Feedback Directed Implicit Parallelism

Tim Harris
Microsoft Research, Cambridge, UK
tharris@microsoft.com

Satnam Singh
Microsoft Research, Cambridge, UK
satnams@microsoft.com

Abstract

In this paper we present an automated way of using spare CPU resources within a shared memory multi-processor or multi-core machine. Our approach is (*i*) to profile the execution of a program, (*ii*) from this to identify pieces of work which are promising sources of parallelism, (*iii*) recompile the program with this work being performed speculatively via a work-stealing system and then (*iv*) to detect at run-time any attempt to perform operations that would reveal the presence of speculation.

We assess the practicality of the approach through an implementation based on GHC 6.6 along with a limit study based on the execution profiles we gathered. We support the full Concurrent Haskell language compiled with traditional optimizations and including I/O operations and synchronization as well as pure computation. We use 20 of the larger programs from the 'nofib' benchmark suite. The limit study shows that programs vary a lot in the parallelism we can identify: some have none, 16 have a potential 2x speed-up, 4 have 32x. In practice, on a 4-core processor, we get 10-80% speed-ups on 7 programs. This is mainly achieved at the addition of a second core rather than beyond this.

This approach is therefore not a replacement for manual parallelization, but rather a way of squeezing extra performance out of the threads of an already-parallel program or out of a program that has not yet been parallelized.

Categories and Subject Descriptors D.1.3 [*Programming Techniques*]: Concurrent Programming—Parallel programming

General Terms Algorithms, Languages, Performance

Keywords Implicit Parallelism, Functional Programming, Haskell

1. Introduction

Parallel processing has now reached consumer desktop machines: all major vendors offer *multi-core* processors which are capable of executing 2, 4, and soon 8 threads in parallel. Where will we find enough profitable work for these threads to do so that a user perceives a 2-core machine as being better than a uni-processor, or a 4-core system better than a 2?

In this paper we return to an old idea: can we find this kind of parallelism *automatically* in existing programs? If we can do this then it would avoid programmers needing to grapple with explicit abstractions for parallel programming.

We work with programs written in Haskell (Peyton Jones et al. 1996), a pure, lazy, functional language which supports monadic I/O. In principle this language is a great fit for multi-core hardware: purity means that the compiler or run-time system can evaluate multiple parts of a program in parallel without needing to worry about data races. In practice we encounter five problems:

- Programs vary in the amount of parallelism that is actually available. As we show, some have a lot but some have very little.
- Even in programs with abundant parallelism, the work must be at a sufficiently coarse granularity that the parallel speed up compensates for the overheads introduced in managing the work.
- In languages with lazy evaluation, it is not immediately clear which pieces of computation will actually contribute to the 'real' work of the program. Performing un-needed work can harm performance – for example it can allocate a lot of memory and trigger extra garbage collections.
- Even if the source code for a piece of work appears pure, its compiled implementation may contain side effects (e.g. to perform memoisation).
- Although the core of a language may be pure, practical programs written in it are likely to involve some kinds of I/O or updates to mutable storage locations.

We base our work around the use of *thunks* allocated by optimized Haskell programs. Thunks provide a natural abstraction with which to look for implicit parallelism: in principle the run-time system can evaluate *any* of the thunks that have been allocated. We quantify this with a limit study examining the parallel speed-up we would achieve if we could evaluate *all* thunks as soon as they are allocated (Section 2).

This limit study is clearly unrealistic from a practical point of view – it assumes there is an infinite number of processors, and that there are no overheads introduced by running thunks in parallel. Our results show that many programs *do* actually exhibit substantial degrees of implicit parallelism in this simulated environment: 16 of our 20 test programs show at least a 2x speed-up and 4 show at least an 32x speed-up.

Can we achieve such a speed-up in practice? To do that we must be able to *predict* which thunks are likely to be good candidates for parallel execution. We want to select thunks that are likely to be needed by the program and which will run for long enough that the parallelism gained compensates for the overheads introduced. In Section 3 we show that, for most programs, we can make these predictions based on a thunk's allocation site. Restricting ourselves to selecting long-running thunks loses some potential parallelism, but we still see simulated 2x speed-ups on 7 of our tests.

In Section 4 we show how we use these predictions in practice in the Glasgow Haskell Compiler (GHC). When we predict that a thunk allocation site will produce useful work we 'spark' the

ICFP'07, October 1–3, 2007, Freiburg, Germany.
Copyright © 2007 ACM 978-1-59593-815-2/07/0010. . . $5.00

thunk, putting it into a shared pool of work that can be performed speculatively. We modified the GHC run-time system to add (*i*) a work-queue mechanism for managing sparked thunks, (*ii*) a new thunk-locking mechanism to prevent duplication of work between speculative execution and direct execution, and (*iii*) a mechanism to detect and prevent I/O operations that may reveal the presence of speculation.

Section 5 summarizes the results from our implementation of feedback-directed implicit parallelism (FDIP). We see performance improvements in all of the programs that our simulator predicted, but the practical overheads eat into the parallelism, leaving 10-80% speed-ups on the 7 programs that offered the potential for feedback-directed implicit parallelism. We show results for all of our benchmarks, including those which get a negligible speed-up and those which are slowed down. In practice we would use the simulation results from Section 4 to select whether or not to use FDIP for a particular program to try to counter this speculation risk.

As we conclude in Section 7, FDIP *can* find useful parallelism in some programs and get useful performance improvements on real commodity hardware. However, this approach is clearly not a silver bullet for parallel programming: it provides a way of squeezing extra performance out of the threads of an already-parallel program or out of a program that has not yet been parallelized.

In particular, we make the following contributions:

- We introduce a low-overhead mechanism for building parallelism profiles from optimized compiled code.
- We introduce heuristics that use these profiles to predict which thunk-allocation sites are likely to represent coarse-grained sources of parallelism.
- We show how these predictions can be used with full-run feedback to improve the performance of real large applications running on commodity multi-core hardware.

2. Profiling

In this section we examine the amount of implicit parallelism available in optimized programs compiled by GHC (specifically, we use ‘-O’ on the GHC command line). This is a limit study in which we ignore practical matters such as the overhead of scheduling work on multiple processors: as we will show, much of this parallelism is at too fine a granularity for it to be exploitable on stock hardware. However, the results are still important – the limit study gives an upper bound on what we could hope to achieve in practice.

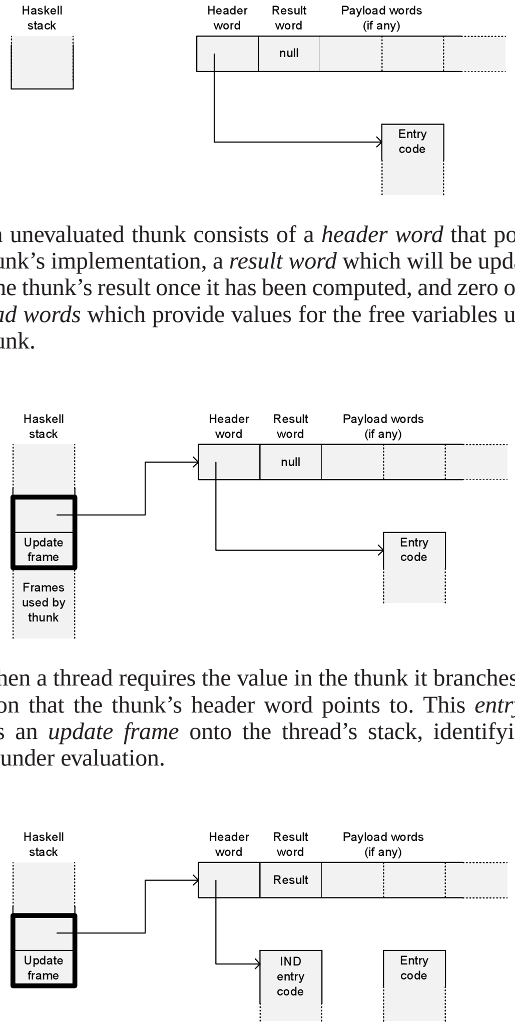
Section 2.1 outlines the implementation of lazy evaluation in GHC; the techniques it uses are central to our definition of implicit parallelism. Then, in Sections 2.2–2.5 we describe our techniques for tracing compiled Haskell programs and measuring the implicit parallelism that they actually contain.

2.1 Lazy evaluation in GHC

Lazy evaluation in GHC is based on the allocation and execution of *thunks* which represent suspended computations whose results may or may not be needed. Consider this expression as an example[1]:

```
let x = f(1) -- T1
    y = f(2) -- T2
in y + x
```

For clarity in these short examples we assume that ‘let’ is implemented by allocating a thunk for each of the variables it introduces, and that evaluating ‘+’ will require its left argument before its right one. In this example thunk T1 will compute the value of f(1), and similarly for T2 and f(2). In this case the value of y is needed almost immediately: the thunk is said to be *entered* (causing f(2) to be evaluated) and then, assuming f(2) terminates, the thunk is *updated* (overwritten in memory so that the result of f(2) is immediately available if it is needed again). Figure 1 shows the life-cycle of a thunk in more detail.



