Optimistic Evaluation: an adaptive evaluation strategy for non-strict programs

Robert Ennals
Computer Laboratory, University of Cambridge
Robert.Ennals@cl.cam.ac.uk

Simon Peyton Jones
Microsoft Research Ltd, Cambridge
simonpj@microsoft.com

Abstract

Lazy programs are beautiful, but they are slow because they build many thunks. Simple measurements show that many of these thunks are unnecessary: they are in fact always evaluated, or are always cheap. In this paper we describe an evaluation strategy based on *optimistic evaluation*, that exploits this observation. Optimistic evaluation complements compile-time analyses with run-time experiments: it evaluates a thunk speculatively, but has an abortion mechanism to back out if it makes a bad choice. A run-time adaption mechanism records abortions so that the system can avoid making the same bad choice again.

We have implemented optimistic evaluation in the Glasgow Haskell Compiler. The results are encouraging: many programs speed up significantly (5-25%), some improve dramatically, while a very few go slightly slower.

1 Introduction

Lazy evaluation is great for programmers [14], but it carries significant run-time overheads. Instead of evaluating a function's argument before the call, lazy evaluation heap-allocates a *thunk*, or *suspension*, and passes that. If the function ever needs the value of that argument, it *forces* the thunk. This argument-passing mechanism is called *call-by-need*, in contrast to the more common call-by-value. The extra memory traffic caused by thunk creation and forcing makes lazy programs perform noticeably worse than their strict counterparts, both in time and space.

Good compilers for lazy languages therefore use sophisticated static analyses to turn call-by-need (slow, but sound) into call-by-value (fast, but dangerous). Two basic analyses are used: *strictness analysis* identifies expressions that will always be evaluated [39]; while *cheapness analysis* locates expressions that are certain to be cheap (and safe) to evaluate [6].

However, any static analysis must be conservative, and it turns out that plenty of thunks are still built, and (tantalisingly) *most of them are not necessary*. Figure 1 shows measurements taken from the Glasgow Haskell Compiler, a mature optimising compiler for Haskell. (GHC implements strictness analysis but not cheapness analysis, for reasons discussed in Section 5.1.) The Figure shows that most thunks are either always used or are "cheap and usually used". The exact definition of "cheap and usually-used" is not important: we use the this data only as *prima facie* evidence that there is a big opportunity here. If the language implementation could somehow use call-by-value for most of these thunks, then the overheads of laziness would be reduced signficantly.

This paper presents a realistic approach to *optimistic evaluation*, which does just that. Optimistic evaluation decides at run-time what should be evaluated eagerly, and provides an abortion mechanism that backs out of optimistic computations that go on for too long. We make the following contributions:

- The idea of optimistic evaluation *per se* is rather obvious (see Section 5). What is *not* obvious is how to (a) make it *cheap enough* that you get an overall win; (b) make it *comprehensive enough* that no program slows down markedly (it is much easier to get big speedups on some programs if you accept big slow-downs on others); (c) make it *robust enough* to run full-scale application programs without modification; and (d) do all this in a mature, optimising compiler that has already exploited most of the easy wins.

  In Section 2 we describe mechanisms that meet these goals. We have demonstrated that they work in practice by implementing them in the Glasgow Haskell Compiler (GHC) [28].
- Optimistic evaluation is a slippery topic. We give an *operational semantics* for a lazy evaluator enhanced with optimistic evaluation (Section 3). This abstract machine allows us to refine our general design choices into a precise form.

The performance numbers are extremely encouraging: we observe speedups in the range 5-50% with a geometric mean of 20% (Section 4). These are seriously good results for a compiler that is as mature as GHC, where 10% is now big news. (For example, strictness analysis buys 10-20% [30] — and our measurement baseline includes a strictness analyser that finds and elminates many of the unnecessary thunks.) While the baseline is mature, we have only begun to explore the design space for optimistic evaluation, so we are confident that our initial results can be further improved.

2 Optimistic Evaluation

Our approach to compiling lazy programs involves modifying only the back end of the compiler, and the run-time system. All existing analyses and transformations are first applied as usual, and the program is simplified into a form in which all thunk allocation is done by a `let` expression (Section 3.1). The back end is then modified as follows:

- Each `let` expression is compiled such that it can either build

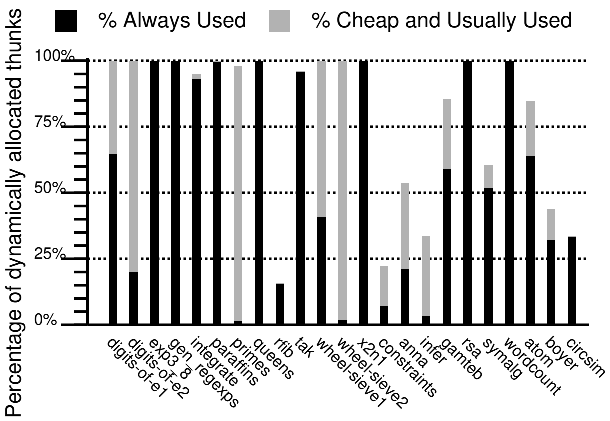


Figure 1. How thunks are commonly used

a thunk for its right hand side, or evaluate its right hand side immediately (Section 2.1). Which of these it does depends on the state of a runtime adjustable *switch*, one for each `let` expression. The state of all such switches is known as the *speculation configuration*. The configuration is initialised so that all `lets` are optimistic.

- If the evaluator finds itself optimistically evaluating a very expensive expression, then *abortion* is used to suspend the optimistic computation, building a special continuation thunk in the heap. Execution continues with the body of the optimistic `let` whose right-hand side has now been suspended 2.2).
- Recursively-generated structures such as infinite lists can be generated in *chunks*, thus reducing the costs of laziness, while still avoiding doing lots of unneeded work (Section 2.3).
- *Online profiling* is used to find out which thunks are expensive to evaluate, or are rarely used (Section 2.4). The profiler then modifies the speculation configuration, by switching off optimistic evaluation, so that the offending thunks evaluate their right hand sides lazily rather than strictly.
- When the program has finished running, the speculation configuration is saved to a file. The next time the program is run, this file is read in, restoring the configuration, thereby avoiding the "warm-up" period.

Just as a cache exploits the "principle" of locality, optimistic evaluation exploits the principle that most thunks are either cheap or will ultimately be evaluated. In practice, though, we have found that a realistic implementation requires quite an array of mechanisms — abortion, adaption, online profiling, support for chunkiness, and so on — to make optimistic evaluation really work, in addition to taking care of real-world details like errors, input/output, and so on. The following sub-sections introduce these mechanisms one at at time. To avoid continual asides, we tackle related work in Section 5.

