads (Section 6).
- The bottom line is wall-clock speedup for real applications. We demonstrate such a speedup, with only two processors, compared to the same program compiled for a uni-processor (Section 7). Our benchmark program is no toy: it is GHC itself. The

[1] "For free" means, as it always does, already paid for.

fact that we could parallelise it so easily gives substance to the claim that parallel functional programming offers a relatively low-effort way to exploit the power of multi-core processors.

2. Background: multi-threading in Haskell

We begin by setting the context for our work.

2.1 The programmer's eye view

The programmer may want multi-threaded execution for two distinct reasons:

Expressiveness. Many programs are concurrent by design. For example, a web-server may run a concurrent, I/O-performing thread to service each incoming request. In *Concurrent Haskell* [19], these threads are forked explicitly, using the `forkIO` combinator:

```
forkIO :: IO a -> IO ThreadId
```

Like conventional parallel programs, a Concurrent Haskell program is (by design) non-deterministic, because of the unpredictable interleaving of thread executions.

Performance. A long-standing claim of the functional programming community has been that the absence of side effects makes it possible to harness the power of multi-processors, without changing the semantics of the language at all. In principle one may extract parallelism automatically — for example, to compute $e_1 + e_2$ we may evaluate e_1 and e_2 in parallel — but it is extremely difficult to ensure that the granularity of such sub-computations is large enough. In practice, attention has focused on using programmer annotations to identify promising sub-computations. The simplest such annotation is the `par` combinator, used by *Glasgow Parallel Haskell* (GPH) [24]:

```
par :: a -> b -> b
```

The idea is that evaluating (`par` e_1 e_2) first adds e_1 to a pool of work available for unemployed processors, and then continues by evaluating e_2 [23]. In contrast to Concurrent Haskell, adding `par` annotations cannot affect the result of the program – that is why `par` is such an attractive way of exploiting parallelism.

It makes perfect sense to run a Concurrent Haskell program on a uni-processor, and GHC's standard distribution does exactly that. To keep things clear, we call this "Uni-GHC". Our goal in this paper is to extend Uni-GHC to work on shared-memory multi-processors as well: SMP-GHC.

Once that is done, adding `par` is relatively easy, because many of the underlying mechanisms (threads, scheduling, mutual exclusion) are the same for both Concurrent and Parallel Haskell. At the time of writing, we have not yet implemented `par`. However, it makes perfect sense to use `forkIO` to spawn explicitly-concurrent computations for the purpose of performance; the purity of the language makes it much easier to see where these threads may interact, and the STM makes it easy to synchronise them correctly where they do.

2.2 Threading model in Uni-GHC

Concurrent Haskell is designed to scale to applications involving hundreds or thousands of threads of execution, even on uniprocessor machines. Consequently, to make threading lightweight, Uni-GHC multiplexes *Haskell threads* onto a single *OS thread*, called a *worker-thread*. A worker thread only switches between Haskell threads at carefully controlled points, such as explicit yields, invocations on synchronisation primitives, or periodically on storage allocation.

GHC also supports interaction with native code – both calls from Haskell code into functions imported from native code, and calls from native code into functions exported from Haskell. This adds two complications to the threading model. Firstly, if a native call blocks, then it is necessary to create a new (OS) worker-thread, so that progress can be made with other Haskell threads. Secondly, a Haskell thread can be marked as *bound* [15], which means that a dedicated OS thread is reserved for it – this may be needed when interacting with external code that uses native per-thread storage. However, in Uni-GHC, although there are multiple OS threads *only one of them is ever executing Haskell code at any one time*.

2.3 Towards SMP-GHC

Unfortunately, developing SMP-GHC is not simply a case of creating multiple worker threads to run Haskell code in parallel on separate CPUs: the assumption of single-threaded execution pervades the Uni-GHC runtime system. Instead, we must be examine all of the state accessed by a worker thread while running Haskell code and either (a) replicate it for each worker, or (b) ensure that it is accessed in a safe, synchronised way so that multiple OS worker threads do not confuse each other.

To do this we use the notion of a *capability*. A capability holds all of the private state that a worker needs in order to execute Haskell code – for instance, as we see below, each capability has its own allocation area.

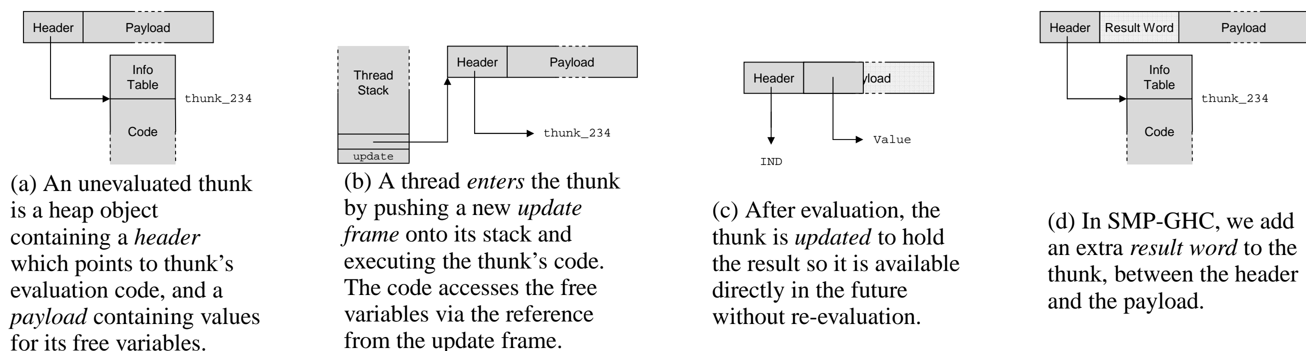
A worker thread must hold a capability in order to execute Haskell code and so the supply of capabilities serves to control the level of parallelism that the runtime system can achieve. Uni-GHC can then be seen as a degenerate case in which there is only a single capability and consequently only a single worker thread executing Haskell code; SMP-GHC is the more general case where there may be multiple capabilities, usually one for each available CPU.

Many kinds of shared state are straightforward to deal with:

1. *Immutable objects*, such as constructor cells in the heap, can be shared directly without synchronisation.
2. *Mutable objects* (`IORefs`, `MVars`, `TVars`) are all heap objects and hence are globally shared; as we discuss in Section 5 we must make these objects safe for concurrent access by multiple worker threads.
3. The *run-queue*, along with other scheduler data structures, remains globally shared between all of the worker threads. The queue is protected by a lock that must be held when manipulating it. Each worker-thread that holds a capability proceeds by taking a runnable Haskell thread from the run-queue, running it for a while, returning it to the run-queue, picking another thread, and so on. One could imagine a less centralised implementation of the run-queue, but this simple design does not form a bottleneck on today's machines, so we have not pursued this.
4. Each capability has its own private *allocation area*, or nursery, so that allocation can proceed without expensive, per-object synchronisation. Nurseries are expanded by memory supplied from a global *block allocator*.
5. The *remembered set*, used by the generational garbage collector, is globally shared. It could easily be replicated per-capability, but in most cases updates to it are rare and so the overheads of synchronised access are not high.

In short, almost everything is either replicated in each capability (e.g. the allocation pointer), or used exclusively by one capability at any instant in time (e.g. a given Haskell thread's stack), or is immutable (e.g. a constructor cell), or is seldom mutated (e.g. the run queue, or an `MVar`).

Unfortunately there is one big exception to this happy story: thunks. In a lazy language, many thunks (or suspensions) are allocated, and later evaluated. When evaluation of a thunk is complete,



(a) An unevaluated thunk is a heap object containing a *header* which points to thunk's evaluation code, and a *payload* containing values for its free variables.

(b) A thread *enters* the thunk by pushing a new *update frame* onto its stack and executing the thunk's code. The code accesses the free variables via the reference from the update frame.

(c) After evaluation, the thunk is *updated* to hold the result so it is available directly in the future without re-evaluation.

(d) In SMP-GHC, we add an extra *result word* to the thunk, between the header and the payload.

Figure 1. Thunk evaluation in Uni-GHC (a)–(c), and the new thunk format used in SMP-GHC (d).

it is overwritten with (an indirection to) its value, so that subsequent evaluations of the same shared thunk do not repeat the work of evaluating it. This allocate/evaluate/update sequence is in the inner loop of almost any Haskell program, and so it must be done efficiently. Our solution to this challenge is the main technical contribution of the paper, tackled in Section 3 and then refined in Sections 4.