(a) An unevaluated thunk consists of a *header word* that points to the thunk’s implementation, a *result word* which will be updated to hold the thunk’s result once it has been computed, and zero or more *payload words* which provide values for the free variables used by the thunk.

(b) When a thread requires the value in the thunk it branches to the function that the thunk’s header word points to. This *entry code* pushes an *update frame* onto the thread’s stack, identifying the thunk under evaluation.

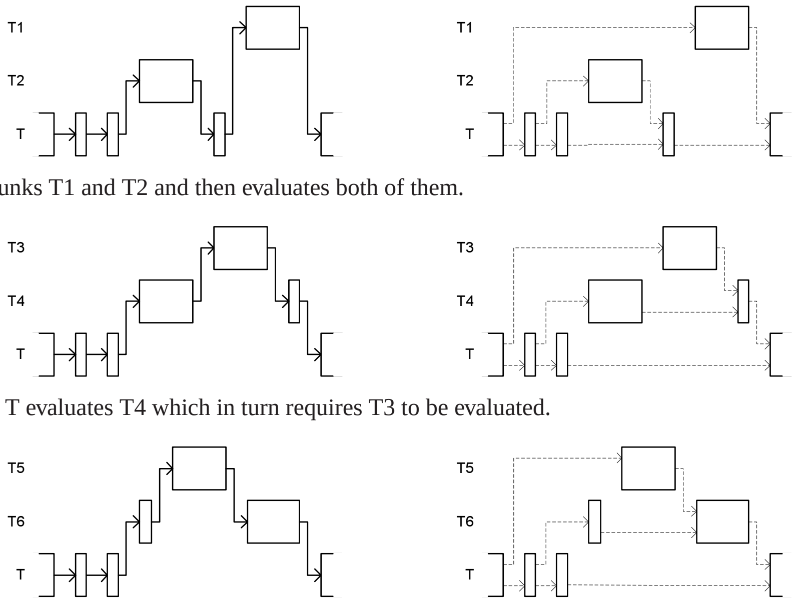
(c) The thunk’s evaluation is complete when execution returns to the update frame. The thunk is then *updated*, placing its value in the *result word* and replacing the header word with a pointer to *indirection entry* code (IND). If the thunk’s value is needed again then the IND code returns the contents of the thunk’s result word.

Figure 1. Thunk evaluation in GHC.

Many of the transformations made by an optimizing compiler focus on eliminating thunk creation in this kind of code: this is worthwhile because it avoids heap-allocating the thunk itself, eliminates the book-keeping work on entry and update, and can expose opportunities for further traditional optimizations (for instance if the body of f is inlined and f(2) can then be evaluated at compile time). In our example one would not expect a thunk to be created for T2 because the let expression will always need the result of f(2).

As we said in the introduction, our implementation and performance measurements are all designed to work with code with these optimizations applied. As we discuss in the conclusions in

[1] The syntax ‘--’ causes the rest of a line to be treated as a comment.



```
let x = f(1) -- T1
    y = f(2) -- T2
in y + x
```

(a) T allocates thunks T1 and T2 and then evaluates both of them.

```
let x = f(1)     -- T3
    y = f(2) + x -- T4
in y
```

(b) T allocates T3 and T4. T evaluates T4 which in turn requires T3 to be evaluated.

```
let x = f(1)     -- T5
    y = x + f(2) -- T6
in y
```

(c) T allocates T5 and T6. T evaluates T6 which immediately requires T5 to be evaluated.

Figure 2. Examples of thunk allocation and lazy evaluation.

Section 7, there is an interesting trade-off between parallelism and optimization opportunities.

Figure 2 illustrates some examples of lazy evaluation. In each case the `let` expression computes `f(1)+f(2)`. Looking first at Figure 2(a), following our first example, the code in the left column allocates two thunks on entry to the `let` expression. Evaluating `y+x` in the body of the `let` forces `T2` to be evaluated and then `T1`. The center column of Figure 2(a) shows what happens at run-time. The boxes show pieces of work involved in the implementation of each thunk, represented in execution order from left to right. If the `let` expression is in thunk `T` then execution starts with short steps that allocate thunks `T1` and `T2`. Execution then branches to `T2` which computes `f(2)` before returning to `T`. Execution proceeds to `T1` which computes `f(1)` and returns again to `T`. We return to the right column of the figure later.

In Figure 2(b) the addition is moved into one of the thunks. This is shown in the center by the fact that execution proceeds directly from `T4` to `T3` rather than back to `T` in between.

Finally, in Figure 2(c), the operands of the addition are swapped over. At run-time this means that `T6` almost immediately enters `T5` and the body of `T6` only executes when the result from `T5` has been computed.

2.2 Computing available parallelism

In the examples Figure 2(a)-(c) the total amount of work performed at run-time is the same. However, they differ in the amount of implicit parallelism that is available at the granularity of thunks.

In Figure 2(a) thunks `T1` and `T2` can start being evaluated in parallel as soon as they have been allocated. The same is true for most of the execution of the thunks in Figure 2(b), aside from the very last step of `T4` that must wait until `T3` has been completed. Conversely there is little parallelism available in Figure 2(c): the very first thing that `T6` does is to enter `T5` and so the bulk of `T6` cannot proceed until `T5` has been evaluated.

Do real programs behave like examples (a) and (b), or do they behave like example (c)?

We can answer this question by tracing the execution of real programs and examining how they use thunks. Our basic approach is (*i*) to modify GHC so that programs can be run in a tracing mode which records the dependencies between different pieces of thunks' execution, (*ii*) from this trace build a graph whose nodes are pieces of work labeled with the work's duration, and whose edges represent the dependencies that execution must respect, (*iii*) find the critical path through this graph from the thunk representing the start of execution to that computing the program's final result.

The critical path length gives us a lower bound on how fast this same execution could have run on a multi-processor machine. If there is little parallelism in this idealized view then there is certainly none in practice. Conversely, even if the critical path suggests that there is abundant parallelism then, of course, we must remain sceptical of the result; this limit study assumes an unbounded number of cores are available, that no additional book-keeping overheads are introduced, and no further slow-down is caused by additional pressure on the memory management subsystem or other resources in the run-time system or hardware.

At a high level there are three kinds of dependency between pieces of work involved in evaluating thunks:

- Pieces of work from the same thunk must execute serially. For example, in Figure 2(a), all of the pieces of `T` execute in order.
- A thunk cannot start being executed until it has been allocated. For example, in Figure 2(a), the thunks `T1` and `T2` cannot start to be evaluated until they have been allocated by `T`.
- If a thunk `U` requires the result of a thunk `V` then `U`'s execution cannot proceed past that point until `V` has been completed and the result supplied. This is what harms parallelism in Figure 2(c): `T6` requires the result of `T5` early in its own execution.

The right hand column of Figure 2 illustrates these dependencies for our three examples. The blocks represent the individual 'steps' of execution that each thunk is broken up into, and the dashed lines represent the dependencies that exist. For example, in Figure 2(a), the first step of `T` allocates `T1`, the second step allocates `T2`. The third step of T enters T2. Notice how this *does not* introduce a dependency from T to T2: T2 was available for execution at any point since its allocation. However, the fourth step of T is dependent on receiving the result of T2, and the final step of T on receiving the result of T1.

2.3 Implementation

It is vital that we minimize the impact that our trace-collection has on the program under test; many steps are very small and so there is a risk that the probe effect introduced by tracing will affect the results. We do this by trying to perform most work as part of garbage collection (GC). The current garbage collectors in GHC are all *stop-the-world* designs in which all mutator threads are suspended during collection. GC will already drastically perturb the contents of the processor's caches and so it provides a good time to add extra work associated with profiling.

In this limit study our input programs are all sequential. However, our tracing infrastructure does support programs using multiple Haskell threads multiplexed over a single operating system thread.

Events. We add tracing to each thunk at each stage of its allocation-entry-update lifecycle. We batch up trace-events in a simple in-memory buffer between GCs. To gather the step-dependency graphs we need to trace five kinds of event, (*i*) thunk allocation, (*ii*) thunk entry, (*iii*) thunk update, (*iv*) Haskell thread switches, and (*v*) thunk relocation events generated at GC time.

The first three kinds of event correspond to the steps in the thunk life-cycle in Figure 1. Each event includes the processor cycle-counter value when it is generated. In each case thunks are identified by their address in the heap and, in thunk allocation events, each static allocation site is identified by a unique integer ID allocated sequentially by the compiler.