2.1 Switchable Let Expressions

Consider the following `let` expression:

```
let x = <rhs> in <body>
```

Our compiler translates it to code that behaves logically like the following:

```
if (LET237 != 0) {
    x = value of <rhs>
}else{
    x = lazy thunk to compute <rhs> when needed
}
evaluate <body>
```

Our current implementation associates one static switch (in this case `LET237`) with each `let`. There are many other possible choices. For example, for a function like `map`, which is used in many different contexts, it might be desirable for the switch to take the calling context into account. We have not explored these context-dependent possibilities because the complexity costs of dynamic switches seem to overwhelm the uncertain benefits. In any case, GHC's aggressive inlining tends to reduce this particular problem.

In order to reduce code bloat, it is useful to limit the size of `<rhs>`. Our current implementation applies a simple flattening transformation, which makes the right hand side of every `let` be a function call.

2.2 Abortion

The whole point of optimistic evaluation is to use call-by-value in the hope that evaluation will terminate quickly. It is obviously essential to have a way to recover when this optimism turns out to be unfounded. That is the role of *abortion*.

If the evaluator detects that an optimistic evaluation, or speculation, has been going on for a long time, then it aborts all current speculations (they can of course be nested), resuming after the `let` that started the outermost speculation.

Detecting when a speculation has been running for too long can be done in several ways; the choice is not important, so long as it imposes minimal overheads on normal execution. One way would be to have the running code be aware of how long it has been running, and actively call into the runtime system when it has run for more than a fixed amount of time. Another approach (which we currently implement) is to have periodic sample points which look at the state of the running program. If execution remains within the same speculation for two sample points, then we consider the speculation to have gone on for too long.

Abortion itself can be implemented in many different ways. Our current implementation shares its implementation with that already used for handling asynchronous exceptions [19]. A suspension is created in the heap containing the aborted computation. If this computation is later needed, this computation continues from the point where it left off. Abortion turns out to be a very rare event, with only a few tens of abortions in each program run (see the data in Appendix A), so it does not need to be particularly efficient.

Abortion alone is enough to gurantee correctness; i.e. that the program will deliver the same results as its lazy counterpart. However, we can optionally also turn off the static switch for one or more of the aborted `lets`, so that they run lazily. (Currently we turn off the innermost `let` only, although many other strategies are possible.) From this we get a crude bound on wasted work. Since there are finitely many `lets` in a program, turning one off at each abortion bounds how many abortions can take place during the life of a program; in the limit, the entire program runs lazily. Further the wasted work done by each abortion is bounded by twice the sample interval. There is thus also a crude bound on the total time and space consumed by aborted speculations.

2.3 Chunky Evaluation

Consider the following program, which generates an infinite stream of integers.

```
from :: Int -> [Int]
```

```
from n = let rest = from (n+1)
         in (n : rest)
```

If we used optimistic evaluation for the rest thunk, the evaluation would be certain to abort. Why? Because evaluating from n would optimistically evaluate from (n-1), which would optimistically evaluate from (n-2), and so on. It will not be long before the adaption mechanism turns off optimistic evaluation of rest, which will make from completely lazy.

This is a big lost opportunity: when lazily evaluating the lazily generated list, the tail of every cell returned by from is sure to be evaluated, except the last one! (The heads are easily caught by optimistic evaluation, if n is cheap.)

What we would really like instead is to create the list in *chunks* of several elements. While creating a chunk, rest is evaluated speculatively. However once the chunk is finished (for example, after 10 elements have been created) rest switches over to lazy evaluation, causing the function to terminate quickly. This approach largely removes the overheads of laziness, because only a few thunks are lazy — but it still allows one to work with infinite data structures.

This chunky behavior can be useful even for finite lists that are completely evaluated. Lazy programmers often use a generate-and-filter paradigm, relying on laziness to avoid creating a very large intermediate list. Even if the compiler knew that the intermediate list would be completely evaluated, it would sometimes be a bad plan to evaluate it strictly. Chunky evaluation is much better.

One way to implement chunky evaluation is to limit how deeply our speculation is nested. The code for a let then behaves semantically like the following (note that this generalises the code given is Section 2.1):

```
if (LET237 > 0){
    LET237 = LET237 - 1
    x = value of <rhs>
    LET237 = LET237 + 1
}else{
    x = lazy thunk for <rhs>
}
evaluate <body>
```

Here, LET237 can be adjusted at run-time to control the maximum number of nested speculations of this particular let. To make the let lazy, set LET237 to 0. Another alternative is to have a global speculation-depth register, and compare that with a limit stored in LET237. This would cut off optimistic evaluation rather earlier; the intuition is that the deeper the speculation stack, the less likely a new speculation is to be useful.

2.4 Online Profiling

Abortion may guarantee *correctness* (Section 2.2), but it does not guarantee *efficiency*. It turns out that with the mechanisms described so far, most programs run faster under optimistic evaluation, but a few run dramatically slower (see the data for constraints in Section 4.4). Why? Detailed investigation shows that these programs build many moderately-expensive thunks that are seldom used. These thunks are too cheap to trigger abortion, but nevertheless aggregate to waste massive amounts of of time and space.

One obvious solution is this: trigger abortion very quickly after starting an optimistic evaluation, thereby limiting wasted work. However, this approach fails to exploit the thunks that are not dirt-cheap, but are almost always used; nor would it support chunky evaluation.

Our approach to this problem is to use an online profiler. As the program is running, the profiler observes two properties of selected thunks: (a) how expensive its evaluation is, and (b) whether its result is eventually needed. We use that information to set its speculation configuration: we switch off optimistic evaluation, or at least reduce its depth limit (Section 2.3), if the thunk is (on average) not cheap, or not usually used.

To determine (a) — that is, how expensive the right hand side is to evaluate — we count the number of heap allocations that take place during its evaluation. Heap allocations are used as a rough indicator of work done. This can be made sound by ensuring that all recursive calls allocate heap. The advantage of using heap allocation is that it allows extremely fine-grain timing. This is important: a speculation doesn't have to be very big before it is very expensive compared to the cost of creating a lazy thunk.

To determine (b) — that is, how likely the right hand side of a let is to be needed — we wrap the value produced in a special kind of indirection thunk [27]. This thunk informs the runtime system when it is evaluated, allowing statistics to be gathered on how likely such thunks are to be used. We have to be careful with values that are used by another speculation, because the result of that speculation might itself not be used.

Profiling every thunk all of the time would be unreasonably expensive. Instead, profiling is turned on selectively, on a thunk-by-thunk basis. In concrete terms, *we profile any thunk that is under optimistic evaluation at a sample point*. This simple technique automatically focuses profiling on the most heavily-used thunks. Furthermore, by adjusting the sampling intervals we can trade off overhead against precision, while still having a statistical guarantee that all wasteful let expressions will eventually be found. Similar techniques have been described previously by Arnold and Ryder [1].

