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 most of these thunks are unnecessary: they are in fact always evaluated, or are always cheap. In this paper we describe Optimistic Evaluation — an evaluation strategy 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 expressions found to be unsuitable for speculative evaluation, and arranges for them to be evaluated more lazily in the future.

We have implemented optimistic evaluation in the Glasgow Haskell Compiler. The results are encouraging: many programs speed up significantly (5-25%), some improve dramatically, and none go more than 15% slower.

Categories and Subject Descriptors

D.3.2 [**Programming Languages**]: Language Classifications—*Applicative (functional) languages, Lazy Evaluation*; D.3.4 [**Programming Languages**]: Processors—*Code generation, optimization, profiling*

General Terms

Performance, Languages, Theory

Keywords

Lazy Evaluation, Online Profiling, Haskell

1. INTRODUCTION

Lazy evaluation is great for programmers [16], 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 [40]; while *cheapness analysis* locates expressions that are certain to be cheap (and safe) to evaluate [7].

However, any static analysis must be conservative, forcing it to keep thunks that are "probably unnecessary" but not "provably unnecessary". 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 7.1.). 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 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 significantly.

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 eager computations that go on for too long. We make the following contributions:

- The idea of optimistic evaluation *per se* is rather obvious (see Section 7). What is *not* obvious is how to (a) make it *cheap enough* that the benefits exceed the costs; (b) avoid potholes: it is much easier to get big speedups on some programs if you accept big slowdowns 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) [29].

ICFP'03, August 25-29, 2003, Uppsala, Sweden.

- Optimistic evaluation is a slippery topic. We give an *operational semantics* for a lazy evaluator enhanced with optimistic evaluation (Section 5). This abstract machine allows us to refine our general design choices into a precise form.

The performance numbers are encouraging: We have produced a stable extended version of GHC that is able to compile arbitrary Haskell programs. When tested on a set of reasonably large, realistic programs, it produced a geometric mean speedup of just over 15% with no program slowing down by more than 15%. Perhaps more significantly, naively written programs that suffer from space leaks can speed up massively, with improvements of greater than 50% being common.

These are good results for a compiler that is as mature as GHC, where 10% is now big news. (For example, strictness analysis buys 10-20% [31].) 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.

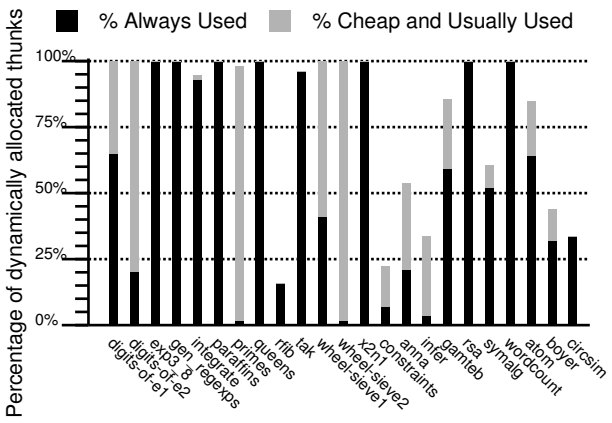

Figure 1: How thunks are commonly used

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 5.1). The back end is then modified as follows:

- Each **let** expression is compiled such that it can either build a thunk for its right hand side, or evaluate its right hand side speculatively (Section 2.1). Which of these it does depends on the state of a run-time adjustable *switch*, one for each **let** expression. The set of all such switches is known as the *speculation configuration*. The configuration is initialised so that all **let***s* are speculative.

- If the evaluator finds itself speculatively evaluating a very expensive expression, then *abortion* is used to suspend the speculative computation, building a special continuation thunk in the heap. Execution continues with the body of the speculative **let** whose right-hand side has now been suspended (Section 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 **let***s* are expensive to evaluate, or are rarely used (Section 3). The profiler then modifies the speculation configuration, by switching off speculative evaluation, so that the offending **let***s* evaluate their right hand sides lazily rather than strictly.

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. Identifying and addressing these concerns is one of our main contributions. To avoid continual asides, we tackle related work in Section 7. We use the term "speculative evaluation" to mean "an evaluation whose result may not be needed", while "optimistic evaluation" describes the entire evaluation strategy (abortion, profiling, adaption, etc).

2.1 Switchable Let Expressions

In the intermediate language consumed by the code generator, the **let** expression is the only construct that allocates thunks. In our design, each **let** chooses dynamically whether to allocate a thunk, or to evaluate the right-hand side of the **let** speculatively instead. Here is an example:

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

The code generator 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.