Thread switch events are generated when the user-level scheduler in the run-time system switches to or from a given Haskell thread. A special thread ID is used to represent switching to work in the run-time system itself; this avoids us accounting work in (e.g.) the GC to the thunk which happened to be active when the collection was triggered. Thunk relocation events are generated at GC time and include the old and new locations of the thunk.

Trace processing. At GC time we process the in-memory buffer to generate a text file on disk that can be processed off-line to determine the implicit parallelism available.

In generating the on-disk file format we map each thunk to a unique sequence number; this provides a stable identifier for the thunk, meaning that trace-processing tools do not need to track the location of the thunk if it is relocated by the GC. The mapping from thunk addresses to unique IDs is held in a hashtable separate from the Haskell heap. This avoids perturbing the in-memory representation of thunks by extending them with space to hold the ID.

Each line in the on-disk file represents either (*i*) a relationship between two steps, or (*ii*) execution of a step, along with the cycle-count timing of that step. We use the notation X.Y to mean step number Y of thunk number X. For example, the execution shown in Figure 2(a) would be represented as follows, assuming that thunk T is numbered 100, T1 101, T2 102:

```
100.1 101.0 A-1234
100.1 100.2 S-700
100.2 102.0 A-1235
100.2 100.3 S-500
100.3 101.0 E
101.0 101.1 S-5000
101.1 100.4 U
100.4 100.5 S-500
100.5 102.0 E
102.0 102.1 S-6000
102.1 100.6 U
```

The A- lines indicate allocation dependencies, e.g. the first line shows that step 100.1 finished by allocating thunk 101. The number 1234 identifies the static allocation site in the program.

The S- lines indicate execution steps, e.g. the second line shows that step 100.1 took 700 cycles to execute measured with the user-mode processor cycle counter instruction. As Figure 2(a) shows, execution starts by T allocating the other thunks in a series of short steps.

The E- lines record when a thunk is first entered in sequential execution. They are superfluous from the point of view of the limit study but allow us to record the path taken by sequential execution for comparison with the possible parallel execution.

Finally, the U- lines record when one thunk completes execution by being updated. For example, the line 101.1 100.4 U shows that execution returns from thunk T1 (number 101) to thunk T (number 100).

Validation. We validated the tracing infrastructure by having it maintain a *shadow stack* of the thunks it believed were under evaluation in each Haskell thread. A *thunk entry* event pushes a new thunk onto the stack. A *thunk update* event pops a thunk from the stack. We report an error if (*i*) the thunk popped from the shadow stack does not match the thunk supplied as a parameter to *thunk update*, (*ii*) a thunk is entered before an allocation event is seen for it, (*iii*) a thunk is updated before an entry event is seen for it.

This approach was invaluable in finding places in the run-time system where thunks were manipulated without recording the necessary trace events[2]. Ultimately we could compile traced versions of GHC itself (including the Haskell libraries it uses) and run these without any errors being reported. Validation is often overlooked when collecting traces (Jain 1991) and our experience is that, without validation, subtle problems would have remained undetected and skewed the results by causing work to be attributed to the wrong thunk.

2.4 Replaying execution

We measure the implicit parallelism available in a program's execution by using the log to reconstruct a graph of the kind shown on the right hand side in Figure 2. Each execution step becomes a node in the graph and each allocation or update event becomes a dependence edge between steps from different thunks.

For this limit study we find the fastest possible execution of the trace by using a simple discrete event simulator modeling a machine with an infinite number of processors. Simulation events correspond to the completion of a step and, when an event fires, any subsequent step that is now eligible to run is started and an event scheduled for that step's completion.

The simulator outputs a summary of the execution time on this idealized parallel machine and a trace showing how the number of active cores varies over time.

2.5 Results

Figure 3 summarises the test programs that we used and their thunk usage. For each program we recorded (*i*) the total run-time of the original compiled without any instrumentation, (*ii*) the total run-time that our instrumentation accounts to the evaluation of thunks, (*iii*) the mean thunk size, and (*iv*) the fraction of allocated thunks

[2] For readers familiar with the GHC run-time system: (*i*) when dealing with static thunks, (*ii*) thunks allocated by the run-time system to raise asynchronous exceptions, (*iii*) AP_STACK thunks generated from update frames whose execution is suspended.

		Eval time (10^9 cycles)		Thunk size	Thunks
		Baseline	Instrumented	(cycles)	needed
atom	Floating point simulation, nofib/spectral	0.34	0.64 (185%)	254	94%
boyer	Gabriel suite 'boyer' benchmark, nofib/spectral	0.47	1.09 (232%)	284	61%
bsort-1	Sorting circuit model, locally written	0.77	1.18 (152%)	725	99%
bsort-2	Sorting circuit model, locally written	1.68	2.64 (157%)	1245	99%
cacheprof	Cache profiling tool, nofib/real	1.82	2.81 (154%)	1369	97%
calendar	Prints a given year's calendar, nofib/spectral	0.68	1.27 (186%)	585	99%
circsim	Circuit simulator, nofib/spectral	0.66	1.56 (237%)	315	84%
clausify	Put propositions into clausal form, nofib/spectral	0.58	0.85 (146%)	405	93%
compress	Text compression algorithm, nofib/real	4.61	5.30 (114%)	1969	99%
fft2	Fourier transforms, nofib/spectral	0.48	0.59 (123%)	1382	99%
fibheaps	Fibonacci heaps, nofib/spectral	0.26	0.25 (96%)	765	98%
hidden	Line rendering, nofib/real	0.70	1.02 (146%)	441	83%
lcss	Hirschberg's LCSS algorithm, nofib/spectral	0.33	0.47 (142%)	324	99%
multiplier	Binary-multiplier simulator, nofib/spectral	0.67	1.70 (254%)	213	99%
para	Paragraph formatting, nofib/spectral	1.98	3.26 (164%)	1301	92%
primetest	Primality testing, nofib/spectral	2.33	2.21 (95%)	9266	99%
rewrite	Equational rewriting system, nofib/spectral	0.29	0.23 (80%)	1059	82%
scs	Circuit simulator, nofib/real	0.78	0.98 (125%)	862	84%
simple	Hydrodynamics and heat-flow, nofib/spectral	1.80	2.87 (159%)	526	99%
sphere	ray tracer, nofib/spectral	0.70	0.74 (106%)	1832	85%

Figure 3. Summary of the test programs used.

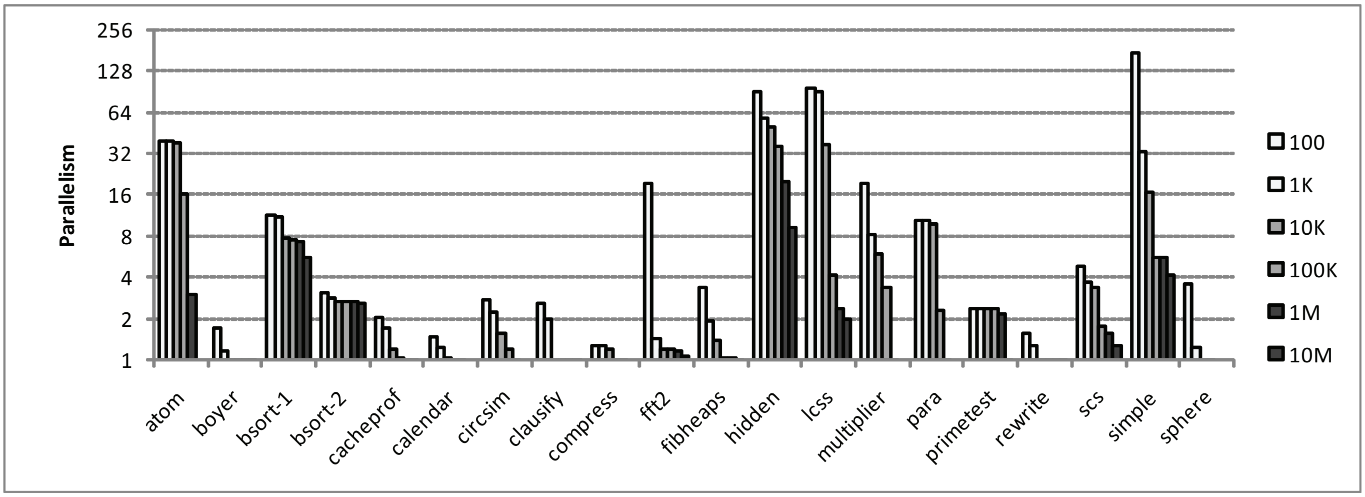


Figure 4. Implicit parallelism for our test programs with different execution thresholds. The y-axis shows the parallelism achieved, so 1 means 'the same as sequential execution', and 2 'twice as fast as sequential execution'.

that are needed by the computation. The difference between the first two numbers gives an upper bound on how much our instrumentation adds to the time spent evaluating thunks; as expected we tend to see larger differences when thunks are short.