We have skated somewhat lightly over the details of the online-profiling implementation, for lack of space, but the exact technique chosen does not matter. The main points are: that it is important to switch off even moderately-cheap thunks if they are seldom used; and that online profiling techniques can be used to do this without imposing heavy run-time penalties. Section 4.4 gives data to support both these claims.

2.5 Exceptions and Errors

Consider the following function:

```
f x y = let { bad = error "urk" }
        in if x then bad else y
```

In Haskell, the error function prints an error message and halts the program. Optimistic evaluation may evaluate bad without knowing whether bad is actually needed. It is obviously unacceptable to print "urk" and halt the program, because lazy evaluation would not do that if x is False. The same issue applies to exceptions of all kinds, including divide-by-zero and black-hole detection [24].

In GHC, error raises a catchable exception, rather than halting the program [31]. The exception-dispatch mechanism tears frames off the stack in the conventional way. The only change needed is to modify this existing dispatch mechanism to recognise an optimistic-evaluation let frame, and bind its variable to a thunk that re-raises the exception. A let thus behaves rather like a catch statement, preventing exceptions raised by speculative execution from escaping.

2.6 Unsafe input/output

Optimistic evaluation is only safe because Haskell is a *pure* language: evaluation has no side effects. Input/output is safely partitioned using the IO monad [32], so there is no danger of

(VAR1)	$\Gamma[x \mapsto V]; x; s; \Sigma$	$\Rightarrow$	$\Gamma[x \mapsto V]; V; s; \Sigma$	
(VAR2)	$\Gamma[x \mapsto E]; x; s; \Sigma$	$\Rightarrow$	$\Gamma; E; (\mathbf{update}\, x) : s; \Sigma$	if E is not a value
(UPDATE)	$\Gamma; V; (\mathbf{update}\, x) : s; \Sigma$	$\Rightarrow$	$\Gamma[x \mapsto V]; V; s; \Sigma$	
(APP1)	$\Gamma; E\, x; s; \Sigma$	$\Rightarrow$	$\Gamma; E; (\bullet\, x) : s; \Sigma$	
(APP2)	$\Gamma; \lambda x.E; (\bullet\, x') : s; \Sigma$	$\Rightarrow$	$\Gamma; E[x'/x]; s; \Sigma$	
(APP3)	$\Gamma; \mathbf{exn}\,; (\bullet\, x') : s; \Sigma$	$\Rightarrow$	$\Gamma; \mathbf{exn}\,; s; \Sigma$	
(CASE1)	$\Gamma;\ \mathbf{case}\ E\ \mathbf{of}\ P_0 \mid \ldots \mid P_n; s; \Sigma$	$\Rightarrow$	$\Gamma; E;\ \mathbf{case}\ \bullet\ \mathbf{of}\ P_0 \mid \ldots \mid P_n : s; \Sigma$	
(CASE2)	$\Gamma; C\, x'_0 \ldots x'_n;\ \mathbf{case}\ \bullet\ \mathbf{of}\ P_0 \mid \ldots \mid P_n : s; \Sigma$	$\Rightarrow$	$\Gamma; E[x'_0/x_0, \ldots, x'_n/x_n]; s; \Sigma$	where $P_i = C\, x_0 \ldots x_n \rightarrow E$
(CASE3)	$\Gamma; \mathbf{exn}\,;\ \mathbf{case}\ \bullet\ \mathbf{of}\ P_0 \mid \ldots \mid P_n : s; \Sigma$	$\Rightarrow$	$\Gamma; \mathbf{exn}\,; s; \Sigma$	
(LAZY)	$\Gamma;\ \mathbf{let_a}\ x = E_1\ \mathbf{in}\ E_2; s; \Sigma[a:0]$	$\Rightarrow$	$\Gamma[x' \rightarrow E_1]; E_2[x'/x]; s; \Sigma$	where x' is new
(OPT1)	$\Gamma;\ \mathbf{let_a}\ x = E_1\ \mathbf{in}\ E_2; s; \Sigma[a:n]$	$\Rightarrow$	$\Gamma; E_1;\ \mathbf{let_a}\ x = \bullet\ \mathbf{in}\ E_2 : s; \Sigma[a:n-1]$	
(OPT2)	$\Gamma; V;\ \mathbf{let_a}\ x = \bullet\ \mathbf{in}\ E : s; \Sigma[a:n]$	$\Rightarrow$	$\Gamma[x' \rightarrow V]; E[x'/x]; s; \Sigma[a:n+1]$	where x' is new

Figure 2. Operational Semantics: Evaluation

optimistic computations performing I/O. However, Haskell programs sometimes make use of impure I/O, using the "function" `unsafePerformIO`. Optimistically evaluating calls to this function could cause observable behavior different to that of a lazy implementation.

Our solution is brutal: `unsafePerformIO` checks whether the evaluator is evaluating optimistically, and immediately aborts if so, disabling the youngest optimistic `let`. In this way, no effects inside `unsafePerformIO` will be performed until the "real" demand reaches it. Sometimes, however this is too brutal and prevents optimistic evaluation in cases where it would be harmless. We are currently experimenting with the possibility of splitting `unsafePerformIO` into two functions, one of which can be executed optimistically, and one of which must not.

2.7 Summary

By this time, the reader may be thinking "isn't this all rather complicated?". Does one really need abortion, adaption, chunky evaluation, *and* online profiling? Our answer is two-fold.

First, just as a compiler needs a lot of "bullets in its gun", to tackle widely varying programs, we believe that the same is true of optimistic evaluation. To make idea work, in practice, on real programs, in a mature compiler, there just are a lot of cases to cover. One of the contributions of this paper is precisely that we have explored the idea deeply enough to expose these cases. Section 4.4 quantifies the effect of removing individual "bullets".

Second, the actual implementation in GHC is not overly complex. The changes to the compiler itself are slight (a few hundred lines at most, in 70,000 or so). The changes to the run-time system are around 1,000 lines of C in a run-time system of around 30,000 lines — certainly not trivial, but a 3% increase seems reasonable for a major innovation.

3 An Operational Semantics

As we have remarked, optimistic evaluation is subtle. We found it extremely helpful to write a formal operational semantics to explain exactly what happens during optimistic evaluation, abortion, and adaption. In this section we briefly present this semantics.

3.1 A simple language

The language we work with is Haskell. While the externally visible Haskell language is very complex, our compiler reduces it to the following simple language:

E	::=	x	variable
	$\mid$	$C\, x_0 \ldots x_n$	constructor application
	$\mid$	$\mathbf{case}\ E\ \mathbf{of}\ P_0 \mid \ldots \mid P_n$	case analysis
	$\mid$	$\mathbf{let_a}\ x = E\ \mathbf{in}\ E'$	thunk creation
	$\mid$	$\lambda x.E$	abstraction
	$\mid$	$E\, x$	application
	$\mid$	$\mathbf{exn}$	exception or error
P	::=	$C\, \{x_0, \ldots, x_n\} \rightarrow E$	pattern match
V	::=	$C\, x_0 \ldots x_n \mid \lambda x.E \mid \mathbf{exn}$	values

Each occurrence of `let` is given a unique label a that can be used to refer to specific `let` constructs in a program. These labels identify the static optimism switches introduced in Section 2.1. Functions and constructors are always applied to variables, rather than expressions; that is, the program is assumed to be in A-normal form. It follows that `let` is the only point at which a thunk might be created. We omit literals and primitive operators for the sake of brevity. Adding them introduces no extra complications.