If `<rhs>` was large, then this compilation scheme could result in code bloat. We avoid this by lifting the right hand sides of such **let** expressions out into new functions.

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 a 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 run-time 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 also be implemented in many different ways. Our current scheme shares its implementation with that already used for handling asynchronous exceptions [21]. A suspension is created in the heap containing the aborted computation. If the result is found to be needed, then this computation will resume 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, so it does not need to be particularly efficient.

Abortion alone is enough to guarantee 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 **let**s, so that they run lazily in future (many strategies are possible). From this we get a crude bound on the total time and space consumed by aborted computations. Since there are finitely many **let**s 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.

2.3 Chunky Evaluation

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

```
from :: Int -> [Int]
from n = let n1 = n+1 in
         let rest = from n1 in
         (n : rest)
```

If we used speculative evaluation for the `rest` thunk, the evaluation would be certain to abort. Why? Because evaluating `from n` would speculatively evaluate `from (n+1)`, which would speculatively evaluate `from (n+2)`, and so on. It will not be long before we turns off speculative evaluation of `rest`, making `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 `n1` thunks 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 behaviour 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 speculation is nested. The code for a **let** then behaves semantically like the following (note that this generalises the code given in Section 2.1):

```
if (SPECDEPTH < LIMIT237){
    SPECDEPTH = SPECDEPTH + 1
    x = value of <rhs>
    SPECDEPTH = SPECDEPTH - 1
}else{
    x = lazy thunk for <rhs>
}
evaluate <body>
```

Here, `SPECDEPTH` is a count of the number of nested speculations that we are currently inside and `LIMIT237` is a limit on how deeply this particular **let** can be speculated. The intuition is that the deeper the speculation stack, the less likely a new speculation is to be useful. `LIMIT237` can be adjusted at run-time to control the extent to which this **let** may be speculated.

2.4 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 [26].

In GHC, `error` raises a catchable exception, rather than halting the program [32]. 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 a speculative-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.

2.5 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 [33], so there is no danger of speculative computations performing I/O. However, Haskell programs sometimes make use of impure I/O, using the "function" `unsafePerformIO`. Speculatively evaluating calls to this function could cause observable behaviour

different to that of a lazy implementation. Moreover, it is not safe for us to abort `IO` computations because an IO operation may have locked a resource, or temporarily put some global data structures into an invalid state.

For these reasons, we have opted to disallow speculation of `unsafePerformIO` and `unsafeInterleaveIO`. Any speculation which attempts to apply one of these functions will abort immediately.

3. 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, some programs run faster, but some run dramatically slower. For example the `constraints` program from NoFib runs over 30 times slower. 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 time and space.

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

In order to know whether an expression should be speculated or not, we need to know whether the amount of work potentially wasted outweighs the amount of lazy evaluation overhead potentially avoided. We obtain this information by using a form of online profiling.

3.1 Idealised Profiling

We begin by describing an idealised profiling scheme, in which every **let** is profiled all the time. For every **let**, we maintain two static counters:

`speccount` is incremented whenever a speculative evaluation of the right hand side is begun.

`wastedwork` records the amount of work wasted by as-yet-unused speculations of the **let**.

The quotient `(wastedwork/speccount)` is the *wastage quotient* - the average amount of work wasted work per speculation. If the wastage quotient is larger than the cost of allocating and updating a thunk, then speculation of the **let** is wasting work, and so we reduce the extent to which it is speculated. (In fact, the critical threshold is tunable in our implementation.)

By "work" we mean any reasonable measure of execution cost, such as time or allocation. In the idealised model, we just think of it as a global register which is incremented regularly as execution proceeds; we discuss practical implementation in Section 3.4.

The `wastedwork` we record for a speculation includes the work done to compute its value, but *not* the work of computing any sub-speculations whose values were not needed. Operationally, we may imagine that the global work register is saved across the speculative evaluation of a **let** right-hand side, and zeroed before beginning evaluation of the right-hand side.

But what if a sub-speculation *is* needed? Then the profiler should attribute the same costs as if lazy evaluation had been used. For example, consider:

```
let
   x = let y = <expensive> in y+1
in ...
```

The evaluation of `x` will speculatively evaluate `<expensive>`, *but it turns out that* `y` *is needed by* `x`, so the costs of `<expensive>` should be attributed to `x`, not `y`. Speculating `y` is perfectly correct, because its value is always used.

How do we implement this idea? When the speculation of `y` completes, we wrap the returned value in a special indirection closure that contains not only the returned value v, but also the amount of work w done to produce it, together with a reference to the **let** for `y`. When the evaluation of `x` needs the value of `y` (to perform the addition), it evaluates the closure bound to `y` (just as it would if `y` were bound to a thunk). When the indirection is evaluated, it *adds* the work w to the global work counter, and *subtracts* it from `y`'s static `wastedwork` counter, thereby transferring all the costs from `y` to `x`.