Our test programs were selected as follows. We started with the full 'real' and 'spectral'[3] sections of the nofib benchmark suite, along with a number of other locally-written programs. We removed programs which could not be immediately built with our tracing system (for example because of dependencies on Haskell packages we had not installed). We also removed programs which we could not readily configure to run for at least 1s so that the running time is large compared with our measurement precision. Finally, we examined the tests and made sure that they did not simply repeat a small code fragment in a way that would lead our system to conclude that each repetition could run in parallel; it would be misleading to generalize from a test program where repeating an operation 500 times allows a speed up of a factor of 500 through parallelism. We did this by confirming that the parallelism was independent of the number of repetitions.

We preliminarily experimented with whole-program instrumentation, using an instrumented version of the Haskell libraries along with instrumentation in the program under test. Using instrumented libraries did not affect the implicit parallelism in these tests and so, in all the measurements reported here, we use ordinary uninstrumented libraries. From a practical viewpoint, this means that our results correspond to applying FDIP without needing to recompile or modify libraries on a per-program basis.

[3] 'Real' contains real applications written in Haskell. 'Spectral' contains substantial kernels of applications.

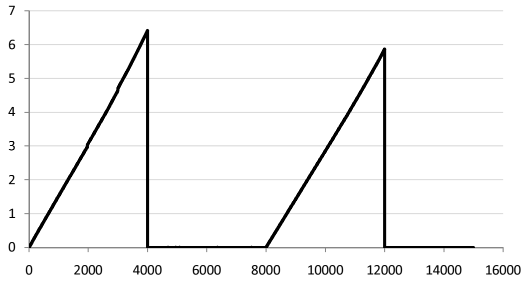


Figure 5. An allocation site that should *not* be sparked despite having a high mean execution time. The x-axis shows the thunks created at the site in allocation order, the y-axis shows their execution time in 1M-cycle units.

Figure 4 summarizes our results, showing the parallelism achieved for each of the programs and for a range of execution time thresholds. The programs are shown on the x-axis, and the speed-up achieved is shown on a logarithmic scale on the y-axis: 1 means 'the same as sequential execution', 2 means 'twice as fast as sequential execution', and so on.

For each program we show a cluster of bars corresponding to different thresholds on thunks' execution times: only thunks that take *longer* than this threshold are executed in parallel, shorter thunks are executed sequentially at the point that they are first entered. We define the execution-time of a thunk as the number of cycles in the sequential trace between when it is entered and when it is updated. The lowest threshold of 100 cycles causes all thunks to be executed in parallel as soon as they are allocated: the per-thunk logging operations add around 100 cycles to each thunk's execution.

Behavior clearly varies between programs. Some programs show virtually no possible parallelism even for very small thresholds – boyer, calendar, and rewrite. Others provide some parallelism, but only with very small thresholds – fft2 and sphere. It is unlikely that this could be exploited in practice; the book-keeping cost of dispatching a thunk for execution on another thread is likely to be at least 10k cycles.

However, some programs do continue to offer parallelism with thresholds over 10K: atom, bsort-1, bsort-2, hidden, lcss, primetest and simple all show a potential speed up of 2x or more even when using a 1M cycle threshold.

3. Feedback generation, recompilation

The results from the limit study show that several programs do provide appreciable amounts of implicit parallelism, even when restricting ourselves to large thunks of 1M cycles or more. Can we actually obtain any of this parallelism in practice?

The problem is that, to obtain this parallelism at run-time, we need to *predict* which thunks will be worth running concurrently with application threads. To take a contrived example, suppose that the limit study shows that a program has 4-way implicit parallelism. This is easy to exploit if the program allocates exactly 4 long running thunks – simply enqueue all of the thunks for concurrent execution. However, Figure 3 shows that real programs allocate a vast number of thunks, some long running, some short running, and some which are allocated but never evaluated. To generate good-quality feedback we must predict which thunks are (*i*) likely to represent pieces of work that are needed by the application, and (*ii*) likely to represent large pieces of work.

Our basic approach is to use static allocation sites to predict whether or not thunks will meet these criteria and to make a binary spark / not-spark prediction for each site. Intuitively thunks allocated at the same point in the program may be expected to be used in similar ways. Earlier work in Haskell (Ennals 2004) has suggested that this is true in practice, and earlier work on storage management in other languages has shown static allocation sites to be a predictor of properties of the data's usage (Harris 2001). We justify this decision later in this section by comparing the parallelism achieved by our per-allocation-site decisions with that achieved in our limit study.

Selecting thunks that will be needed. We do not want to spark thunks that are not needed: even if there are idle cores available then the extra work will add pressure on the garbage collector. We select allocation sites based on a simple threshold on the fraction of thunks allocated at that site in the profiling run which were eventually entered. In our results this parameter has little effect on the feedback quality over a wide range: our results use a threshold of 3/4 but the number of sparked thunks is unaffected up to a ratio of at least 255/256, and the run-time performance is unaffected by requiring a 1/1 ratio. One explanation of this, at least in our test programs, is that allocation sites producing long-running thunks are also sites that produce thunks that are always needed.

Selecting thunks that provide coarse-grained parallelism. The key problem, however, is selecting thunks that are likely to represent a substantial piece of work at run-time: we want to ensure that each thunk we spark will provide enough work to compensate for the overhead of sparking it.

In Section 2 we defined a thunk's execution time as the number of processor cycles in the sequential trace between when the thunk was entered and when it is updated. We initially hoped to use an allocation site's average execution time to select sites to spark. However, *this does not work well*. For example, consider this function:

```
noRealWork :: Int -> Int
noRealWork 0 = 0
noRealWork x =
  let t = noRealWork (x-1) -- T1
  in t
```

The function `noRealWork` recurses deeply before returning 0. If called with a large parameter then a lot of thunks will be allocated at `T1` and, if the recursion is deep enough, the mean execution time will make `T1` look as though it is worth sparking. We might ideally spark the first thunk allocated in a given recursion, but not the subsequent ones. However, this cannot be expressed by a binary spark / not-spark decision at the allocation site.

Figure 5 shows a second problematic example taken from an actual program (`bsort-2`). The graph plots the execution time of 15 000 thunks from a given allocation site in units of 1M cycles. The execution times vary from around 0 up to 6.2M cycles – although some short-running thunks are allocated at this site, the site as a whole still has a mean workload substantially over 1M cycles. Examining the behavior of this allocation site in more detail shows that the first peak comprises 4 000 thunks which are allocated together near the start of the program. Thunk 4 000 is entered first. Thunk 4 000 then enters thunk 3 999, which enters thunk 3 998 and so on in turn. Only a few hundred cycles of work are performed at each stage. The long running time of e.g. thunk 3 999 represents almost exactly the work as that of 4 000. However, while this case is similar to `noRealWork`, we would ideally spark the *last* thunk allocated in the recursion rather than the *first* thunk.

We therefore took the approach of *not* sparking thunks created by allocation sites like these. Instead we try to identify allocation sites where each sparked thunk provides its own 'real' work. We do this by considering the *total work* of an allocation site which we

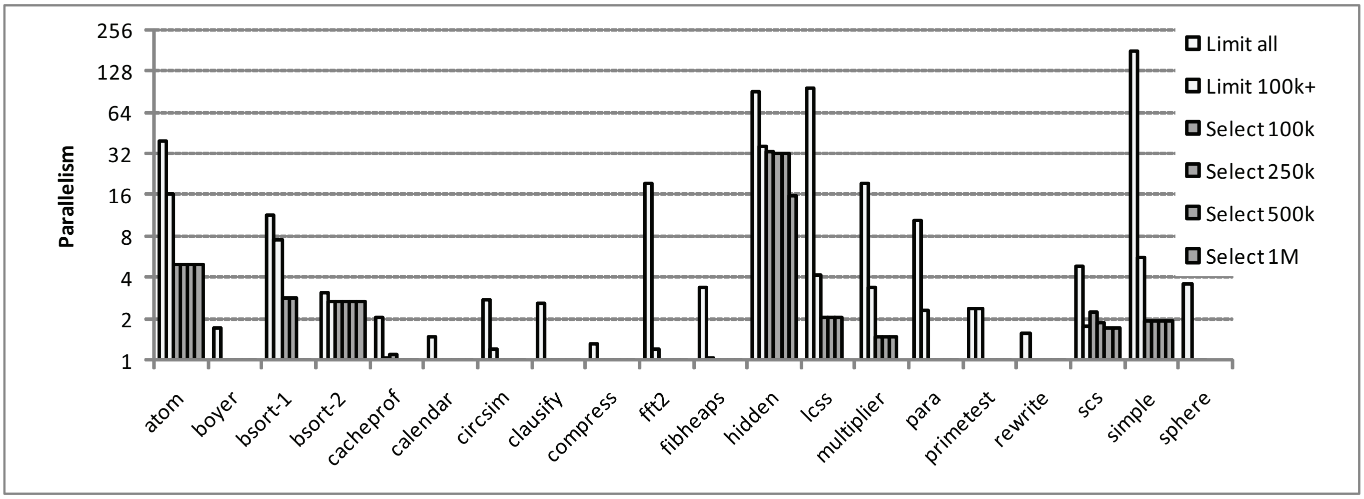


Figure 7. Simulated performance for sparking based on allocation site. We vary the mean work-size threshold between 100K-1M cycles with a fixed 3/4 needed-thunk threshold.

define as the time during which at least one thunk allocated at that site is under evaluation, and then find the *mean work* by dividing this by the number of thunks allocated. For example, in Figure 5, a total of around 6.2M cycles is spent during the evaluation of the first 4 000 thunks, and so the mean contribution of these thunks is only 6.2M / 4 000 = 1.5K cycles.