Exceptions and errors (Section 2.5) are handed exactly as described in [31], namely by treating an exception as a *value* (and not as a control operator).

3.2 The Operational Framework

We describe program execution using a small step operational semantics, describing how the program state changes as execution proceeds. The main transition relation, $\Rightarrow$, takes the form:

$$\Gamma; E; s; \Sigma \ \Rightarrow\ \Gamma'; E'; s'; \Sigma'$$

meaning that the state $\Gamma; E; s; \Sigma$ evolves to $\Gamma'; E'; s'; \Sigma'$ in one step. The components of the state are as follows.

- Γ represents the heap. It is a function mapping names to expressions.

$$
\begin{array}{rrcll}
(!LET) & \Gamma; E_1;\ \mathbf{let_a}\ x = \bullet\ \mathbf{in}\ E_2 : s; \Sigma & \rightsquigarrow & \Gamma[x' \to E_1]; E_2[x'/x]; s; \Sigma & \text{where } x' \text{ is new} \\
(!CASE) & \Gamma; E_1;\ \mathbf{case}\ \bullet\ \mathbf{of}\ P_0 | \ldots | P_n : s; \Sigma & \rightsquigarrow & \Gamma;\ \mathbf{case}\ E_1\ \mathbf{of}\ P_0 | \ldots | P_n; s; \Sigma & \\
(!APP) & \Gamma; E; (\bullet\ x) : s; \Sigma & \rightsquigarrow & \Gamma; E\ x; s; \Sigma & \\
(!UPDATE) & \Gamma; E; (\mathbf{update}\ x) : s; \Sigma & \rightsquigarrow & \Gamma[x \mapsto E]; x; s; \Sigma &
\end{array}
$$

Figure 3. Operational Semantics : Abortion

- E is the expression currently being evaluated.
- s is a stack of *continuations*, each taking one of the following forms (c.f. [9]):

$\mathbf{let_a}\ x = \bullet\ \mathbf{in}\ E$	Let
$\mathbf{case}\ \bullet\ \mathbf{of}\ P_0 \| \ldots \| P_n$	Case
$\bullet\ x$	Application
$\mathbf{update}\ x$	Thunk update

  Each of these represents work still to be done, with $\bullet$ being the place we will put the result of the current evaluation when we return into the continuation.
- Σ maps a `let`-expression label a to an integer n. This integer says how deeply this `let` can be speculated before it should become lazy. If the `let` is lazy, then this integer will be 0. Run-time adaption may change the mapping Σ to choose a better strategy.

The transition rules for $\Rightarrow$ are given in Figure 2. The first nine are absolutely conventional [34], while the last three are the interesting ones for optimistic evaluation. Rule (LAZY) is used if the speculation configuration Σ indicates that this particular `let` expression should be treated lazily ($\Sigma[a:0]$). The rule simply builds a thunk in the heap Γ, with address x', and binds x to x'. Rule (OPT1) is used if $\Sigma[a]$ is greater than zero; it decrements $\Sigma[a]$ (as discussed in Section 2.3) and begins evaluation of the right-hand side. Finally, (OPT2) increments $\Sigma[a]$ and binds x to (the address of) the computed value.

At periodic *sample points*, the *run-time strategy manager* runs. It can do two things: first, it can change speculation configuration Σ; and second, it can abort one or more running speculations. The process of abortion is described by the $\rightsquigarrow$ transitions given in Figure 3. Each abortion rule removes one continuation from the stack, undoing the rule that put it there. It would be sound to keep applying these rules until the stack is empty, but there is no point in continuing when the last optimistic-let continuation ($\mathbf{let_a}\ x = \bullet\ \mathbf{in}\ E$) has been removed from the stack, and indeed our implementation stops at that point.

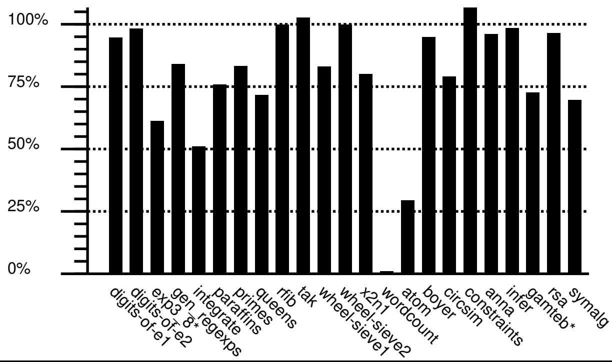
Like a cache-management policy, the strategy manager operates in a large design space of heuristics that we have only begun to explore. The encouraging thing is that simple heuristics give very promising results, as we discuss next.

4 Performance

We measured the effect of optimistic evaluation on several programs. All but one of the programs are taken from the NoFib [26] suite. The other is `wordcount` which we refer to in Section 4.2.

We ran each program with our modified compiler, and with the GHC compiler that our implementation forked from (version 5.02.2), on a 750MHz Pentium III with 256Mbytes of memory. Each benchmark was run once to provide a "warm up" for the adaptive strategy manager, and then run five more times on the same input data. (We believe that the performance improvement would be nearly as good if we had used different input data after the warm-up, but we do not yet have figures to substantiate this claim.) The detailed figures, including code size, heap allocation, and warm-up performance, are given in Appendix A.

Figure 4. Benchmark Run-Time relative to Normal GHC



4.1 Execution time

Figure 4 shows the changes in execution time, normalised to normal GHC (using full optimization). Most programs improve, in some cases dramatically, with a (geometric) mean of 20%. This is very encouraging: GHC is a mature compiler, and we regard a 10% improvement as big news. As one would expect, the results depend dramatically on the nature of the program. For example, `rfib` and `wheel-sieve2` both have inner loops that the strictness analyser solves, so there is no laziness left for optimistic evaluation to improve; and `rsa` spends most of its time in the GMP arbitrary-precision arithmetic package. At the other end of the spectrum, optimistic evaluation can give truly dramatic improvements (for example, `atom` goes nearly four times faster).