2.4 Garbage collection

In our current implementation, when memory is exhausted, all worker-threads stop work, and then a single OS thread performs garbage collection. This is a stop-gap measure; clearly we would like parallel garbage collection. That should not be too hard; many techniques exist [5] and they impose no new overheads on the mutator threads. Furthermore, the benefits of parallel GC would be available even to single-threaded programs, provided multiple CPUs are available. Beyond that, concurrent garbage collection (concurrent with mutation, that is) might seem attractive, but it imposes quite serious new overheads on mutation [11].

3. Lock-free thunk evaluation

A *thunk* is a heap object that represents an unevaluated expression in the program. The problem with evaluating thunks in parallel is that although the computation performed by evaluating a thunk is logically side-effect free, the actual process of evaluating it involves updates to the shared heap – these updates are crucial for efficiency because they prevent the same thunk from being re-evaluated.

The structure of a thunk object is shown in Figure 1(a); it consists of a *header word* and a variable-sized *payload*. The header word points to the thunk's *info table* and its *entry code*. The payload contains pointers to the free variables of the expression represented by the thunk (perhaps themselves other thunks). The info table describes the layout of the heap object to the garbage collector.

In Uni-GHC, thunk evaluation proceeds in the following way:

1. A thread that needs the value of the thunk *enters* it by loading a pointer to the thunk into a register and jumping to the entry code for the thunk.
2. The entry code for the thunk does the following:
   - It pushes an update frame on the stack.
   - It evaluates the expression represented by the thunk.

   An update frame comprises two words: a pointer to the thunk to be updated, and a return address, `update`, pointing to a run-time system routine that will update the thunk when execution returns to this frame. This is shown in Figure 1(b).
3. When the computation of the expression is complete, the computed value (always another heap object) is put in a register, and control is transferred to the topmost return address on the stack, in this case `update`. The `update` code overwrites the original thunk with an *indirection* to the value, so that the next time its value is required, it doesn't have to be recomputed, and the value computed the first time can be returned. This is shown in Figure 1(c).

An indirection is a two-word heap object. Like all heap objects, its header word `IND` points to executable code, while the second word is the payload. The code for an indirection simply enters the payload object (just as in Step 1 above). This design means that a thread does not need to explicitly test whether a thunk has already been evaluated when it enters it: it either proceeds with evaluation, or enters the `IND` code and retrieves the existing result.

3.1 A bad idea: locking thunks

In a parallel world, two threads might attempt to evaluate the same thunk at the same time. Since evaluating a thunk can require an unbounded amount of work, duplicate evaluation is clearly a Bad Thing. The obvious solution is to lock the thunk while it is under evaluation, using either a standard mutex supplied by the OS, or by rolling our own locking implementation, for example doing compare-and-swap (CAS) on the header word of the thunk.

The trouble is that thunk evaluation is extremely common, and CAS instructions are extremely expensive – at least two orders of magnitude more expensive than ordinary instructions (and the ratio is getting worse). This matters: in Section 6 we show that adding two CAS instructions to every thunk's evaluation (one in the entry code and one in the `update` code) increases execution time by an average 50% with a maximum of 300%. In an earlier complete (but now-bit-rotted) implementation, we observed execution time increasing by 100% when locking thunks. We consider this to be unacceptable: even if there is plentiful parallelism, you would need an entire extra processor just to get the same performance that our sequential implementation has on a single processor.

In short, full thunk locking is unreasonably expensive.

3.2 A good idea: lock-free thunks

The key idea of this paper is this: *evaluate thunks with no locking instructions whatsoever*. This lock-free approach is based on the following observations:

1. Because a thunk always represents a pure expression, semantically it doesn't matter if two threads evaluate the same thunk, because they will both return equivalent values. It doesn't matter which one "wins", since the values will be equivalent – any difference will be unobservable by the program (but see Section 3.5).
2. Many thunks are cheap, so duplicate evaluation often doesn't matter.

3. Concurrent evaluation of a thunk by two different threads is rare.

If these observations are true, then all we need do is (a) ensure that concurrent lock-free evaluate/update operations on a thunk do not confuse each other, (b) narrow (but not close) the window during which it is possible for two threads to begin concurrent evaluation of the same thunk and (c) provide some mechanism to recover from the rare case of concurrent evaluation of an expensive thunk.

The devil is in the details however. We tackle (a), which concerns correctness, in this the rest of this section, leaving (b) and (c), which concern efficiency, for Section 4.

3.3 The first enter/update race

The first thing we must do is ensure that if two threads succeed in entering the same thunk, they do not trip over each other. Although the expression being evaluated is pure, the *update* step at the end of evaluation rewrites the thunk's header and the first word of its payload. The first concern is that one thread might complete evaluation and overwrite the thunk with an indirection to the result, while the other thread is still reading the payload of the thunk. Consider this evaluation:

	Thread A	Thread B
1.	Jump to thunk's entry code	Jump to thunk's entry code
2.	Load free variables	
3.	Evaluate thunk	
4.	Return to update frame	
5.	Update thunk with indirection	
6.		Load free variables

At step (5), the pointer to the result of the thunk has overwritten one of the free variables, so in step (6), thread B reads an invalid value for the first free variable and proceeds with evaluation using this bogus value.

The solution to this race is straightforward: we extend the size of the thunk by one word, adding a *result word* before the first free variable. This new structure is shown in Figure 1(d). Extending the size of thunks by one word is not trivial in terms of its impact on performance, but it is acceptable; we present some measurements in Section 6.

3.4 The second enter/update race

The second problem we must address occurs if one thread is entering a thunk just as another thread is updating it. This is because the update step involves two separate writes to memory, one to store `IND` in the header word and one to store the result itself: the thread entering the thunk may see one write but not the other.

For instance, suppose that the updater writes to the header word first and then stores the result:

	Updating thread	Another thread
1.	Write `IND`	
2.		Read `IND`
3.		Read bogus result field
4.	Write result field	

It is straightforward to prevent this problem on Intel and AMD x86 processors. We simply need to write the `update` function so that the result is stored *first* and the header *second*: even in a multi-processor system with caches, write buffers, and so on, the hardware guarantees that a thread that sees the update to the header will see the result.

The situation is more complex on other processor architectures: processors vary in exactly what guarantees they make when executing code where memory is being shared without using locks. Typically, some form of *memory fence* instruction is needed to constrain the order in which unsynchronised memory accesses take place [1, 26]. Unfortunately these memory fences are often as slow as atomic compare and swap operations which, as we saw above, are unacceptable to add to the fast paths through thunk entry and update. There are two problems to consider: whether writes performed by the updater may be re-ordered by the hardware, and whether the reads performed by another thread may be re-ordered.

If the processor allows writes to be re-ordered then unfortunately we do need a memory fence before executing the `update` code for the thunk. This ensures that data structures reachable from the result will be visible to other threads that use the result. Of course, memory fences are needed for the same reason in other languages, for example to ensure that initial field values written in a constructor are seen correctly by other threads.

If a processor allows reads to be re-ordered within the memory subsystem then we can still avoid adding a barrier to the `entry` code by exploiting the fact that result of evaluating a thunk is always a non-zero pointer into the heap. If we ensure that the result field is initialised to zero, and that this initialisation is visible to all processors, then if a thread enters an indirection closure and sees zero then it simply busy-waits or yields until the result reaches memory.

3.5 UnsafePerformIO

So far we have assumed that a thunk represents a pure computation, with no side effects whatsoever. GHC, however, supports `unsafePerformIO`, a primitive with type `IO a -> a` [18]. As its name suggests, it is unsafe, but it is occasionally useful. An example of a safe use would be to wrap a foreign call to C function, that was in fact pure.

However, less savoury uses of `unsafePerformIO` could be in big trouble if they could be (unpredictably) executed twice if parallel threads enter the same thunk. We have not tackled this yet, but the appropriate thing is probably to provide a combinator

```
justOnce :: a -> a
```

that does proper locking on its thunk argument. Ennals encountered just the same issue in his work on adaptive evaluation [4].

3.6 Summary

To summarise, we can perform correct, lock-free thunk evaluation as follows on Intel and AMD x86 processors:

- Every thunk contains a *result word*, to receive the updated value,
- When updating, store the result before writing the indirection header word.