Figure 6 shows how we compute an allocation site's total work in a single pass over the trace file. We keep various mappings indexed by thunks and by allocation sites, updating these as the algorithm runs. `ThunkToSite` maps thunks to the allocation site that created them. `EntryTime` records the earliest sequential execution time at which any currently-active thunk at a given site was entered. `EntryCount` records the number of currently-active thunks at a given site. `TotalWork` records the total work accounted to the allocation site. `CurTime` records the current simulated time, advanced when processing each step in the trace.

This approach deals with allocation sites generating thunks which recursively enter other thunks from the *same* site. What about *different* sites? For example:

```
inner :: Int -> Int
inner x =
  let t = realWork x -- T2
  in t

outer :: Int -> Int
outer x =
  let t = inner x -- T3
  in t
```

If we assume `realWork` is a long-running computation then `T2` and `T3` will *both* appear to generate long-running thunks even though `T3` itself generates little work beyond its calls to `inner`.

Do we need to avoid sparking both allocation sites? We considered an extension to the algorithm in Figure 6 which took a *tentative* selection of thunks and replayed the execution trace to identify the work provided by each tentatively selected thunk above and beyond that provided by other tentatively selected thunks that it recursively entered.

We have not yet implemented that extension. There are two reasons. First, it requires *two* passes over the trace which would prevent us from applying it on-line in future work. Second, the depth of recursion through wrappers, and hence the number of small wrapper thunks created, is bounded above by the number of thunk allocation sites that we select for sparking – beyond this depth one of the sites involved must be providing 'real' work.

3.1 Simulating the performance of our selections

We re-ran our limit study to explore the impact of the different mean-work thresholds. Figure 7 illustrates this. We use the same axes as our earlier results from the limit study and, for each program, we include two bars from the limit study for comparison: the parallelism achieved by sparking all thunks ('Limit all') and the parallelism achieved by sparking all thunks whose execution time is 100k cycles or more ('Limit 100k+'). We then add four bars showing the speed-up when we select allocation sites using 100k, 250k, 500k and 1M-cycle mean-work thresholds.

In many cases the simulated results are identical for 100k-500k thresholds. This is because the sparking decisions are identical: the parallelism is coming from a particular allocation site or set of allocation sites, and disappears entirely when the threshold exceeds this site's mean work.

We lose some parallelism going from the limit study to the per-allocation-site predictions. We examined why this is in `atom` which is the program with the largest drop when comparing 'Limit 100k+' with 'Select 100k'. In this case it is because of thunk allocation sites that generate thunks with a variety of work times rather than because of allocation sites that generate thunks that are not always needed. We did not study the other tests in detail, but high fractions of needed thunks in Figure 3 suggest that this same conclusion is likely to be true in all of the programs where we lost parallelism.

Figure 8 shows the number of allocation sites selected for sparking with a 100k threshold, the number of thunks that they allocate, and the fraction of these that are needed by the program. The low numbers – both of sparked sites and sparked thunks – justify our decision not to avoid sparking short wrapper thunks: the total number of wrapper thunks must be low.

4. Run-Time system

The previous section showed that making per-allocation-site predictions of which thunks to spark is able to achieve a parallel speed-up in several of our test programs. In this section we show how we achieve a speed-up in practice rather than just in a simula-

```
ThunkToSite :: Thunk -> AllocationSite
EntryTime :: AllocationSite -> Time
EntryCount :: AllocationSite -> Int
TotalWork :: AllocationSite -> Time
CurTime :: Time

foreach record r {
  // Update the total sequential work performed
  if (r is step) {
    CurTime += r.size;
  }

  // Thunk allocation: record mapping from this
  // thunk to its allocation site
  if (r is allocate-thunk) {
    ThunkToSite(r.thunk) = r.site
  }

  // Thunk entry: check whether we are already
  // evaluating a thunk from this allocation site
  if (r is enter-thunk) {
    site = ThunkToSite(r.thunk);
    if (EntryCount(site) == 0) {
      EntryTime(site) = CurTime
      EntryCount(site) = 1
    } else {
      EntryCount(site) ++;
    }
  }

  // Thunk completion (update): check whether
  // we are still evaluating other thunks from
  // this allocation site
  if (r is update-thunk) {
    site = ThunkToSite(r.thunk);
    EntryCount(site) --;
    if (EntryCount(site == 0)) {
      // Last thunk at this site: record
      // elapsed time since entering the
      // first thunk
      TotalWork(site) += CurTime - EntryTime(site)
    }
  }
}
```

Figure 6. Pseudo-code to compute the total work of an allocation site in a single pass over a trace file.

	Sites sparked	Thunks sparked	Thunks needed
atom	3	801	100.00%
boyer	1	1	100.00%
bsort-1	5	4097	100.00%
bsort-2	14	469	100.00%
cacheprof	21	9161	100.00%
calendar	1	400	100.00%
circsim	11	3604	100.00%
clausify	1	20	100.00%
compress	4	4	100.00%
fft2	15	2573	100.00%
fibheaps	2	17	100.00%
hidden	15	7284	92.77%
lcss	5	776	100.00%
multiplier	10	4008	99.98%
para	5	31351	100.00%
primetest	4	602	100.00%
rewrite	4	20	100.00%
scs	62	8805	98.17%
simple	104	532	100.00%
sphere	3	3	100.00%

Figure 8. The number of thunks sparked during the simulation, and the percentage of sparked thunks that are actually needed.

tor. The main idea is to add additional Haskell threads whose sole job is to evaluate sparked thunks speculatively (Section 4.1). The sparked thunks are managed with a simple work-stealing system (Section 4.2) with the addition of locks to prevent duplication of the same piece of work between a speculative thread and an application thread (Section 4.3). These mechanisms replace GHC's existing support for sparking thunks on shared-memory multi-processor machines. We summarize the differences and compare the performance of these two approaches in Section 4.4.

We must be careful to preserve the semantics of the original program. The problem here is that GHC provides two operations through which I/O can be preformed outside the normal monadic I/O system:

```
unsafePerformIO :: IO a -> a
unsafeInterleaveIO :: IO a -> IO a
```

Evaluating `unsafePerformIO x` causes the I/O action `x` to be performed immediately. This 'back door' can be used for ad-hoc profiling and debugging, e.g. to probe which order different expressions are evaluated. It can also be used to encapsulate I/O operations which the programmer asserts are free from side effects and independent of their environment.

Performing `unsafeInterleaveIO y` provides a way to lazily defer I/O: the action `y` is performed when the value yielded by `unsafeInterleaveIO` is demanded. This is used in the GHC libraries to implement lazy file reading.

We must be careful not to perform unsafe I/O inside speculative thunks: we do not know whether or not the speculative work should be performed and, even if it should be performed, we do not want to re-order the I/O operations that are executed.

We deal with this problem by dynamically detecting attempts to perform unsafe I/O operations while speculating. If an unsafe I/O operation is attempted then we suspend speculation of the current spark. The suspension mechanism means that if the thunk is subsequently entered then evaluation will resume at the I/O operation. We implement this by defining a new I/O action:

```
nonSpeculatively :: IO ()
```

This has no side effect in an application thread, but it suspends speculation if it is attempted during speculation. We then redefine `unsafePerformIO` and `unsafeInterleaveIO` in terms of the original versions of these operations:

```
-- Perform IO action 'm' only if running in an
-- application thread
doNonSpeculatively :: IO a -> IO a
doNonSpeculatively m = do { nonSpeculatively ; m }

unsafePerformIO :: IO a -> a
unsafePerformIO x =
  oldUnsafePerformIO (doNonSpeculatively x)

unsafeInterleaveIO :: IO a -> IO a
unsafeInterleaveIO y =
  oldUnsafeInterleaveIO (doNonSpeculatively y)
```

Although these two unsafe operations are similar, note the asymmetry in what is being prevented. Speculative work must not be allowed to call unsafePerformIO. In contrast, speculative work can call unsafeInterleaveIO but evaluating the result of type a must be prevented.