There is one other small but important point: when speculating `y` we save and restore the work counter (for `x`), but we also increment it by one thunk-allocation cost. The motivation is to ensure the cost attribution to `x` is the same, regardless of whether `y` is speculated or not.

This scheme can attribute more cost to a speculative evaluation than is "fair". For example:

```
f v = let x = v+1 in v+2
```

The speculative evaluation of `x` will incur the cost of evaluating the thunk for `v`; but in fact `v` is needed anyway, so the "real" cost of speculating `x` is tiny (just incrementing a value). However, it is safe to over-estimate the cost of a speculation, so we simply accept this approximation.

3.2 Safety

We would like to guarantee that if a program runs more slowly than it would under lazy evaluation, the profiler will spot this, and react by reducing the amount of speculation.

Our profiler assumes that all slowdown due to speculation is due to wasted work[1]in speculations. If work is being wasted then at least one **let** must have a wastage quotient greater than one thunk allocation cost. When the profiler sees a **let** with this property, it reduces the amount of speculation done for that **let**. Eventually, the speculation configuration will either settle to one that is faster than lazy evaluation, or in the worst case, all speculation will be disabled, and execution reverts back to lazy evaluation.

This safety property also holds for any approximate profiling scheme, provided that the approximate scheme is conservative and overestimates the total amount of wasted work.

3.3 Random Sampling

It would be inefficient to profile every **let** all the time, so instead we profile a random selection of speculations. At regular intervals, called *sample points*, we profile each active speculation, *from the sample point until it finishes*. We double the work measured to approximate the complete execution cost of the speculation.

There are two sources of error. First, since we sample randomly through time, we are likely to see expensive speculations more frequently than inexpensive ones. However,

[1]In fact, as we see in section 6.2, space usage is often a major factor.

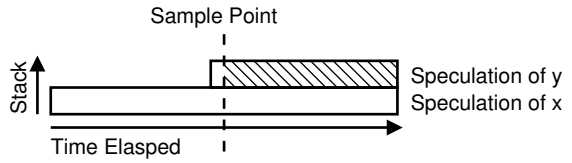

Figure 2: Underestimating Wasted Work

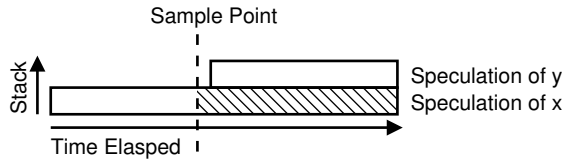

Figure 3: Overestimating Wasted Work

we believe that in practice, this should not be a problem, particularly as it is the expensive speculations that are likely to be the ones that are wasting the most work.

Second, we can reasonably assume that the sample point is uniformly distributed between the moment a speculation begins and the moment it ends; however, *the work ticks attributed to that speculation may not be uniformly distributed over that period.* For example:

```
let x = if <expensive1> then
            let y = <expensive2> in (y,y)
else ...
```

Most of the work attributed to `x` is the cost of evaluating `<expensive1>`; the work of evaluating `<expensive2>` will be attributed to `y`. Furthermore, evaluation of `<expensive1>` precedes the evaluation of `<expensive2>`. Hence, if we were to place our sample point in the middle of the speculation for `x`, then we would get an under-estimate of the work done. This is illustrated by Figure 2, in which work counted is shaded and work ignored is unshaded. Alas, under-estimates are not safe!

However, it turns out that this is not a problem, for a rather subtle reason. In our example, the speculation of `x` does no work after the speculation of `y` has finished. Let us assume that `<expensive1>` does exactly the same amount of work W as `<expensive2>`. Hence, the sample point will have a 50% chance of falling before the start of the speculation of `y`, and a 50% chance of falling after it. If the sample point falls after the speculation of `y` starts, then we will attribute no work to `x`, thus underestimating the work by W units (Figure 2). However, if the sample point falls before the speculation of `y` starts, then *we will not profile the speculation of* `y` and so will attribute all of the work done by `<expensive2>` to `x`; thus overestimating the work by an average of $2W$ units (Figure 3) – the factor of 2 comes from doubling the measured cost, as mentioned above. The net effect is that we still over-estimate `x`'s cost.

This example, in which all `x`'s work is front-loaded, is an extreme case. The informal argument we have given can be generalised to show that our sampling approach can only over-estimate costs.