We want performance as well as correctness, however. The following section discusses how to recover from the situation when two threads are evaluating the same thunk, and how to narrow the window during which two threads may *start* to evaluate the same thunk.

4. Recovering from duplicate evaluation

Most thunks are cheap: they are entered, evaluated, and updated relatively quickly. For these thunks we want lock-free evaluation, and we are prepared to risk duplicating their work in the unlikely case that two threads evaluate them concurrently – after all, they are cheap. In contrast, for expensive thunks the overheads of locking are quite acceptable. In this section we describe how to lock only expensive thunks.

4.1 The key (old) idea: black-holing

Recall that the stack of a Haskell thread contains update frames, each of which points to a thunk that the thread is evaluating. *We can*

therefore arrange that periodically, each thread scans the update frames in its stack, and uses a CAS instruction to gain exclusive access to the thunk. We call this "claiming the thunk".

In more detail, to claim a thunk, the thread (let's call it A) uses a CAS instruction to swap the header word with BLACKHOLE. The swapped-out contents of the header word could be one of three things:

- thunk_234 (the original header word of the thunk): Thread A has successfully claimed the thunk, leaving it as shown in Figure 2(a).
- BLACKHOLE: another thread B has already claimed the thunk.
- IND: another thread B has already updated the thunk.

Suppose for the moment that Thread A succeeds in claiming all the thunks pointed to by its update frames, after which it resumes normal evaluation. Now suppose that another thread B tries to enter one of those thunks; it will land in the code for BLACKHOLE (remember, every header word is a code pointer). This code must arrange for Thread B to *block*, waiting for Thread A to complete evaluation of the thunk. We discuss the mechanism for blocking in Section 4.2.

Suppose that Thread A finds an update frame while scanning its stack pointing to a thunk that already contains IND or BLACKHOLE (the latter two cases above). Then everything on the stack subsequent to (i.e. younger than) this update frame represents redundant computation. Hence, we want to truncate A's stack to this update frame, and leave Thread A in a state such that when it resumes execution it will enter the thunk as if for the first time. If it enters a BLACKHOLE, it will block, as above; but it the thunk is an IND, it will simply find the value. If several update frames on A's stack have IND or BLACKHOLE thunks, we want to truncate the stack to the deepest (i.e. oldest) one. The operation of "truncating A's stack" is a little trickier than it sounds, as we discuss in Section 4.3, but the effect is to abort A's redundant computation.

Note that the fact that Thread A succeeds in claiming a thunk does *not* guarantee that no other thread B is evaluating it, because B might not have gotten around to trying to claim it yet. Indeed, B might even get all the way through to updating it (if the thunk is cheap). But if the thunk takes a long time to evaluate, B will try to claim it, and will back off then.

Since Thread A scans its own stack repeatedly, it must take care not to scan the same update frame more than once, because the second time it, of course, will find BLACKHOLE, put there during the previous scan! This is easily arranged – by marking the update frame, with a bit or by change the update code pointer – and has the side benefit of saving work: once we find a marked update frame, we can stop scanning, and no frame is scanned more than once.

The idea of overwriting a thunk with a "black hole" while the thunk is being evaluated is far from new. It is useful even in a sequential implementation to plug space leaks [10], and to detect certain sorts of infinite loops. Because it is deferred until a stack-scan is performed, we sometimes call it *lazy* black-holing. We can also use an eager, lock-free, variant of black-holing to dramatically reduce the window in which duplicate evaluation can occur (Section 4.4).

4.2 Blocking

When a (Haskell) thread enters a black hole – that is, a thunk with BLACKHOLE as its header word – we want to arrange to block the thread until the thunk's evaluation is complete. In Uni-GHC the blackhole entry code places the thread on a queue attached to the thunk itself; the update code (executed when a thunk is updated with its value) checks for waiting threads and wakes them up. Another new header word, BLACKHOLE_BQ identifies black-holed thunks which form the head of queues of blocked threads. This is shown in Figure 2(b).

In SMP-GHC, this technique runs into difficulties. Some care must be taken to co-ordinate multiple threads that block simultaneously on the same thunk, although here we use proper locking instructions, since blocking is rare; and we would probably need yet another word to contain the (almost invariably empty) blocking queue. The worst thing, though, is that the update code, which updates the thunk with its value, must check for blocked threads, and it is hard to see how to make that lock-free.

To avoid these problems SMP-GHC abandons the Uni-GHC approach. Instead, we keep blocked threads in a separate global queue. The entry code for BLACKHOLE places the thread on the global *black-hole queue* as shown in Figure 2(c). Note that each thread points back to the black hole on which it is blocked – threads on the black-hole queue are checked at regular intervals to see whether the computation they are waiting for is complete, so they can be woken up.

Of course, we cannot avoid locking or CAS when updating the global queue, but unlike the fast-path code on thunk entry and update, we aren't too concerned about atomic actions here because we expect blocking on black holes to be relatively rare (see measurements in Section 7.5). The queue could be made per-capability in any case.

The global queue brings some new problems. Firstly, traversing the queue is $O(n)$, so we must not traverse it too often. Our current implementation traverses it at least at every GC (when every thread is touched anyway), and also when there is an idle CPU. Secondly, blocked threads don't get woken up as promptly as in the previous scheme. It's possible that a thread might get unfairly starved if it often blocks on BLACKHOLEs. Our implementation doesn't do anything to mitigate this, but we don't expect it to be a serious problem in practice.

4.3 Truncating the stack

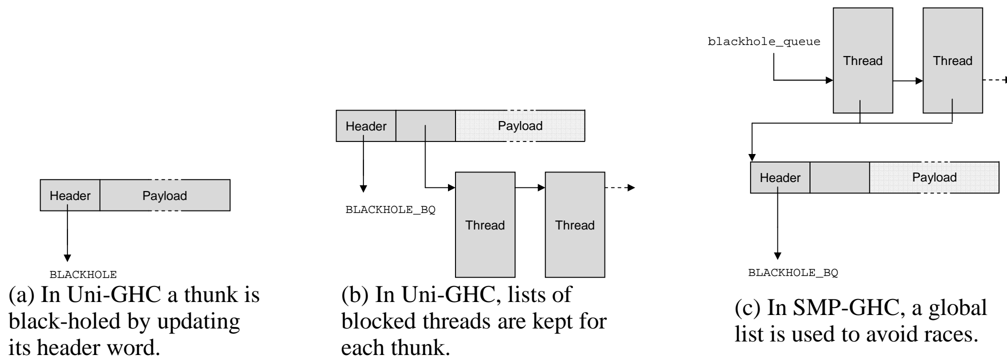
It is tempting to think that when we truncate a stack we can simply discard the truncated portion wholesale. After all, there are no effects to undo – this is a functional program! However, this stack chunk may itself contain update frames for other thunks under evaluation. Some of these thunks may be visible to other threads, so we cannot simply discard this stack chunk and the work it represents, because that would leave these shared thunks in a semi-evaluated state (probably BLACKHOLEs), and the next thread to enter one of them would block forever.

This is not a new problem. Exactly the same issues arise whenever a thread's execution must be abandoned for some reason:

- GHC supports asynchronous interrupts, which allow one thread (or an external source) to interrupt another [16]. The interrupted thread abandons its stack until it finds an exception handler.
- In our implementation of Software Transactional Memory (STM), we periodically "validate" the thread's transaction log, to check that it has seen a consistent view of memory; if not, we abandon the transaction and re-execute it [9].

The requirement, then, is to ensure that the black holes pointed to by the update frames of an aborted stack chunk are left in a sensible state. We could consider reverting each of the black holes back to its unevaluated state, but that would require keeping the original state of the thunk until its evaluation is complete; recall that one of the purposes of black-holing is to eliminate the space leak caused by retaining the free variables of a thunk under evaluation. Moreover, reverting the thunk would throw away the work that has been performed on it so far.

Fortunately a better solution is known [20]. The trick is to save the stack onto the heap in such a way that if any of the thunks



(a) In Uni-GHC a thunk is black-holed by updating its header word.

(b) In Uni-GHC, lists of blocked threads are kept for each thunk.

(c) In SMP-GHC, a global list is used to avoid races.