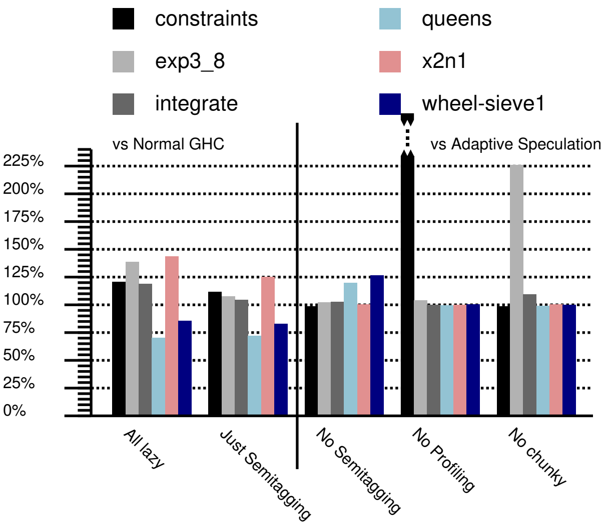
A few programs run slightly slower, notably `constraints`. These are usually programs make essential use of laziness in their inner loops. Such programs suffer from the fact the dynamic test at each `let` (Section 2.1) makes laziness more expensive than in the baseline implementation, even when the speculation configuration has made the correct `let`s lazy.

4.2 Heap residency

Some programs are extremely inefficient when executed lazily, because they contain a space leak. People often post such programs on the `haskell` mailing list, asking why they are going slowly. One recent example was a simple word counting program [22]. The inner loop (slightly simplified) was the following:

```
count :: [Char] -> Int -> Int -> Int -> (Int,Int)
count []        _ nw nc = (nw, nc)
count (c:cs) new nw nc =
```

Figure 5. Runtimes of Simplified Implementations




```
case charKind c of
  Normal   -> count cs 0 (nw+new)  (nc+1)
  White    -> count cs 1 nw        (nc+1)
```

Every time this loop sees a character, it increments its accumulating parameter nc. Under lazy evaluation, a long chain of addition thunks builds up, with length proportional to the size of the input file. By contrast, the optimistic version evaluates the addition speculatively, so the program runs in constant space.

Optimistic evaluation speeds this program up so much that we were unable to produce an input file that was both small enough to allow the lazy implementation to terminate in reasonable time, and large enough to allow the optimistic implementation to run long enough to be accurately timed! This may be a toy program, but it demonstrates a common problem that regularly afflicts beginners. Optimistic evaluation does not cure all space leaks, of course, but it cures some egregious ones, and that is a welcome improvement.

Appendix A shows the effect of optimistic evaluation on heap residency (i.e. maximum heap size) for each test program. The effect is variable; some programs improve dramatically, but others get significantly worse. However, the worst offenders (queens, x2n1) have a tiny residency anyway. Of the programs with significant residency, the geometric mean change in residency shows an increase of 4% with a worst case of 40%.

4.3 Code Size

Appendix A shows that code size increases significantly, by some 70%. We have not yet paid much attention to code bloat, and are confident that it can be reduced substantially, but we have yet to demonstrate this.

4.4 Where the performance comes from

Where does the performance improvement come from? Could we get the same performance results from a simpler system?

Figure 5 shows the performance of several simplified variants of our system. For clarity, we show only six test programs, selected to give a truthful picture of the behaviors we have observed.

Overheads. The first group, "All lazy", shows what happens when we pay all the costs of optimistic evaluation, but get none of the benefits. In this group, we set Σ to map every let to 0, so that all lets are done lazily. Comparing this to Normal GHC, the baseline for this group, shows the overheads that the strategy manager, and the switchable-let mechanism, impose on normal evaluation. The overheads vary, but are always less than 40%. The strange thing is that two programs actually go *faster*. How can that be?

Semi-tagging. Our implementation of optimistic evaluation uses an optimisatation called semi-tagging, which we briefly explain. When evaluating a case expression, GHC normally jumps to the entry point of the scrutinee, passing it a vector of return addresses, one for each possible constructor. If the scrutinee is a value, then the entry point will simply return to the relevant return address. An alternative plan is called *semi-tagging* [29]: before jumping to the scrutinee's entry point, test whether the scrutinee is already evaluated. In that case, we can avoid the (slow) indirect call and return.

The second group of results, "Just semitagging", in Figure 5 shows the effect enabling semi-tagging only (but not optimistic evaluation) relative to the baseline GHC. Under normal lazy evaluation, the scrutinee is often unevaluated, and semi-tagging has mixed results: some programs go 25% faster, but some go 25% slower. (That is why semi-tagging is not enabled in the baseline GHC.) It is the benefits of semi-tagging that exactly account for the programs in the first group that go faster.

However, *under optimistic evaluation most scrutinees are evaluated*, and semi-tagging gives a consistent improvement. The "No semi-tagging" group of results are given relative to the full optimistic evaluator, and show that switching off semi-tagging always makes an optimistically-evaluated program run slower. In short, optimistic evaluation turns semi-tagging from a mixed blessing into a consistent, and sometimes substantial, win. This is a real bonus, and one we did not originally anticipate.

Chunky evaluation and online profiling. The last two groups show the effect of switching off chunky evaluation and online profiling. In each case, a few programs run much more slowly; indeed constraints runs so slowly without profiling that we had to abandon the run. These results convinced us of the need for online profiling, and for support for chunky evaluation.

5 Related Work

5.1 Static analyses

Where static analysis is possible, it is much to be preferred, because the results of the analysis can often enable a cascade of further transformations and optimisations, and static choices can be compiled into straight line code, with better register allocation.

GHC has a fairly sophisticated strictness analyser, and all our results are relative to a baseline in which strictness analysis is on. (If we switched it off, the speedups from optimistic evaluation would be much greater.) The other promising static analysis is Faxén's *cheap eagerness analysis* [6], which attempts to figure out which thunks are guaranteed to be cheap to evaluate, so that call-by-value is sound, and will not waste much work even if the result is not used. A further development, dynamic cheap eagerness, uses a more complicated analysis to add an extra depth parameter to selected recursive functions, plus an explicit cut-off test, to achieve an effect similar to chunky evaluation.

Cheap eagerness is built on a very sophisticated *whole-program* flow analysis. Is a thunk for x+1 cheap? It depends on whether x is evaluated; and if x is an argument to a function, we need to examine all calls to the function — and that is not straightforward in a higher-order program. Worse, a whole-program analysis causes problems for separate compilation, and that is a big problem when shipping pre-compiled libraries. These problems may be soluble — for example by compiling multiple clones of each function, each suitable for a different evaluation pattern — but the additional implementation complexity would be significant. That is why we did not implement cheap eagerness in GHC for comparison purposes.

A sufficiently-clever whole-program static analysis might well discover many of the cheap thunks that we find with our runtime profiler. The critical difference is that we can find all the cheap thunks without being particularly clever, without requiring a particularly complex implementation, and without getting in the way of separate compilation.