4.1 Speculative worker threads

The GHC run-time system performs its own user-level scheduling of Haskell threads. This lets it support a much larger number of Haskell threads than most OS threading libraries would provide.

The maximum number of Haskell threads that can run in parallel is supplied as a start-up parameter to the run-time system. In the terminology of the GHC run-time system, this creates a number of *capabilities*. Each capability holds resources that should be per-core – primarily a run queue of Haskell threads being scheduled over that capability and a local memory allocation region. Haskell threads migrate between capabilities over long timescales for load balancing. The number of capabilities is typically tuned by the user to the number of cores available on the machine.

To deal with speculation we add an additional *speculative work thread* to each capability. In most respects this is an ordinary lightweight Haskell thread which loops attempting to pick up and evaluate thunks that have been sparked for speculative execution.

It differs from ordinary threads in that (*i*) it has lower priority than other Haskell threads on the same capability, (*ii*) if it attempts an operation that would block then the speculative evaluation it was attempting is *suspended* instead of the thread waiting. The intuition behind this is that the blocking operation represents a long delay and so the speculative thread should continue performing useful work from another spark rather than actually waiting.

Speculation is suspended by updating the thunks under evaluation with new thunks that will resume the computation at the point it was suspended. This mechanism is already used in several places in the GHC run-time system – e.g. to save partial evaluation of thunks that are interrupted by the delivery of an asynchronous exception.

4.2 Managing sparked thunks

We manage the sparked thunks with a basic work-stealing system. Each capability has its own spark-pool into which threads running on it publish references to the thunks that they spark. They do this by passing the newly-allocated thunk to a new function sparkSpeculation exported by the GHC run-time system. These calls are intended to only by added by the FDIP tool, not directly by the programmer. If the capability's speculative work thread runs then it preferentially takes sparks from its own pool. Otherwise it takes sparks from a random capability's pool. We take a random thunk from the pool.

Since we assume that sparks generated by speculation represent large pieces of work we use simple per-spark-pool mutexes and a steal-one policy. The benefits of a finer-grained work-stealing system (Blumofe and Leiserson 1994; Hendler et al. 2005) would be negated by the thunk-locking mechanism described below.

4.3 Preventing duplicate evaluation

The existing implementation of thunk evaluation in GHC is designed to be thread-safe (Harris et al. 2005). The approach taken there is to make it safe for concurrent threads to enter, evaluate, and update the same thunk at the same time. This can lead to duplicate evaluation. However, referring back to Figure 1, such duplicate evaluation will only occur if a second thread enters a thunk in the window between a first thread entering it (Figure 1(a)) and updating it (Figure 1(c)).

This approach is effective in practice because it avoids the cost of locking thunks while they are under evaluation. However, this relies on a number of assumptions about how thunks are used. First, many programs are single-threaded: duplicate evaluation cannot occur in these. Second, in programs using multiple threads, different threads are often working on different parts of a problem. This can lead to an affinity between threads and thunks. Third, many thunks are short-running and so the window during which duplicate evaluation is possible is short (our statistics in Figure 3 reconfirm this) and the cost of occasional duplication is low.

The thunks sparked by FDIP do not behave like this. Most importantly, we deliberately attempt to spark long-running thunks: the cost of duplicate evaluation is high, and the window during which it is possible is long. Furthermore, the work-stealing model introduces correlations between the thunks being entered speculatively and those entered by application threads. For example, this happens if an application allocates a series of thunks, adds these to the spark-pool, and then proceeds to evaluate one of the thunks itself.

To avoid these problems we introduce locking on thunks, *but only on ones that are added to the spark-pool*. As our simulation results show in Figure 8, the number of sparked thunks is low compared with the total number.

Sparked thunks are locked when they are under evaluation, either by a speculative thread or by an application thread. If an application thread tries to enter a locked thunk then it blocks until the thunk's value is available. If a speculative thread attempts to enter a locked thunk then that suspension is suspended and it can resume speculation from another spark. The intuition behind this is that the locked thunk represents a large piece of work (because it was selected for sparking) and so the speculative thread should continue performing useful work from another spark rather than waiting.

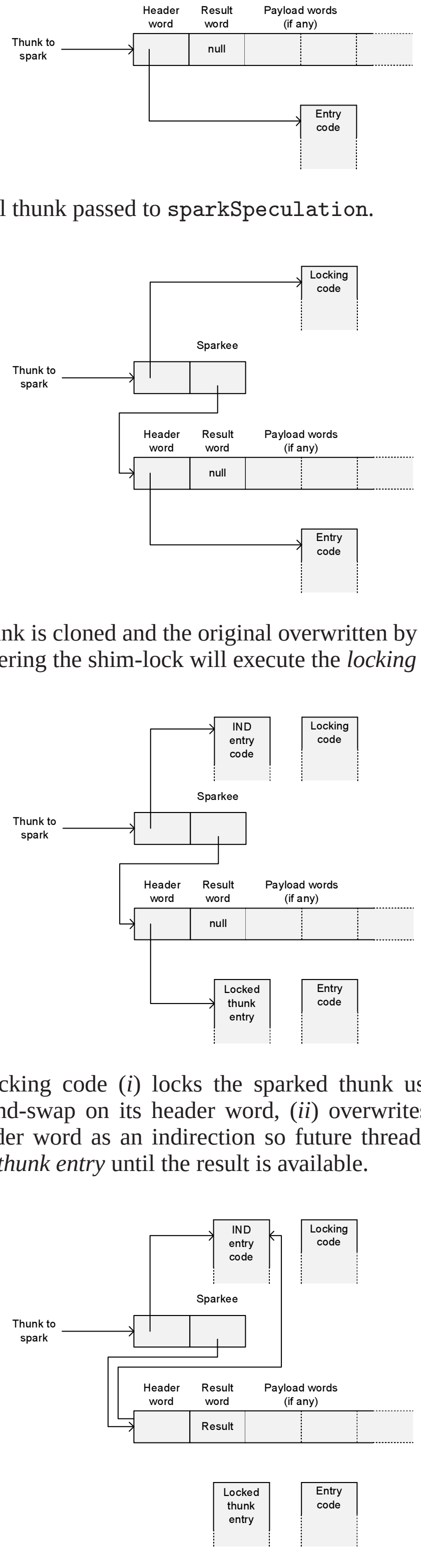
We introduce this locking in the sparkSpeculation function that is called at the allocation sites selected for feedback-directed sparking. Figure 9 shows what we do. The original thunk (Figure 9(a)) is cloned and overwritten by a *shim-lock* object (Figure 9(b)). Remember that we call sparkSpeculation on a newly-allocated thunk: it is thread-local at this point. Overwriting the thunk means that all references which would have referred to the original thunk will now refer to the shim lock.

When the shim-lock is entered the locking code is executed instead of the original thunk's entry code. This uses an atomic compare-and-swap on the header word of the clone, attempting to replace the pointer to its *entry code* with a pointer to a new *locked-thunk entry*. The first thread to attempt this will succeed: it has locked the thunk. The lock holder branches to the original entry code to evaluate the thunk. Other threads branch to the locked-thunk entry code which blocks them (in the case of application threads) or suspends their speculation (in the case of speculating threads). The shim-lock is released by the normal thunk-update operation: as Figure 9(d) shows, the update overwrites the header word once again.

As an optimization, the lock holder can overwrite the shim-lock's entry code with an indirection. This avoids any subsequent threads from attempting to acquire the lock: they will dereference the indirection to either (*i*) enter a further indirection to the thunk's result value, (*ii*) enter the locked-thunk entry code.

4.4 Comparison with GHC 6.6

We compared the performance of the new implementation of sparked thunks with the existing implementation in GHC 6.6. The existing implementation is used in Concurrent Haskell to build the par combinator. The primary differences between the implementations are that (*i*) we introduce locking around sparked thunks, whereas GHC 6.6 does not, (*ii*) our speculative threads pro-actively steal work from other spark pools, whereas GHC 6.6 periodically pushes work from one spark pool to another pool that is empty, (*iii*) we use long-running Haskell worker threads for speculation,



(a) Original thunk passed to `sparkSpeculation`.

(b) The thunk is cloned and the original overwritten by a *shim-lock* object. Entering the shim-lock will execute the *locking code*.

(c) The locking code (*i*) locks the sparked thunk using atomic compare-and-swap on its header word, (*ii*) overwrites the shim-lock's header word as an indirection so future threads branch to the *locked thunk entry* until the result is available.

(d) The normal thunk-update operation releases the shim-lock, leaving a double-indirection to be removed at GC time.

Figure 9. Shim-lock implementation.