Figure 2. Thunk black-holing in Uni-GHC (a)–(b), and in SMP-GHC (c).

are entered again, the saved stack is reconstructed on the entering thread's stack, and the evaluation of the thunk resumes where it left off. We call this "freezing" the state of the evaluation. A nice property is that if the frozen thunks are not shared with any other threads, then the garbage collector will quickly throw away the frozen state.

Uni-GHC already implements this strategy, and we simply adopt it for SMP-GHC. No new concurrency issues arise, because updating the black hole with the suspended stack is just like updating the black hole with its final value (except, of course, that the result field points to a new thunk, representing the frozen stack, rather than to an immutable value).

4.4 Narrowing the window using grey-holing

So far we have assumed that until a thunk is black-holed by the claiming operation, it remains unmodified, so there is quite a wide window in which two threads might begin evaluating it simultaneously. It is easy to narrow the window: as soon as a thread enters the thunk, it writes `GREYHOLE` into the header, without taking any locks. The entry code for `GREYHOLE` is the same as for `BLACKHOLE`, so that any other thread entering the thunk will now block (Section 4.2). Of course, the window is not closed entirely: when one thread has read the header of the thunk, but not yet written `GREYHOLE`, a second thread could also read the header word and begin a duplicate evaluation. But now the window is only one instruction wide.

Why not simply write `BLACKHOLE`? Because we need to use a different header word so the lazy black-holing mechanism of Section 4.1 can distinguish (a) when it has successfully claimed exclusive access from (b) when it comes across a thunk that has been claimed by another thread.

However, since a thunk is now mutated *twice* in a lock-free way, once to grey-hole it and once to update it with its final value, grey-holing introduces a new race condition:

	Thread A	Thread B
1.	Enter thunk's code	Enter thunk's code
2.	Write `GREYHOLE` header	
3.	Evaluate thunk	
4.	Write result field	
5.	Write `IND` header	
6.		Write `GREYHOLE` header
7.		Evaluate thunk

This race does not threaten correctness. All that happens is that Thread B will evaluate the thunk all over again. Notice that the race occurs because Thread A completes *all of the thunk's evaluation* between *two instructions* in Thread B's execution. Therefore the more expensive that evaluation is, the smaller the chance that Thread B will be asleep during the entire evaluation by Thread A. So it is quite acceptable simply to ignore this race.

4.5 Duplicate unshared thunks

Even if we can guarantee to catch any duplicate evaluation of a shared thunk within a bounded amount of time, this does not unfortunately place a bound on the amount of duplication we can expect. For example, consider the following thunk `z`:

```
z = let x = ... expensive ...
    in Just x
```

`z` is very cheap to compute, but its value contains an expensive-to-evaluate thunk `x`. If the evaluation of `z` is duplicated, then there will be multiple results each pointing to a different version of `x`. The runtime cannot detect that the two versions of `x` are equivalent, so the evaluation of `x` will be completely duplicated.

The best we can do in our lock-free design is to make this scenario highly unlikely to occur. Grey-holing already significantly reduces the possibility that evaluation of `z` will be duplicated, but we might not want to use grey-holing because of the performance penalty.

A cheaper technique is to check the header word of the thunk in the update code: if the header word is already `IND`, then we can return the existing result rather than the result we have just computed. This trick isn't foolproof because two updates can still happen simultaneously, but if successful, it does recover the sharing at the expense of an extra read during update.

To date, we haven't implemented this technique or measured its overhead. However, we expect the overhead to be low: reading the header word in the update code doesn't increase memory traffic, because the header word is being written to anyway, so it needs to be in the cache. In fact, Uni-GHC already performs this read because it checks for `BLACKHOLE_BQ` on update. It does represent a control flow decision based on the results of a memory read, so there might be a pipeline stall, but we expect the majority of thunks to be cheap to evaluate and therefore still in the cache when the update code runs.

It takes two physical processors to duplicate work, and in that case there are two physical processors to excecute the duplicates, so one might wonder whether the (highly-unlikely) worst case is to slow down to the speed of a uni-processor. Sadly, this is not quite right, because one of the duplicate threads might be de-scheduled, and the two processors might accidentally duplicate more work in the other — and then do the same in the de-scheduled thread. It seems that there is no hard upper bound, but that the chances of repeated duplication decrease exponentially with the number of duplicates. This problem does not keep us awake at night.

4.6 Summary

The runtime has complete freedom to decide how frequently to scan the update frames on a thread's stack. One plausible possibility would be to scan the stack when the thread is descheduled; that is, when it blocks, runs out of allocation area, or its time-slice expires. Only active threads need their stacks scanned at all; sleeping or blocked threads need no such attention. However, if grey-holing is being used, it is extremely unlikely that two threads will manage to squeeze through the one-instruction window, and thereby evaluate the same thunk, so it probably makes sense to scan the stacks very seldom.

Suppose that each Haskell thread scans its stack every T ticks. Then the scheme guarantees that any thunk whose evaluation takes longer than T ticks will be claimed by a unique thread; and any other threads that manage to squeeze into the one-instruction window, and thereby evaluate the thunk concurrently, will waste at most T ticks each.

5. Atomic blocks in SMP-GHC

In Sections 3 and 4 we showed how to support safe parallel evaluation of pure functional code without having to introduce per-thunk locking. We now turn to the problem of *impure* multi-threaded code where threads communicate with one another through explicit updates to shared memory. As with parallel thunk evaluation, we want the underlying primitives to be safe, fast and scalable.

Our recent work in Uni-GHC provides *atomic memory transactions* as an abstraction for composable inter-thread communication [9]. These are built using a software transactional memory (STM) [21] which allows a set of accesses to a shared mutable heap to be performed atomically.

The STM is implemented using *optimistic concurrency control* in which an atomic block executes building up a Haskell-thread-local log of all of the transactional variables (`TVar`s) that it has read from and, in the case of updates, the value that it wants to write. At the end of the atomic block, the thread invokes a `commit` operation that iterates over the log checking that the `TVar`s still hold the values seen in them: if so then the updates are written, if not the log is discarded and the atomic block is re-executed.

This scheme is relatively straightforward to implement in Uni-GHC because only one thread can be evaluating Haskell code at any time, so there is no interleaving between different `commit` operations. The implementation in SMP-GHC is more intricate but largely employs the same techniques that we have used in earlier work on STMs for multiprocessor systems [8, 6].

The basic idea is to implement per-`TVar` locks using atomic CAS instructions. As usual, we implement these locks by overloading the `current_value` field in a `TVar`: a single CAS instruction thereby serves to acquire a lock and to check that the `TVar` held the value expected there by the transaction. However, notice that these locks are held *only when committing a transaction* and not throughout its execution – contention is therefore expected to be rare.

We avoid locking altogether for `TVar`s that have been read but not updated. This aids scalability when dealing with shared data structures that are often read but seldom updated: a read-only transaction can operate without introducing costly contention in the memory hierarchy. As in earlier work, we do this by adding a `version` field to each `TVar` that is protected by the `TVar`'s lock and is updated on commit. During a `commit` operation we make two passes over the `TVar`s that have been read but not updated – the first pass records the versions seen in each of them, and the second pass checks that none of these versions has changed. This guarantees that we see a consistent view of the set of `TVar`s. Figure 3 summarises this algorithm; Fraser provides a more detailed description in the context of an STM library for C [6].

1. *Lock tvars*

```
for each transaction log entry:
  if the entry is an update:
    try to lock the tvar
    if successful:
      continue
    else:
      unlock tvars and abort
  if the entry is a read:
    record tvar's version number
```

2. *Check reads*

```
for each transaction log entry:
  if the entry is a read then
    re-read the tvar's version number
    if this matches the one we recorded:
      continue
    else:
      unlock tvars and abort
```

3. *Make updates*

```
for each transaction log entry:
  if the entry is an update:
    store new value to tvar, unlocking the tvar
```

Figure 3. Committing a transaction, allowing non-conflicting updates and reads to proceed in parallel.

Although this overall structure is conventional, there are three novel aspects of our STM design. Firstly, unlike earlier STMs, we do not aim to make the `commit` operation lock-free – that is, if an OS thread is pre-empted mid-way through a call to `commit` then other OS threads will be unable to perform conflicting updates until the first thread is rescheduled. Lock-free behaviour is important in languages with an unconstrained number of OS threads operating without co-operation from the scheduler. However, in SMP-GHC, the number of OS threads is set to match the number of available CPUs, and scheduling between Haskell threads is under the control of our scheduler. This makes pre-emption during commit operations extremely unlikely in SMP-GHC.