Faxén reports some promising speedups, generally in the range 0-25% relative to his baseline compiler, but it is not appropriate to compare these figures directly with ours. As Faxén is careful to point out, (a) his baseline compiler is a prototype, (b) his strictness analyser is "very simple", and (c) all his benchmarks are small. The improvements from cheapness analysis may turn out to be less persuasive if a more sophisticated strictness analyser and program optimiser were used, which is our baseline. (Strictness analysis does not require a whole-program flow analysis, and readily adapts to separate compilation.)

5.2 Eager Haskell

Eager Haskell [18, 17] was developed simultaneously, but independently, from our work. Its basic premise is identical: use eager evaluation by default, together with an abortion mechanism to back out when eagerness turns out to be over-optimistic. The implementation is rather different, however. Eager Haskell evaluates absolutely everything eagerly, and periodically aborts the running computation right back to the root. Abortion must not be too frequent (lest its costs dominate) nor too infrequent (lest work be wasted). The abortion mechanism is also different: it allows the computation to proceed, except each function call builds a thunk instead of making the call. The net effect is somewhat similar to our chunky evaluation, but appears to require an "entirely new" code generator, which ours does not.

The main advantage of our work over Eager Haskell is that we *adaptively* decide which `let` expressions are appropriate to evaluate eagerly while Eager Haskell always evaluates everything eagerly. On some programs eager evaluation is a good plan, and the Eager Haskell gets similar speedups to Optimistic Evaluation. On other programs, though, laziness plays an important role, and Eager Haskell can slow down (relative to normal GHC) by a factor of 2 or more. In an extreme case (the `constraints` program) Eager Haskell goes over 100 times slower than normal GHC, while with Optimistic Evaluation it only slows down by around 6%. Eager Haskell users are encouraged to deal with such problems by annotating their programs with laziness annotations, which Optimistic Evaluation does not require. It is much easier to get good speedups in some case by accepting big slow-downs in others. It is hard to tell how much Eager Haskell's wins will be reduced if it were to solve the big-slow-down problem in an automated way.

5.3 Speculative parallelism

The parallel programming community has been making use of speculation to exploit parallel processors for a long time [4]. There, the aim is to make use of spare processors by arranging for them to evaluate expressions that are not (yet) known to be needed. There is a large amount of work in this field, of which we can only cite a small subset.

Several variants of MultiLisp allow a programmer to suggest that an expression be evaluated speculatively [10, 25]. Mattson [20, 21] speculatively evaluates lazy expressions in parallel. *Local Speculation* [21, 5] does some speculations on the local processor when it would otherwise be waiting, reducing the minimum size for a useful speculation. Huntback [15] speculatively evaluates logic programming expressions in parallel, with the user able to annotate speculations with a priority. A major preoccupation for speculative parallelism is achieving large enough granularity; otherwise the potential gain from parallelism is canceled out by the cost of spawning and synchronisation. In our setting, the exact reverse holds. A large thunk might as well be done lazily, because the cost of allocating and updating it are swamped by its evaluation costs; it is the *cheap* thunks that we want to speculate!

Haynes and Friedman describe an explicit "engines" process abstraction that can be used by the programmer to spawn a *resource-limited* thread [12]. An engine has a certain amount of fuel; if the fuel runs out, the engine returns a continuation engine that can be given more fuel, and so on. Engines are rather coarse-grain, and under explicit user control, both big differences from our work.

Another strand of work takes eager parallelism as the baseline, and strives to aggregate, or *partition*, tiny threads into larger compound threads [37, 33]. In some ways this is closer to our work: call-by-need is like parallel evaluation (scheduled on a uniprocessor), while using call-by-value instead aggregates the lazy thread into the parent. However, the issues are quite different; as Schauser puts it "the difficulty is not what *can* be put in the same thread, but what *should* be ... given communication and load-balancing constraints". Furthermore, thread partitioning is static, whereas our approach is dynamic.

Yet another strand is that of *lazy thread creation*, where one strives to make thread creation almost free in the case where it is not, in the end, required [23]. A very successful variant of this idea is Cilk [8], in which the parent thread saves a continuation before optimistically executing the spawned child; if the child blocks, the processor can resume at the saved continuation. Other processors can also steal the saved continuation. The big difference from our work is that in Cilk all threads are assumed to be required, whereas the possibility that a lazy thunk is not needed is the crux of the matter for us.

In summary, despite the common theme of speculative evaluation in a declarative setting, we have found little overlap between the concerns of speculation-for-parallelism and optimistic evaluation.

5.4 Other work

Stingy Evaluation [38] is an evaluation strategy designed to reduce space leaks such as the one described in Section 4.2. When evaluating a `let` expression, the evaluator does a *little* bit of work on the expression, with the hope of evaluating it, and avoiding having to build a thunk. As with Eager Haskell, all expressions are eagerly evaluated, however the amount of evaluation done before abortion is significantly smaller, with only very simple evaluations allowed. Often this small amount of work will not be useful, causing some programs to run slower. No compiler known to us uses stingy evaluation.

Branch Prediction [35] is present in most modern processors. The processor will speculatively execute whichever side of a branch that is thought most likely to be used. As with optimistic evaluation, observation of the running program is used to guide speculation. Static analysis can be used to guide branch prediction [2].

Runtime Profiling is used in many existing language implementations, including several implementations for the Java [16] language, such as HotSpot [36] and Jalapeno [3]. One of the first implementations to use such techniques was for Self [13]. These systems use similar techniques to optimistic evaluation, but do not apply them to laziness.

Feedback Directed Optimization [7] is a widely used technique in static compilers. A program is run in a special profiling mode, recording statistics about the behavior of the program. These statistics are then used by the compiler to make good optimization choices when compiling a final version of the program. Many commercial compilers use this technique. In principle we could do the same, compiling the configuration Σ into the program, rather than making it adapt at run-time, but we have not done so.

6 Conclusions and Future Work

Our goal is to allow a programmer to write programs in a lazy language without having to use strictness annotations in order to obtain good performance. Our measurements show that optimistic evaluation can improve overall performance by 20%, or more, relative to a very tough baseline.

Our implementation represents just one point in a large design space that we are only just beginning to explore. It is very encouraging that we get good results so early; things can only get better! In particular, we are looking at the following:

Variable Frequency Sampling. Inspecting the running state at a fixed frequency is a very crude approach. It might be better to sample frequently at the beginning of the run, and reduce the sample rate later. A further refinement is to place a trap on the first call to any function. Such a trap indicates that new – and thus probably poorly adjusted — code is being executed, so increasing the sampling frequency might be a good plan.

Heap Usage Profiling. As shown in Section 4.2, optimistic evaluation often speeds programs up by preventing them from filling the heap up with thunks. However our current system does not take heap behavior into account when deciding which `let` expressions should be evaluated optimistically, and so could be missing opportunities to speed programs up. We plan to explore ways of making better decisions, by taking into account the behavior of the heap.