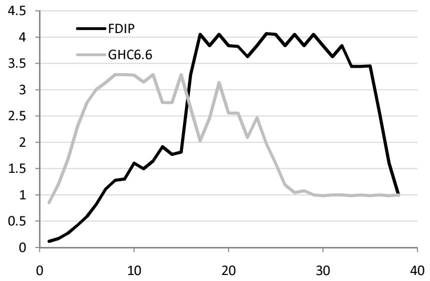


Figure 10. Parallel speed-up on a 4-core processor (y-axis) against thunk granularity (x-axis) for GHC 6.6 and for the modifications we introduced to support FDIP. For this workload the GHC 6.6 implementation provides a better parallel speed-up for very short-running thunks, but the FDIP mechanisms provide a near-optimal 4-fold speed-up for the longer thunks that we are interested in.

whereas GHC 6.6 creates a new Haskell thread for each spark that is 'fished' from a pool.

For our comparison we used a naive parallel implementation of `fib n`, computing the $\texttt{n}^{\text{th}}$ Fibonacci number by simple recursive evaluation of `fib n-1` and `fib n-2`. The call to `fib n-1` is sparked and so, in principle, each level of recursion introduces parallelism. The parallel work becomes very fine grained if we continue sparking thunks all the way down to `fib 1` and so we set a threshold below which we switch to sequential evaluation. This lets us control the minimum size of the thunks we spark.

Figure 10 shows the parallel speed-up achieved for various thresholds in the computation of `fib 38` (selected to take around 1s on our test machine). GHC 6.6 achieves no parallel speed-up for high thresholds: the spark-pushing algorithm is tied to allocation work and does not run frequently with this workload. Speed-ups of 2x - 3x are achieved on a 4-core machine for thresholds from around 25 (mean thunk size 4M cycles) down to 5 (mean thunk size 2.5k cycles). The modifications we made achieve near-optimal 4x speed-ups for larger thunks because work can be stolen by idle cores as soon as it has been sparked. This continues down to around threshold 17 corresponding to 50k cycle thunks. Below this performance falls off rapidly and is worse than GHC 6.6. However, as we showed in the limit study in Figure 7 we are primarily interested in thunks above this size where the FDIP modifications perform well. The performance of the FDIP run-time modifications led us to pick 100k as the threshold for selecting a thunk allocation site for sparking.

There are two reasons for GHC 6.6 performing better with smaller thunks: (*i*) we incur run-time and space costs from locking sparked thunks whereas GHC 6.6 does not, (*ii*) our simple work-stealing system involves locking the local spark pool when sparking a thunk whereas GHC 6.6's spark pools are each entirely thread local. We believe the peak speed-up in GHC 6.6 is lower because of the cost of creating a new Haskell thread for each spark fished from the pool. The two schemes are clearly complementary and we would envisage combining them based on thunk size.

5. Results

We tested our FDIP implementation using the same benchmarks we used in the limit study in Section 2. These do not ordinarily include separate input sets for testing feedback-directed techniques. We therefore modified the input sets, where practical, to provide different input problems for our timed experiments from our profiling runs. This is to ensure that sources of parallelism we find are due to the structure of the program under test rather than simply

atom	Longer simulation run
boyer	Unchanged
bsort-1	Larger input
bsort-2	Larger input
cacheprof	Unchanged
calendar	More repetitions
circsim	Longer simulation run
clausify	More repetitions
compress	Longer input text
fft2	Larger input problem
fibheaps	Larger input problem
hidden	More complex scene to render
lcss	Larger input problem
multiplier	Size of multiplication inputs
para	Larger input text
primetest	Unchanged
rewrite	Unchanged
scs	Reduce simulation timesteps
simple	Unchanged
sphere	Scene size

Figure 11. Differences between training and test data.

due to properties of the training input. Figure 11 summarizes these changes.

Figure 12 shows the performance measurements running on a machine with two 4-core CPUs. We measure the total wall-clock time spent outside garbage collection and normalize this to the time spent outside garbage collection in a single-threaded execution without any work sparking thunks. We omit garbage collection time because the GHC garbage collector is single-threaded. We plot the best-of-5 runs to avoid interaction with other services on the machine. For each program we plot the performance with the GHC run-time system configured to use 1..4 cores.

The results show that our actual implementation can still achieve a parallel speed-up, although at a much reduced level than the simulated performance in Figure 7. There are several possible explanations for the difference. First, the simulation ignores contention within the GHC run-time system – for example for access to the lock that protects the storage manager from which threads replenish their local allocation buffers. Second, the simulation ignores the overheads of sparking work and the cache effects of moving data from a ‘sparking’ core to one running work speculatively. Third, the simulation ignores the overheads of the shim-lock implementation. We have not yet had time to investigate the relative contributions of these factors.

In practice we would use the simulation results from Figure 7 to select whether or not FDIP is likely to benefit a given application. If it is not then we would run the application on an ordinary version of the GHC run-time system. In our example programs a simple comparison of the predicted 2-core performance against a 10% improvement threshold would identify the programs that could benefit from FDIP. However, for this paper, we show results for *all* of the programs to assess the costs introduced by FDIP when parallelism is not found.

We compared the number of bytes allocated under FDIP with the number of bytes allocated by the same program compiled without any calls to `sparkSpeculation`. On 1-core runs, calls to `sparkSpeculation` add less than 0.01% to the amount allocated by each program. On most programs they add less than 1% on 1-4 core runs. The worst case is 5% in `bsort-2` and `hidden`. This may be due to duplicate evaluation of thunks that are *not* sparked.

Finally, we examined one of the benchmarks to see how the allocation sites selected for sparking corresponded to pieces of the original program. We selected `bsort-2` because of our local knowledge in the algorithms it uses. The program is a sequential implementation of bitonic sorting networks (Batcher 1969) applied to 32 streams of integers. It models a pipelined hardware implementation in which each clock cycle inputs a new set of 32 integers and outputs a completed set of 32 integers.

The sorting network has a hierarchical structure, with the 32-way sorter being built from two 16-way sorters. This is expressed in the source code using a combinator `two`:

```
two :: ([a] -> [b]) -> [a] -> [b]
two r = halve >-> toBoth r r >-> unhalve
```

This combinator takes a function `r` from lists of `a` to lists of `b` and produces a function which splits its input list into two halves, applies `r` to each half, and then merges the results to form a list of `b`. The ‘`>->`’ combinator passes pairs of lists between these steps. FDIP identifies the two places where bsort-2 uses `two`, sparking one half of the work so that it can be computed in parallel with the other half.

In this case the decision of where to spark work corresponds closely to what would be done manually to parallelize the same program using the `par` and `seq` combinators. We have not yet had time to investigate the other programs to see whether or not this is true in general.

6. Related work

Hammond’s introduction to parallel functional programming introduces the main concepts and history (1994), highlighting the vital importance of introducing parallelism at the right granularity: too coarse and there will be idle processors, too fine and the overheads will be unacceptable.

Mechanisms to express speculative evaluation have been present in many functional programming languages. For example, Osborne (1990) explored the use of speculation in Multilisp, a language with explicit side effects and *futures* as the mechanism for expressing parallel tasks. Speculative tasks are explicitly created by the programmer. Explicit operations are also provided to control groups of tasks – for instance to cull speculative computation that is no longer needed. Our work aims to provide an entirely automatic mechanism for speculation.

Roe (1991) examined the use of the `par` and `seq` combinators to control parallelism. Our approach can be seen as an automated identification of where to use `par`; the placement in `bsort-2` we examined in Section 5 coincides with sensible manual placement. Of course, we anticipate that careful manual use of these combinators and evaluation strategies (Trinder et al. 1998) built over them will yield better performance than automated placement.

Mattson (1993) describes the use of speculative evaluation of Haskell programs on the BBN ‘butterfly’ multiprocessor machine, using the `par` combinator to identify non-speculative parallelism and additional priority annotations to identify possible speculative work. Mattson writes that ideally ‘the compiler employs heuristics to annotate the program graph automatically... heuristics to automatically determine safe and effective speculative annotations have not yet been developed’. Our work provides one heuristic for doing this, although we do not reify the annotations in the original source program.

Languages like Id90 and pH (Nikhil and Arvind 2001) combine non-strictness with eager evaluation. This ‘lenient’ evaluation (Traub 1991) provides abundant fine-grained parallelism. For example, given `let x=E in E’`, a lenient language will create separate tasks for `E` and `E’`, and similarly the body of a function and all of its arguments.