Secondly, the fact that we are not lock-free means that we must avoid deadlock when locking `TVar`s during a `commit`. We do not want to rely on sorting the entries in the transaction log because of the work that sorting entails [14], and the fact that contention for these locks is rare. Instead we simply abort and re-execute a transaction if we fail to acquire a lock during `commit`. However, if contention is more frequent, then we could instead release any locks acquired so far and then proceed to sort the transaction log before reacquiring the locks. This may reduce the number of needless aborts while still avoiding the need to sort the transaction log in every case. In practice the rarity of contention for `TVar` locks means that we have not needed to explore this more complicated implementation.

Finally, a novel feature of Concurrent Haskell's STM is that it supports a `retry` operation: conceptually, if a thread calls `retry` then its current transaction is abandoned with no side effects and then re-executed from scratch. However, there is no point in actually re-executing the transaction until *at least one of the* `TVar`*s read during the attempted transaction is written by another thread*. This observation is exploited by using a per-`TVar` queue of Haskell threads that are waiting for an update to be made. A `retry`ing thread adds itself to the queue attached to each of the `TVar`s that

Program	Code size	Runtime
anna	+8.0	+39.1
cacheprof	+7.5	+74.4
circsim	+5.3	+88.7
compress	+4.7	+14.5
exp3_8	+4.3	+320.0
fft	+5.1	+30.5
fibheaps	+4.4	+50.3
fulsom	+7.3	+50.1
sched	+4.3	+78.5
wang	+5.2	+35.3
Min	+3.2	-4.5
Max	+8.0	+320.0
Geometric Mean	+4.9	+53.9

Figure 4. Overhead of using CAS

Program	Code size	Allocations	Runtime
anna	+0.7	+16.9	+6.2
cacheprof	+1.1	+17.3	+0.7
circsim	+1.7	+17.4	+6.6
compress	+2.3	+1.4	+3.4
exp3_8	+2.0	+20.0	+22.3
fft	+1.5	+12.2	+11.7
fibheaps	+1.9	+9.5	+1.9
fulsom	+1.2	+16.6	-0.8
sched	+1.9	+12.8	+1.6
wang	+1.5	+15.2	+34.3
Min	+0.7	+0.0	-8.2
Max	+3.3	+24.5	+41.0
Geometric Mean	+1.9	+12.4	+5.8

Figure 5. Lock-free implementation

the transaction read, and the `commit` operation re-awakens any thread waiting on a `TVar` written by the `commit` [9]. We make these queues safe for SMP-GHC by re-using the per-`TVar` lock to protect the wait queues. A thread that is about to wait must acquire all of the per-`TVar` locks it needs before adding itself to their wait queues: this prevents lost wake-up problems resulting from concurrent commit operations to those `TVar`s.

6. Measuring the overhead of parallel execution

In this section we measure the overhead imposed by the measures we have taken to allow parallel execution of Haskell code by multiple threads on a shared heap.

6.1 Methodology

We used the Glasgow Haskell Compiler version 6.4 plus modifications (corresponding to the CVS sources around the date of 31 May 2005), running on Linux. Our measurements are taken across all 88 programs in the `nofib` benchmark suite [17], which range from micro-benchmarks such as `tak` and `rfib`, to "real" programs. For example `cacheprof` is a program for automatically translating assembly code to insert instructions for dynamic cache profiling, `compress` is an implementation of LZW compression, and `hidden` is a program for hidden-line removal in 3D rendering.

Although we measured all 88 programs, our tables only show a subset of the results. Nevertheless, the averages and min/max figures do take into account *the whole suite*. When taking the average of percentages, we give the geometric mean. However, to reduce spurious figures, any program that ran for less than 0.5 seconds on our dual 2.4GHz Intel Xeon was discounted from the aggregate figures.

Runtimes are wall-clock times, taken as the average of 5 runs.

6.2 Overhead of atomic instructions

Our first experiments looked into the total cost of adding a single un-contended compare-and-swap instruction to the code for every thunk entry and to the update code. Although this does not implement a proper lock on the thunk, it suggests the kind of performance that would be achieved by a basic lock-based scheme.

Figure 4 gives the measurements. We found that simply adding atomic compare-and-swap instructions to the thunk entry and update imposes a significant performance penalty: from 0-300% slower, with the average being about 50% slower. The small benchmarks (`tak`, `listcopy`, `exp3_8`) show the most extreme behaviour, whereas larger programs such as `anna` and `fulsom` display behaviour closer to the average.

It might be feasible to construct a locking implementation around a single CAS instruction per evaluation rather than the two we have measured here[2] But even halving the penalty we have observed still leaves a prohibitively expensive overhead, and a real locking implementation will need to do more than just a CAS instruction.

6.3 Overhead of the lock-free implementation

Figure 5 gives measurements of the sequential overhead for SMP-GHC, and described in Sections 3-5.

Our baseline figures were measured on a system that did (lazy) black-holing, but not (eager) grey-holing. The black-holing implementation is incomplete compared to that described in Section 4, in that it does not use CAS to black-hole the thunk, and makes no attempt to detect or recover from duplicate evaluation. This shortcoming should only worsen performance, because duplicate evaluation is possible, although unlikely. (The performance boost from not using CAS during black-holing will be very small, because lazy black-holing is, by design, not the inner loop.) Concerning memory ordering (Section 3.4), the Intel Xeon processor on which we ran the benchmarks has strong guarantees about memory ordering, which means that a processor can never see the writes performed by an update occur out-of-order, so it is neither necessary to zero the result word nor to add memory fence instructions.

We can see that the overhead of the lock-free implementation is around 6% of runtime, which is significantly lower than the overhead of using atomic compare-and-swap instructions.

However, there are still a few outliers in the benchmark suite: `treejoin` takes a 41% performance hit when compiled for parallel execution, for example and `wang` a hit of 34%. A combination of factors is at work in these cases. Firstly, they perform an unusually large number of updates to thunks that are in the old generation. This creates contention for the lock which currently protects the GC's remembered set – as we mentioned earlier, we intend to add per-capability remembered sets so that we can remove this lock. Secondly, these tests are reasonably short running and perform only a small number of old-generation garbage collections. Execution finishes just before a further garbage collection would occur and so the additional storage space required for result words fills the heap, causing a further old-generation collection. Aside from these factors, the performance is close to the mean.

6.4 Grey-holing

We also measured the impact of adding grey-holing (sometimes also called eager black-holing) to the lock-free implementation.

[2] Note that CAS-free lock reservation schemes, such as Kawachiya's [12], rely on repeated acquisitions and releases of the same lock: thunks are only 'locked' once.

Program	Code size	Runtime
anna	+1.3	-1.5
cacheprof	+0.9	+0.3
circsim	+0.9	-1.7
compress	+0.6	+1.1
exp3_8	+0.6	+6.6
fft	+0.7	+3.8
fibheaps	+0.6	+17.0
fulsom	+1.1	+2.0
sched	+0.6	+53.5
wang	+0.8	-0.4
Min	+0.3	-5.2
Max	+1.3	+53.5
Geometric Mean	+0.7	+2.1

Figure 6. Lock-free implementation with grey-holing

The results are given in Figure 6, where the baseline is the system measured in Figure 5.

While the code-size overhead is almost negligible at less than 1%, the effect on the runtime is sometimes dramatic: one program, `sched`, showed a 53% increase in runtime. We double-checked the measurement and it is correct. Theoretically, writing to the thunk's header in the entry code for the thunk *could* cause cache-thrashing, because the thunk would not otherwise be written to at that point during execution; we don't know whether in fact this is the cause of the performance drop for `sched`, however.

7. Case study: parallelising GHC

In this section we describe a case study using our shared-memory parallel implementation of GHC, and demonstrate a real speedup achieved for a distinctly non-trivial program, with only minor changes to the source code of the application.

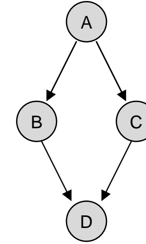
The application we chose to parallelise is GHC itself, for two reasons:

- It is compute-bound, but has a natural granularity for parallelism: compiling modules in parallel. Parallelisation is not completely trivial, however, since there is some shared state between the compilations.
- The authors are already intimately familiar with the architecture of GHC, so identifying the potential areas of concern for a parallel implementation was not difficult.