Proving Worst Case Performance. We have stressed the importance of avoiding bad performance for particular programs. But experiments can never show that *no* program will run much slower under our system than with normal GHC. Instead we would like to *prove* this property. Based on the operational semantics in Section 3, preliminary work suggests that we can indeed prove that the online profiler will eventually find all `let` expressions that waste work, and hence that that optimistic evaluation is at most a constant factor worse than ordinary call-by-need (because of its fixed overheads), once the speculation configuration has converged [Paper in preparation].

There are hundreds of papers about cunning compilation techniques for call-by-value, and hundreds more for call-by-need. So far as we know, this paper and Maessen's recent independent work [17] are the first to explore the rich territory of choosing dynamically between these two extremes.

Acknowledgments

We are grateful for the generous support of Microsoft Research, who funded the studentship of the first author. We would also like to thank Manuel Chakravarty, Fergus Henderson, Jan-Willem Maessen, Simon Marlow, Greg Morrisett, Alan Mycroft, Nick Nethercote, Norman Ramsey, and Richard Sharp, who provided useful suggestions.

7 References

[1] M. Arnold and B. G. Ryder. A framework for reducing the cost of instrumented code. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*, 2001.

[2] T. Ball and J. R. Larus. Branch prediction for free. In *ACM Conference on Programming Languages Design and Implementation (PLDI'93)*, pages 300–313. ACM, June 1993.

[3] M. G. Burke, J.-D. Choi, S. Fink, D. Grove, M. Hind, V. Sarkar, M. J. Serrano, V. Sreedhar, H. Srinivasan, and J. Whaley. The Jalapeno dynamic optimizing compiler for Java. In *Proceedings of the ACM Java Grande Conference*, 1999.

[4] F. W. Burton. Speculative computation, parallelism and functional programming. *IEEE Trans Computers*, C-34(12):1190–1193, Dec. 1985.

[5] M. M. T. Chakravarty. Lazy thread and task creation in parallel graph reduction. In *International Workshop on Implementing Functional Languages*, Lecture Notes in Computer Science. Springer Verlag, 1998.

[6] K.-F. Faxén. Cheap eagerness: Speculative evaluation in a lazy functional language. In *ACM SIGPLAN International Conference on Functional Programming (ICFP'00)*, Montreal, Sept. 2000. ACM.

[7] *ACM Workshop on Feedback Directed and Dynamic Optimization (FDDO)*, 2001.

[8] M. Frigo, C. E. Leiserson, and K. H. Randall. The implementation of the Cilk-5 multithreaded language. In *ACM Conference on Programming Languages Design and Implementation (PLDI'98)*, volume 33, pages 212–223, Atlanta, May 1998. ACM.

[9] J. Gustavsson. A type-based sharing analysis for update avoidance and optimisation. In *ACM SIGPLAN International Conference on Functional Programming (ICFP'98)*, volume 34(1) of *ACM SIGPLAN Notices*, Baltimore, 1998. ACM.

[10] R. H. Halstead. Multilisp - a language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems*, 7(4):501–538, Oct. 1985.

[11] K. Hammond and J. O'Donnell, editors. *Functional Programming, Glasgow 1993*, Workshops in Computing. Springer Verlag, 1993.

[12] C. Haynes and D. Friedman. Engines build process abstractions. In *Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming*, 1984.

[13] U. Hölzle. *Adaptive optimization for Self: reconciling high performance with exploratory programming*. Ph.D. thesis, Computer Science Department, Stanford University, Mar. 1995.

[14] J. Hughes. Why functional programming matters. *The Computer Journal*, 32(2):98–107, Apr. 1989.

[15] M. Huntback. Speculative computation and priorities in concurrent logic languages. In *ALPUK Conference*, 1991.

[16] H. M. J Gosling. The Java Language Environment: a White Paper. Technical report, Sun Microsystems, 1996.

[17] J.-W. Maessen. Eager Haskell: Resource-bounded execution yields efficient iteration. In *The Haskell Workshop, Pitts-*

burgh, 2002.

[18] J.-W. Maessen. *Hybrid Eager and Lazy Evaluation for Efficient Compilation of Haskell*. PhD thesis, Massachusetts Institute of Technology, June 2002.

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

[20] J. Mattson. *An effective speculative evaluation technique for parallel supercombinator graph reduction*. Ph.D. thesis, Department of Computer Science and Engineering, University of California, San Diego, Feb. 1993.

[21] J. S. Mattson Jr and W. G. Griswold. Local Speculative Evaluation for Distributed Graph Reduction. In Hammond and O'Donnell [11], pages 185–192.

[22] L. Maurer. Isn't this tail recursive? Message posted to the Haskell mailing list: http://haskell.org/pipermail/haskell/2002-March/009126.html, Mar. 2002.

[23] E. Mohr, D. Kranz, and R. Halstead. Lazy task creation - a technique for increasing the granularity of parallel programs. *IEEE Transactions on Parallel and Distributed Systems*, 2(3), July 1991.

[24] A. Moran, S. Lassen, and S. Peyton Jones. Imprecise exceptions, co-inductively. In *Higher Order Operational Techniques in Semantics: Third International Workshop*, number 26 in Electronic Notes in Theoretical Computer Science, pages 137–156. Elsevier, 1999.

[25] R. Osbourne. *Speculative computation in Multilisp*. PhD thesis, MIT Lab for Computer Science, Dec. 1989.

[26] W. Partain. The `nofib` benchmark suite of Haskell programs. In J. Launchbury and P. Sansom, editors, *Functional Programming, Glasgow 1992*, Workshops in Computing, pages 195–202. Springer Verlag, 1992.

[27] S. Peyton Jones. Implementing lazy functional languages on stock hardware: The spineless tagless G-machine. *Journal of Functional Programming*, 2(2):127–202, Apr. 1992.

[28] S. Peyton Jones, C. Hall, K. Hammond, W. Partain, and P. Wadler. The Glasgow Haskell Compiler: a technical overview. In *Proceedings of Joint Framework for Information Technology Technical Conference, Keele*, pages 249–257. DTI/SERC, Mar. 1993.

[29] S. Peyton Jones, S. Marlow, and A. Reid. The STG runtime system (revised). Technical report, Microsoft Research, February 1999. Part of GHC source package.

[30] S. Peyton Jones and W. Partain. Measuring the effectiveness of a simple strictness analyser. In Hammond and O'Donnell [11], pages 201–220.

[31] S. Peyton Jones, A. Reid, C. Hoare, S. Marlow, and F. Henderson. A semantics for imprecise exceptions. In *ACM Conference on Programming Languages Design and Implementation (PLDI'99)*, pages 25–36, Atlanta, May 1999. ACM.

[32] S. Peyton Jones and P. Wadler. Imperative functional programming. In *20th ACM Symposium on Principles of Programming Languages (POPL'93)*, pages 71–84. ACM, Jan. 1993.