Arvind et al (1988) used an idealized interpreter to show that many traditional algorithms exhibit ample parallelism in this kind

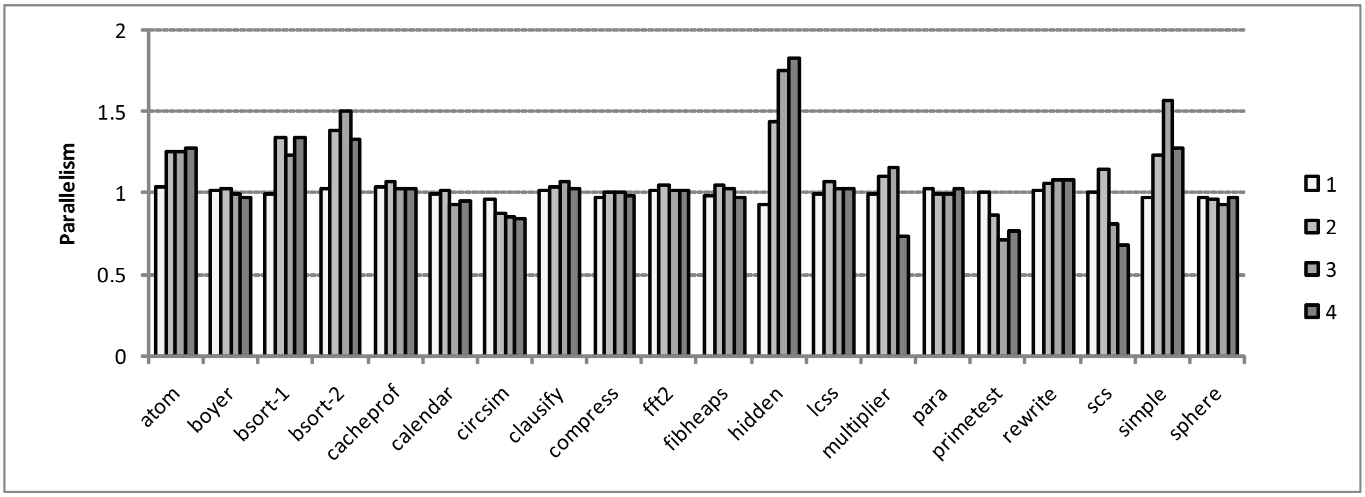


Figure 12. Actual performance on multi-core hardware, normalized against single-core performance with FDIP disabled and showing performance using 1, 2, 3, and 4 cores. As with previous graphs, 1 means 'the same as sequential execution', 2 means 'twice as fast as sequential execution', and so on.

of model and that it is effective both for using parallelism across processing elements and for masking large, unpredictable memory latencies. This form of parallelism is particularly effective on dataflow hardware such as Monsoon (Traub et al. 1991) which can support fine-grained concurrency. This is a much finer granularity of parallelism than our limit study identifies.

Ennals and Peyton Jones introduced the idea of *optimistic evaluation* for non-strict languages (Ennals and Peyton Jones 2003; Ennals 2004). The idea is to identify `let` expressions whose right-hand sides are quick to evaluate and likely to be needed. These can be optimistically evaluated without creating a thunk. The selection of where to use optimistic evaluation is based on dynamic feedback of which `let` expressions generate suitable thunks: optimistic evaluation is disabled for `let` expressions that generate long-running thunks or ones that are frequently un-needed.

Our decision to make spark / not-spark choices for each static allocation site was partly based on Ennals' and Peyton Jones' decision to make optimistic / lazy evaluation decisions for each `let` expression. It is interesting to consider whether dynamic feedback would be capable of making spark / not-spark decisions. It is not obvious that this is the case – the number of thunks sparked (Figure 7) is a tiny fraction of the total number created: probabilistic or burst-based profiling is likely to miss these, whereas profiling every thunk will have a high performance cost because many thunks are small (Figure 3).

GranSim (Loidl 1998) is a flexibile simulator for studying the performance of prorgams written in Glasgow Parallel Haskell. We developed a new tracing tool and simple simulator (Section 2) because we wished to study sequential benchmarks which had not yet been parallelised. It would be interesting to use GranSim to study the behavior of the parallel programs resulting from FDIP.

7. Conclusions and future work

In this paper we have shown that, for some programs, we can achieve 10-80% parallel speed-ups automatically on stock multi-core hardware. Furthermore, we can use a simulator to identify which programs are likely to have such speed-ups and avoid introducing any overheads in programs that will not. This clearly does not provide a silver bullet for using the full computational power of these machines effectively, but it suggests that perhaps parallelizing a single application thread across two cores is a practical proposition.

The key new technique in achieving this speed-up is the prediction of which thunk-allocation sites in the program are likely to produce long-running pieces of work based on profiling information collected from earlier program runs. A clear question for future work is whether we can adapt this technique to work within a single program run, rather than needing a profile-collection phase.

A further interesting direction to explore is the relationship between optimizations and available parallelism. As researchers have previously explored, there are several tensions here: optimizations for non-strict languages often try to avoid or delay thunk allocation, whereas we exploit thunk allocation as a potential source of parallelism. It would be interesting to explore the impact of reducing the level of optimization on the parallelism seen in our limit study and on the eventual performance that we achieve in practice. Perhaps it is practical to gather profiling data from an un-optimized program, identify profitable thunk-allocation sites, and then re-compile the program with some annotations to prevent these sites from being removed by optimizations. This would require tighter integration between the compiler and feedback-generation system than we are using at the moment.

Finally, although the goal of our work has been automatic parallelization, our techniques could also be applied to guide manual parallelization. As we showed with `bsort-2`, the selected thunk-allocation sites may correspond to meaningful points in the program source code, and so simply identifying these points from an execution trace could be a valuable programming aid.

Acknowledgments

We would like to thank John DeTreville for feedback and advice while starting this work, Maurice Herlihy, Jan-Willem Maessen, Simon Marlow and Simon Peyton Jones for discussions, the anonymous reviewers for their suggestions, and Andrew Birrell for help preparing the figures.

References

Arvind, David E. Culler, and Gino K. Maa. Assessing the benefits of fine-grained parallelism in dataflow programs. Technical Report 279, Computation Structures Group, MIT, March 1988.

K. E. Batcher. Sorting networks and their applications. In *Proceedings of the AFIPS Spring Joint Computing Conference*, 1969.

R. Blumofe and C. Leiserson. Scheduling multithreaded computations by work stealing. In *Proceedings of the 35th Annual Symposium on Foundations of Computer Science, Santa Fe, New Mexico.*, pages 356–368, November 1994.

Robert Ennals. *Adaptive Evaluation of Non-Strict Programs*. PhD thesis, Cambridge University Computer Laboratory, 2004.

Robert Ennals and Simon Peyton Jones. Optimistic evaluation: an adaptive evaluation strategy for non-strict programs. In *ICFP '03: Proceedings of the eighth ACM SIGPLAN international conference on Functional programming*, pages 287–298. ISBN 1-58113-756-7.

Kevin Hammond. Parallel functional programming: An introduction. In *International Symposium on Parallel Symbolic Computation*, Hagenberg/Linz, Austria, September 1994.

Tim Harris. Dynamic adaptive pre-tenuring. In *International Symposium on Memory Management (ISMM '00)*, volume 36(1) of *ACM SIGPLAN Notices*, pages 127–136, January 2001.

Tim Harris, Simon Marlow, and Simon Peyton Jones. Haskell on a shared-memory multiprocessor. In *Haskell '05: Proceedings of the 2005 ACM SIGPLAN workshop on Haskell*, pages 49–61, September 2005.

Danny Hendler, Yossi Lev, Mark Moir, and Nir Shavit. A dynamic-sized nonblocking work stealing deque. Technical Report TR-2005-144, Sun Microsystems Laboratories, 2005.

Raj Jain. *The art of computer systems performance analysis*. Wiley, 1991.

H-W. Loidl. *Granularity in Large-Scale Parallel Functional Programming*. PhD thesis, Department of Computing Science, University of Glasgow, March 1998.

James Stewart Mattson. *An effective speculative evaluation technique for parallel supercombinator graph reduction*. PhD thesis, University of California at San Diego.

Rishiyur S Nikhil and Arvind. *Implicit Parallel Programming in pH*. Morgan Kaufman, 2001.

Randy B. Osborne. Speculative computation in multilisp. In *LFP '90: Proceedings of the 1990 ACM conference on LISP and functional programming*, pages 198–208. ISBN 0-89791-368-X.

Simon Peyton Jones, Andrew Gordon, and Sigbjorn Finne. Concurrent haskell. In *POPL '96: Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 295–308, New York, NY, USA, 1996. ISBN 0-89791-769-3.

Paul Roe. *Parallel Programming Using Functional Languages*. PhD thesis, Department of Computing Science, University of Glasgow, 1991.

Kenneth R. Traub. *Implementation of non-strict functional programming languages*. MIT Press, Cambridge, MA, USA, 1991. ISBN 0-262-70042-5.

Kenneth R. Traub, Gregory M. Papadopoulos, Michael J. Beckerle, James E. Hicks, and Jonathan Young. Overview of the Monsoon project. In *ICCD '91: Proceedings of the 1991 IEEE International Conference on Computer Design on VLSI in Computer & Processors*, pages 150–155. ISBN 0-8186-2270-9.

P. W. Trinder, K. Hammond, H.-W. Loidl, and S. L. Peyton Jones. Algorithm + strategy = parallelism. *J. Funct. Program.*, 8(1):23–60, 1998. ISSN 0956-7968.