For some time now, GHC has had the ability to compile several modules in sequence, without having to be restarted between compilations. This is called the `--make` mode of GHC. There are two main benefits to using GHC in `--make` mode:

- Speed: GHC caches information about modules between compilations, so the information is immediately available to subsequent compilations without having to be re-read from the disk. This applies not just to modules compiled in the current session, but also pre-compiled modules in libraries; reading interfaces for pre-compiled modules such as the `Prelude` is a significant factor in GHC's compilation time, especially for non-optimising compilation.
- Simplicity: GHC does the dependency analysis internally, so only a single command needs to be issued in order to build an entire program or library. The programmer doesn't need to be familiar with any other external tools.

Since a multi-module program often contains subsets of modules which have no dependencies between them, it is natural to wonder whether these compilations could proceed in parallel. For example, in a program with this structure:



where module `A` depends on both `B` and `C`, and `B` and `C` both depend on `D`, we could proceed by first compiling `D`, then compiling `B` and `C` in parallel, and finally compiling `A`.

Indeed, the Unix `make` tool already has such a facility: issuing the command `make -j2` will compile the program using at most 2 processes in parallel whenever possible. Inspired by this, we added a similar feature to GHC's `--make` mode.

7.1 The `--make` compilation engine

In order to explain our parallel implementation, it is necessary to understand a little about how GHC's `--make` mode is structured. A compilation session proceeds in the following stages:

- Perform a dependency analysis of the modules in the program, and construct a module dependency graph.
- Flatten the module dependency graph in topological order.
- Compile each module.
- Link the final program to form an executable (this step is omitted for libraries).

The interface to compile a single module is (somewhat simplified):

```
compile :: ModSummary
        -> HscEnv
        -> IO (Maybe (ModDetails, Linkable)
```

where

`ModSummary`: contains information about the module to be compiled, including the filenames of the source file, object file and interface file.

`HscEnv`: contains all the information the compiler needs to know about its environment. Most notably, it contains

`HomePackageTable`: a mapping from `Module` to `ModDetails`, this mapping contains information about each of the modules in the current program (`ModDetails` is described below).

`ExternalPackageState`: this is a mutable variable, pointing to a structure containing information about all the pre-compiled modules that have so far been inspected. The `ExternalPackageState` is basically a cache; when information about another module is required, the structure is extended and the new version written into the mutable variable. GHC in fact reads information about pre-compiled modules lazily, so in fact `ExternalPackageState` can be modified at just about any point during compilation. Information is only ever added to the `ExternalPackageState`, never removed.

`ModDetails`: contains information about a single module, including the names and types of all functions exported by the module and definitions of `data` types, `classes` and `instances`. When

optimising, the ModDetails may also contain the *definitions* of functions, so that a function can be inlined at its call site in another module.

When compiling a module, the HomePackageTable must contain information about all the modules on which the current module depends, directly or indirectly.

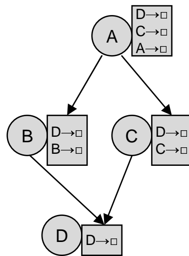
The caller of compile, namely the --make compilation engine, is expected to populate the HomePackageTable with the ModDetails for newly compiled modules before calling compile again. So the idea is that the HomePackageTable is gradually populated as we compile modules, and when the process is complete we have a HomePackageTable containing ModDetails for all the modules of the program.

7.2 Parallelising ghc --make

Now, let us consider how we might parallelise this process. One possibility is to have a compilation supervisor whose job it is to monitor the state of the compilation graph, and start compilations when all their dependencies have completed. It would be the compilation supervisor's job to keep track of the HomePackageTable.

This seems a reasonable approach, if a little heavyweight. For our experiment however, we chose a simpler, arguably more elegant, but perhaps slightly more opaque solution. The idea is to use concurrency to discover the implicit parallelism in the compilation graph, rather than figuring it out for ourselves. The key observation is that the compilation graph is just a dataflow graph, where each node is a compilation that can start as soon as its inputs are ready. We can implement a dataflow graph straightforwardly in Concurrent Haskell by forking a thread for each node, and having an initially-empty MVar to store the output of each node[3]. Each thread waits on the MVars for each of its inputs, so when all of them are ready it can begin its compilation and store the result in its own MVar.

We still need to construct the HomePackageTable for each compilation, and the easiest way to do this is for the HomePackageTable to be the data that flows along the edges of the graph:



The thread for each node performs the following steps:

- Perform a readMVar for each of the nodes that the current node depends on. Each of the readMVars will complete when the relevant node has been compiled, and its HomePackageTable is ready.
- Construct the HomePackageTable for this compilation by taking the union of the HomePackageTables of the dependencies (duplicate entries can be dropped, because they will be identical).
- Compile the current module.
- Extend the HomePackageTable with an entry for this module, and fill the MVar for the current node with this HomePackageTable.
- Now the thread can exit.

In practice, each node needs to store Maybe HomePackageTable, since a compilation may fail. If a node fails to compile, all the other nodes that depend on it will also fail, but others may continue to compile. It may be an improvement to this scheme to terminate the entire compilation session as quickly as possible after an error is discovered, but we have not implemented this yet.

We have so far assumed that we want to extract as much parallelism as possible from the compilation session, but in fact running more parallel threads than there are processors can lead to slowdowns due to contention. So in fact we want to place a limit on the number of parallel compilations that can be running at any time. We achieve this using a simple quantity semaphore: each compilation thread waits for a unit from the semaphore before starting its compilation, and returns the unit afterwards. The initial number of units in the semaphore is selectable via a command-line flag to GHC (ghc --make -j3, for example). It is advisable to make the number of compilation units equal to the number of capabilities in the parallel runtime, which is in turn equal to the number of real CPU cores[4].

7.3 Shared state in GHC

Having determined our top-level strategy for running parallel compilations, we must now investigate what shared state, if any, the parallel threads need to access, and how to make that access safe in a parallel setting.

The most important shared state is the ExternalPackageState, a mutable variable which can be updated at virtually any point during compilation, even from pure code; we treat it as a cache. If at some point during compilation GHC needs to know information about a pre-compiled module, the ExternalPackageState is first inspected to see whether the interface for that module has been loaded. If not, the interface is loaded and the ExternalPackageState is extended with information about the module.

To make access to the ExternalPackageState safe in a parallel setting, we did the obvious thing and replaced the ordinary IORef mutable variable with an MVar (using an STM TVar is planned for the future, and we expect to get some improvements from doing that).

There are a few other items of shared state in GHC, but they all follow the same kind of usage as the ExternalPackageState. For example, there is a global dictionary of strings called the FastString library, which makes string comparison cheap by assigning unique integers to strings; we had to protect access to this global dictionary with an MVar.

We also had to fix a bug in the generation of temporary filenames – the problem here was that the internal mechanism for generating new temporary filenames wasn't robust enough in that it didn't create the temporary file eagerly, so a second request for a temporary filename before the file had been created could return the same filename again. Due to the way GHC uses temporary files this bug had lain dormant until it was exposed by running compilation threads in parallel.

7.4 Results

Implementing our parallel version of GHC involved around 400 lines of changes[5] in total to the compiler sources. The compiler

[3] An STM TVar would be nicer. However, at the time we were implementing this experiment, the parallel STM implementation was not ready.

[4] We haven't measured whether any benefit can be had by treating an Intel Hyper-threading virtual CPU as a real CPU for the purposes of determining how many threads to use.

[5] Counting lines added plus lines removed, where an edited line counts as both a removal and an addition.

consists of around 65,000 lines of non-comment Haskell source, so the changes represent about 0.6% of the compiler.

We now give some measurements obtained by compiling a few programs using our parallelised GHC, on a dual Intel Xeon 2.4GHz machine with plenty of memory, that was otherwise unloaded.

The *maximum* speedup we could possibly hope to achieve is 2, because the test machine has 2 CPUs. However, there are several reasons why the actual speedup will be lower than 2:

- Garbage collection is still single threaded, and takes a not-insignificant amount of time (20-30% is typical; actual figures are given with the results).
- A speedup of 2 might not be available due to dependencies between modules. A program typically has a `Main` module which cannot be compiled in parallel with anything else, because it sits at the top of the dependency graph.
- The dependency analysis phase, which includes pre-processing of the modules (if any) is not parallelised.
- There is an overhead for compiling GHC itself for parallel execution. This overhead is small, as we illustrate below.