[33] K. Schauser, D. Culler, and S. Goldstein. Separation constraint partitioning: a new algorithm for partitioning non-strict programs into sequential threads. In *22nd ACM Symposium on Principles of Programming Languages (POPL'95)*, pages 259–271. ACM, Jan. 1995.

[34] P. Sestoft. Deriving a lazy abstract machine. *Journal of Functional Programming*, 7, 1997.

[35] J. E. Smith. A study of branch prediction strategies. In *International Symposium on Computer Architecture*, 1981.

[36] Sun Microsystems. *The Java HotSpot Virtual machine, White Paper*, 2001.

[37] K. Traub. *Sequential implementation of lenient programming languages*. Ph.D. thesis, MIT Lab for Computer Science, 1988.

[38] C. von Dorrien. Stingy evaluation. Licentiate thesis, Chalmers University of Technology, May 1989.

[39] P. Wadler and J. Hughes. Projections for strictness analysis. In G. Kahn, editor, *Functional Programming Languages and Computer Architecture*. Springer Verlag LNCS 274, Sept. 1987.

A Table of Test Statistics

	Normal GHC -O2					Optimistic Evaluation Relative to Normal GHC						Thunk Statistics	
test name	lines	runtime	code size	allocations	residency	runtime	warmup	code size	allocations	residency	aborts	always	usual
digits-of-e1	46	10.61	522,824	554,792,276	94,656	94.65%	108.79%	164.74%	140.26%	107.85%	2	64.90%	34.99%
digits-of-e2	57	12.64	522,751	363,262,024	238,200	98.26%	131.59%	165.67%	101.52%	101.36%	9	20.5%	79.75%
exp3_8	93	27.30	519,011	1,458,709,812	78,844	61.19%	203.74%	163.30%	43.49%	133.17%	1	100.0%	0.0%
gen_regexps	41	10.80	488,635	337,306,492	10,143,800	83.99%	95.78%	163.19%	72.1%	100.0%	2	99.79%	0.0%
integrate	42	10.40	751,886	908,304,308	12,245,472	51.8%	55.0%	180.46%	60.16%	0.3%	5	92.98%	1.87%
paraffins	91	9.70	537,518	164,095,080	71,835,148	75.94%	76.7%	166.11%	99.85%	101.37%	0	99.76%	0.0%
primes	18	11.24	506,599	369,988,932	154,252	83.22%	127.22%	163.19%	100.93%	73.66%	2	1.71%	96.41%
queens	62	19.95	500,829	941,718,572	788	71.63%	71.14%	162.57%	29.69%	157.36%	0	99.99%	0.0%
rfib	12	9.77	757,478	84,740	0	99.84%	99.22%	180.28%	*922.96%	*NaN	3	14.85%	0.0%
tak	16	10.38	500,991	955,987,316	1,032	102.74%	111.58%	162.79%	106.72%	100.0%	0	93.91%	0.0%
wheel-sieve1	41	10.23	515,576	50,988,176	6,155,720	83.10%	83.74%	163.8%	87.76%	97.33%	2	41.4%	58.94%
wheel-sieve2	47	9.3	518,892	150,251,220	49,760,420	99.91%	99.82%	163.21%	99.48%	135.81%	6	1.74%	98.16%
x2n1	35	2.36	820,512	185,939,020	1,800	80.5%	84.4%	168.50%	55.49%	169.78%	0	99.99%	0.0%
wordcount	34	38.23	486,874	162,869,256	5,076,872	*2.43%	*3.3%	164.10%	97.78%	0.97%	0	99.99%	0.0%
atom	187	22.37	636,363	588,339,292	284,360	29.40%	38.31%	216.29%	49.51%	28.89%	9	64.13%	20.51%
boyer	1,018	13.45	563,257	952,727,688	94,788	94.94%	96.73%	161.69%	94.56%	100.34%	6	32.11%	11.84%
circsim	668	10.63	624,079	651,307,992	19,329,720	78.99%	78.52%	174.12%	76.65%	86.61%	19	33.38%	0.35%
constraints	271	28.36	557,799	963,221,104	20,517,624	106.73%	115.87%	167.72%	93.43%	135.24%	22	7.7%	15.19%
anna	9,561	4.76	2,153,033	143,440,624	1,600,628	84.16%	95.17%	188.16%	78.67%	81.11%	69	21.12%	32.71%
infer	591	10.95	755,395	175,015,092	23,385,296	98.54%	141.74%	171.35%	88.55%	140.16%	15	3.57%	30.21%
gamteb	701	5.95	757,109	399,989,284	4,593,220	72.61%	93.45%	208.67%	71.12%	137.40%	12	59.21%	26.33%
rsa	74	4.55	591,447	192,956,252	99,244	96.48%	99.56%	169.31%	102.6%	105.5%	9	99.91%	0.0%
symalg	1,146	5.34	1,043,898	409,901,572	473,168	69.66%	72.85%	175.5%	100.16%	102.94%	3	49.31%	7.90%
minimum	12	2.36	486,874	84,740	0	2.43%	3.3%	161.69%	29.69%	0.3%	0	1.71%	0.0%
maximum	9,561	38.23	2,153,033	1,458,709,812	71,835,148	106.73%	203.74%	216.29%	922.96%	*NaN	69	100.0%	98.16%
geometric mean*						79.90%	93.95%	171.80%	79.75%	104.3%		57.72%	36.42%

Explanation of columns:

lines	Number of lines of code in each example. Programs from the NoFib “imaginary” suite are fairly small, but other programs are bigger.
runtime	Number of seconds the program took to run. We adjusted the programs to aim to run for a reasonable amount of time.
warmup	How long the initial “warmup” run took to execute, before a good configuration had been found.
code size	The size of the statically linked program binary produced.
allocations	Bytes of heap allocated during the test run.
residency	The maximum amount of “live” data found during a GC.
aborts	The number of abortions that took place in the warmup run and the test run together.
always and **usual**	The proportion of thunks allocated that are “always used” or “cheap and usually used”.

Note (*) that for the geometric mean, extreme outliers (e.g. the runtime of wordcount) were removed to prevent them distorting the results.
Note also that some stats runs were done with smaller data sets to other test runs, to allow them to run in reasonable time.

Comments on results:

runtime	Most programs speed up. Some by a lot, most by a bit. In the worst case, we only slow down a little.
code size	Programs are considerably larger due to extra entry points being needed. Generating such entry points on demand should prevent this.
warmup	Generally rather slower than later runs. `exp3_8` goes a lot slower due to expensive profiling.
allocations	Some decrease as less thunks are allocated. Some increase by evaluating unneeded things. `rfib` isn't significant as very little is allocated.
residency	This generally goes up a little due to chunkily evaluated data structures. Can go down a lot if we prevent a lazy heap explosion.
paraffins and symalg	These go faster, despite not seeming to have less allocations or residency. We are currently unsure as to why.