To test that the parallelism was working properly, we first compiled two identical modules simultaneously, changing the name of one of the modules. This is the ideal case: there are no dependencies between the modules, so this will give us an idea of the maximum available speedup. Figure 7 gives the results.

All our measurements are based on the average of three runs of the compiler. The speedup is measured using the elapsed time against the baseline `ghc-6.5` (the first row of the table).

We obtained a speedup of 1.32 on this example, which is certainly worthwhile, but clearly there's room for improvement.

In this example we increased the default heap size to 64Mb to reduce the costs of garbage collection, nevertheless GC still takes 15% of elapsed time for the sequential compiler, and 27% in the parallel case. Why does GC take more time when compiling in parallel? There are two reasons:

- GHC is carrying more live data because it is compiling two modules at a time rather than one, and
- The time spent compiling is shorter, so GC occupies a greater percentage of the total elapsed time.

Looking at the "user time" figures in the table, we can see that the parallel GHC required more execution time overall than the sequential compiler. There are several reasons for this:

- Roughly half of the increase was due to the extra GC time. This we know for sure, from our measurements; the following reasons are rather more conjectural.
- There will be increased load on the shared memory system from running two compilation threads in parallel. GHC is a fairly heavy user of memory.
- Threads may migrate from one processor to another. Our current scheduler implementation does not attempt to keep any affinity between capabilities and OS threads, or between Haskell threads and capabilities. So even if the OS implements affinity between OS threads and CPUs, it is entirely possible for a Haskell thread's execution to move between CPUs, largely invalidating the contents of both CPU's caches.

Discounting GC from the figures, we obtain a more respectable speedup of 1.54 for our parallel compiler versus the baseline. We don't expect to be able to achieve this figure in practice, because the GC simply has more work to do when compiling in parallel, but a multi-threaded GC would certainly help us get closer to this result.

Another way to look at these figures is to say that, based on the Linux user time figures, we are using about 1.5 CPUs over the lifetime of the program to obtain a 1.3 speedup, so we aren't monopolising all the processing power on the system. Hence the low speedup is due largely to a lack of parallelism rather than overheads in the implementation.

The final row of the table lists the results obtained by using the standard `make` tool to distribute multiple individual compilations over multiple processors. We can think of this as another point on the trade-off between sharing and copying: `ghc --make` shares a lot of data between compilations, but incurs some dependencies that reduce the parallelism. On the other hand, `make -j2` can parallelise the 2 compilations perfectly, but the separate compilations are duplicating some work; namely the reading of interface files for libraries and other modules.

The unmodified GHC 6.5 baseline was used for this test, but we set the heap size for each individual compilation to 32Mb, to give a fairer comparison against the other tests which all use 64Mb in total. The figures show that for this simple 2-module test, `make -j2` obtains a speedup of 1.52 versus the baseline `ghc-6.5 --make`, so at least in this simple case, `make -j2` beats our parallel compiler. However, as we increase the size of the programs in the tests that follow, we will see that the overhead of completely separate compilation will erode this advantage.

Next, we compiled Happy, a medium-sized program consisting of 15 modules with 1700 lines of non-comment code. Figures 8 gives the results. We tried two sets of compilations, firstly with optimisation turned off, and then with optimisation turned on (`-O`). So compiling a realistic program, our speedup has dropped from 1.3 to 1.2. This is still relatively respectable, however.

The rationale for turning optimisation on is to test a hypothesis: parts of the compiler that perform optimisation might involve less contention for the shared state, because optimisation is largely concerned with the code for the current module, compared to type-checking which accesses the shared state frequently to find information about imported modules. Nevertheless, our speedup in this example seems unaffected by turning on optimisation.

The `make -j2` test again comes out on top, but by a smaller margin. Furthermore, we can see by the user time figures that `make -j2` is performing more total work than the parallel compiler, although it is able to parallelise it more effectively.

Next, we compiled Anna from the `nofib` benchmark suite, a larger program consisting of 31 modules and 3800 lines of non-comment code. The results are shown in Figure 9.

In this example, the speedup without optimisation turned on is rather more modest. That might indicate that contention for the shared state in earlier phases of the compiler is in fact an issue, perhaps more so for larger programs.

Now, however, we see that `make -j2` is losing ground to `ghc-smp -j2`. The overhead of restarting the compiler and re-reading all the interface files for each compilation are too great to be overcome by the available parallelism.

7.5 How many thunk entry conflicts are there?

Our current implementation does not implement the recovery scheme described in Section 4, so we do not have accurate measurements for the amount of duplicate evaluation detected and recovered from. However, we have reason to believe that the figure will be small in the case of GHC: we measured the number of times a `GREYHOLE` thunk was entered, in a version of `ghc-smp` compiled with grey-holing support. With grey-holing turned on, the window during which two threads can begin to evaluate the same thunk is extremely small, so taking the average number of `GREYHOLE` entries over a number of runs gives us a reasonable estimate of the number of duplicate evaluations that would occur in a system performing

Test	User time (s)	Elapsed time (s)	GC time (%)	Speedup
`ghc-6.5 -H64m -O`	3.8	4.1	15%	1
`ghc-smp -H64m -O`	3.8	4.1	14%	1
`ghc-smp -H64m -O -j2`	4.6	3.1	27%	1.32
`make -j2`	4.8	2.7	?	1.52

Figure 7. Compiling "ideal" 2-module library

Test	User time (s)	Elapsed time (s)	GC time (%)	Speedup
`ghc-6.5 -H64m`	8.5	9.8	21%	1
`ghc-smp -H64m`	8.9	9.9	22%	0.98
`ghc-smp -H64m -j2`	10.2	8.2	25%	1.20
`make -j2`	12.3	7.9	?	1.24
`ghc-6.5 -H64m -O`	22.2	24.1	16%	1
`ghc-smp -H64m -O`	22.9	24.0	16%	1
`ghc-smp -H64m -O -j2`	25.9	20.1	21%	1.20
`make -j2`	28.0	18.4	?	1.31

Figure 8. Compiling Happy with and without optimisation

Test	User time (s)	Elapsed time (s)	GC time (%)	Speedup
`ghc-6.5 -H64m`	9.9	10.4	13%	1
`ghc-smp -H64m`	10.4	10.8	13%	0.96
`ghc-smp -H64m -j2`	12.4	9.2	23%	1.13
`make -j2`	15.8	12.6	?	0.82
`ghc-6.5 -H64m -O`	20.2	20.6	11%	1
`ghc-smp -H64m -O`	19.9	20.5	11%	1
`ghc-smp -H64m -O -j2`	23.3	16.9	21%	1.22
`make -j2`	27.4	20.7	?	1

Figure 9. Compiling Anna with and without optimisation

blackholing at thread descheduling, as described in Section 4. In such a system, some of these duplicate evaluations would be caught by the system (we don't know how many, though).

Over 10 runs of `ghc-smp -H64m -j2`, with grey-holing, compiling Happy, we saw an average of 11 grey-hole entries, with the maximum being 26 and the minimum zero. With lazy black-holing we saw rather fewer black-hole entries, but we did *not* observe any speedup in the grey-holing version, so we conclude that there was no significant duplicate work being performed.

7.6 Is GHC a realistic case study?

One might reasonably question whether GHC is a "typical" parallel application, and whether we can conclude anything based on the measurements above. Indeed, much of this paper has concentrated on how to efficiently deal with the issue of multithreaded contention for thunks, and yet apparently there is very little contention for thunks in parallel GHC.

It is not true to say there are *no* thunks in the data shared by parallel threads in GHC. The shared `ExternalPackageState` is full of unevaluated data: in fact, it is only the top-level mapping from `Module` to `ModDetails` that is mutated in the IO monad, much of the contents of the `ModDetails` itself is lazilly constructed. For example, the type of each identifier in a `ModDetails` will initially be represented by a thunk, that when evaluated will extract the type from the interface and convert it into GHC's internal representation.

We believe that minimising the contention for unevaluated data in parallel Haskell programs will generally be good advice. However, we do not believe that simplifying the system by requiring the programmer to be explicit about access to shared data is a viable alternative: the aim is to impose as little burden on the programmer as possible in order to make parallel execution highly accessible.

8. Related work

There is an enormous literature on parallel functional programming, to which we cannot hope to do justice here; for example, the recent book by Hammond and Michaelson has a bibliography of over 600 entries [7]. Other good surveys are available in [25, 13].

Driven by the desire for scalability and portability, almost all this research has focused on distributed-memory implementations, which have a separate address space, heap, and runtime system for each processor (e.g. GUM [24], Clean [7, chapter 15]). As Hammond & Michaelson put it "Over the last few years, it has become generally accepted that a message-passing interface (or, less generally accepted, a virtual shared-memory interface) can provide efficient access to a wide class of parallel architectures while enhancing the portability of parallel programs" [7].

These systems are mostly research prototypes. They require careful tuning to ameliorate the overheads of copying between heaps, and they lack the language features to support full-scale applications. The notable exception is Erlang, an explicitly-parallel functional language that uses message-passing to communicate between threads. Erlang is a mature full-scale language used in real applications in telecommunications [2].

We instead focus on the more limited goal of exploiting the shared-memory architecture of existing multi-processors and upcoming multi-core processors. In that sense, our work is closer to much older work, such as the $\langle \nu, G \rangle$ machine of Augustsson and Johnsson [3]. This implementation did locking on every thunk, but

back in the days when locking was less expensive relative to normal execution.

9. Further Work

This paper has shown how to support lightweight parallel thunk evaluation in which it is unnecessary to have "strong" synchronisation operations such as locks or compare-and-swap instructions on the fast-path code when allocating, evaluating and updating thunks. The result is that providing the *ability* to safely evaluate Haskell code in parallel has a cost, on average, of only 6%, almost ten times less than that of a lock-based design.

There are several interesting directions for future work based on this platform. As we move to working on larger multiprocessor machines, we will need to update the garbage collector so that collection work can be parallelised. This should be straightforward: there are numerous existing designs for parallel garbage collection on shared memory machines.

Of course, we also need to investigate more thoroughly where the remaining costs are being incurred, and to confirm our intuition that our lock-free design coupled with black-holing prevents almost all duplicate evaluation work; can we reduce the overhead still further?

More interestingly, we wish to return to the question of how parallel code is best expressed in a language like Haskell – for instance, combinators such as `par` and `seq` and ideas such as Strategies for orchestrating their use [23]. Another direction is to investigate feedback-directed schemes to attempt to identify thunks that may usefully be evaluated in parallel – for instance ones that are likely to be needed in the future, which represent a reasonable amount of work and where speculative parallel evaluation is unlikely to introduce contention with 'mainline' threads.

There are also interesting problems to investigate within the runtime system: our current design is simplistic with a single shared work queue without any notions of affinity. Furthermore, in systems comprising multiple cores spread across multiple CPUs and multiple points on an interconnect network, it may even be worth leaving processing resources idle if performance is being limited by contention in the caches or between the processors. These scenarios both suggest some kind of adaptive feedback-based scheme.

The implementation described here is publicly available from the Glasgow Haskell Compiler CVS repository.

Acknowledgments

Many thanks to Galen Menzel for his helpful feedback on earlier drafts of the paper.

References

[1] *Alpha Architecture Handbook*. Compaq Computer Corporation, 4th edition, Oct. 1998.

[2] J. Armstrong. The development of Erlang. In *ACM SIGPLAN International Conference on Functional Programming (ICFP'97)*, pages 196–203, Amsterdam, Aug. 1997. ACM.

[3] L. Augustsson and T. Johnsson. Parallel graph reduction with the $\langle \nu, g \rangle$-machine. In *ACM Conference on Functional Programming and Computer Architecture (FPCA'89)*, pages 202–213, London, Sept. 1989. ACM.

[4] R. Ennals. *Adaptive Evaluation of Non-String Programs.* PhD thesis, Cambridge University Computer Laboratory, 2004.

[5] C. Flood, D. Detlefs, N. Shavit, and C. Zhang. Parallel garbage collection for shared memory multiprocessors. In *USENIX Java Virtual Machine Research and Technology Symposium*, Monterey, CA, Apr. 2001. Usenix.

[6] K. Fraser. *Practical lock freedom.* PhD thesis, Cambridge University Computer Laboratory, 2003.

[7] K. Hammond and G. Michaelson, editors. *Research Directions in Parallel Functional Programming*. Springer-Verlag, 1999.

[8] T. Harris and K. Fraser. Language support for lightweight transactions. In *Object-Oriented Programming, Systems, Langauges & Applications (OOPSLA '03)*, pages 388–402, Oct. 2003.

[9] T. Harris, S. Marlow, S. P. Jones, and M. Herlihy. Composable memory transactions. In *ACM Symposium on Principles and Practice of Parallel Programming (PPoPP'05)*, June 2005.

[10] R. Jones. Tail recursion without space leaks. *Journal of Functional Programming*, 2(1):73–80, Jan 1992.

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

[12] K. Kawachiya, A. Koseki, and T. Onodera. Lock reservation: Java locks can mostly do without atomic operations. In *OOPSLA*, pages 130–141, 2002.

[13] H.-W. Loidl, F. Rubio, N. Scaife, K. Hammond, S. Horiguchi, U. Klusik, R. Loogen, G. J. Michaelson, R. Pena, A. J. R. Portillo, S. Priebe, and P. W. Trinder. Comparing parallel functional languages: Programming and performance. *Higher-order and Symbolic Computation*, 16(3):203–251, September 2003.

[14] V. J. Marathe, W. N. S. III, and M. L. Scott. Design tradeoffs in modern software transactional memory systems. In *Proceedings of the 7th Workshop on Languages, Compilers, and Run-time Support for Scalable Systems*, Oct. 2004.

[15] S. Marlow, S. P. Jones, and W. Thaller. Extending the haskell foreign function interface with concurrency. In *Proceedings of the ACM SIGPLAN workshop on Haskell*, pages 57–68, Snowbird, Utah, USA, September 2004.

[16] S. Marlow, S. Peyton Jones, A. Moran, and J. Reppy. Asynchronous exceptions in Haskell. In *ACM Conference on Programming Languages Design and Implementation (PLDI'01)*, pages 274–285, Snowbird, Utah, June 2001. ACM.

[17] W. Partain. The `nofib` benchmark suite of Haskell programs. In *Proceedings of the 1992 Glasgow Workshop on Functional Programming*, pages 195–202, London, UK, 1993. Springer-Verlag.

[18] S. Peyton Jones. Tackling the awkward squad: monadic input/output, concurrency, exceptions, and foreign-language calls in Haskell. In C. Hoare, M. Broy, and R. Steinbrueggen, editors, *Engineering theories of software construction, Marktoberdorf Summer School 2000*, NATO ASI Series, pages 47–96. IOS Press, 2001.

[19] S. Peyton Jones, A. Gordon, and S. Finne. Concurrent Haskell. In *23rd ACM Symposium on Principles of Programming Languages (POPL'96)*, pages 295–308, St Petersburg Beach, Florida, Jan. 1996. ACM.

[20] A. Reid. Putting the spine back in the Spineless Tagless G-Machine: An implementation of resumable black-holes. In *Proc. IFL'98 (selected papers)*, volume 1595 of *LNCS*, pages 186–199. Springer-Verlag, 1999.

[21] N. Shavit and D. Touitou. Software transactional memory. In *Proceedings of the 14th Annual ACM Symposium on Principles of Distributed Computing*, pages 204–213. ACM Press, Aug. 1995.

[22] H. Sutter. A fundamental turn toward concurrency in software. *Dr. Dobb's Journal*, March 2005.

[23] P. Trinder, K. Hammond, H.-W. Loidl, and S. Peyton Jones. Algorithm + strategy = parallelism. *Journal of Functional Programming*, 8:23–60, Jan. 1998.

[24] P. Trinder, K. Hammond, J. Mattson, A. Partridge, and S. Peyton Jones. GUM: a portable parallel implementation of haskell. In *ACM Conference on Programming Languages Design and Implementation (PLDI'96)*. ACM, Philadelphia, May 1996.

[25] P. Trinder, H. Loidl, and R. Pointon. Parallel and distributed Haskells. *Journal of Functional Programming*, 12:469–510, July 2002.

[26] D. L. Weaver and T. Germond, editors. *The SPARC architecture manual*. Prentice Hall, 1994. Version 9.