3.4 Implementing Profiling

We implement profiling by overwriting return frames on the stack. At a sample point, we walk down the stack and find the return frames for all active speculations. Each of these return frames is overwritten with the address of a profiling system routine, while the real return address is squirreled away for safe-keeping.

When the speculation finishes, it will jump back into the profiling system through this hijacked return point. The profiler now calculates an estimate of the work done during the speculation. Available timers are not accurate enough to allow for an accurate measure of the time elapsed during a speculation, so we use the amount of heap allocated by the speculation instead. This seems to be a fairly reasonable estimate, provided that we ensure that all recursions allocate heap.

4. IS THIS TOO COMPLEX?

By this time, the reader may be thinking "isn't this all rather complicated?". Does one really need abortion, switchable let expressions, chunky evaluation, special handling for IO, *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 this 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 6.4 quantifies the effect of removing individual "bullets".

Second, the actual implementation in GHC is not overly complex. The changes to the compiler itself are minor (around 1,300 lines in a program of 130,000 or so). The changes to the run-time system are more significant: In a run-time system of around 47,000 lines of C, we have added around 1000 lines and altered around 1,600 lines — certainly not trivial, but a 3% increase seems reasonable for a major innovation.

5. 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 speculative evaluation, abortion, and adaption. In this section, we briefly present this semantics.

5.1 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
	|	$C\ \{x_i\}_0^n$	constructor app
	|	**case** E **of** $\{P_i\}_0^n$	case analysis
	|	**let** $x = E$ **in** E'	thunk creation
	|	$\lambda x.E$	abstraction
	|	$E\ x$	application
	|	**exn**	exception or error
P	::=	$C\ \{x_i\}_0^n \rightarrow E$	pattern match
V	::=	$C\ \{x_i\}_0^n \mid \lambda x.E \mid$ **exn**	values

(VAR1)	$\Gamma[x \mapsto V]; x; s$	$\longrightarrow_\Sigma$	$\Gamma[x \mapsto V]; V; s$	
(VAR2)	$\Gamma[x \mapsto E]; x; s$	$\longrightarrow_\Sigma$	$\Gamma[x \mapsto E]; E; \#x : s$	if E is not a value
(UPD)	$\Gamma; V; \#x : s$	$\longrightarrow_\Sigma$	$\Gamma[x \mapsto V]; V; s$	
(APP1)	$\Gamma; E\ x; s$	$\longrightarrow_\Sigma$	$\Gamma; E; (\bullet\ x) : s$	
(APP2)	$\Gamma; \lambda x.E; (\bullet\ x') : s$	$\longrightarrow_\Sigma$	$\Gamma; E[x'/x]; s$	
(APP3)	$\Gamma; \mathbf{exn}; (\bullet\ i) : s$	$\longrightarrow_\Sigma$	$\Gamma; \mathbf{exn}; s$	
(CASE1)	$\Gamma; \mathbf{case}\ E\ \mathbf{of}\ \{P_i\}_0^n; s$	$\longrightarrow_\Sigma$	$\Gamma; E; \{P_i\}_0^n : s$	
(CASE2)	$\Gamma; C\ \{x'_i\}_0^n; \{P_i\}_0^n : s$	$\longrightarrow_\Sigma$	$\Gamma; E[\{x'_i/x_i\}_0^n]; s$	where $P_k = C\ \{x_i\}_0^n \to E$
(CASE3)	$\Gamma; \mathbf{exn}; \{P_i\}_0^n : s$	$\longrightarrow_\Sigma$	$\Gamma; \mathbf{exn}; s$	
(LAZY)	$\Gamma; \mathbf{let}\ x = E\ \mathbf{in}\ E'; s$	$\longrightarrow_\Sigma$	$\Gamma[x' \mapsto E]; E'[x'/x]; s$	if $\Sigma(x) \leq D(s)$ and x' is new
(SPEC1)	$\Gamma; \mathbf{let}\ x = E\ \mathbf{in}\ E'; s$	$\longrightarrow_\Sigma$	$\Gamma; E; x \nearrow E' : s$	if $\Sigma(x) > D(s)$
(SPEC2)	$\Gamma; V; x \nearrow E : s$	$\longrightarrow_\Sigma$	$\Gamma[x' \mapsto V]; E[x'/x]; s$	where x' is new

Figure 4: Operational Semantics: Evaluation

The binder x is unique for each **let**. This allows us to uniquely refer to a **let** by its binder.

Functions and constructors are always applied to variables, rather than to expressions. It follows that **let** is the only point at which a thunk might be created. (A `case` expression scrutinises an arbitrary expression E, but it does not first build a thunk.).

We restrict ourselves to non-recursive **let** expressions. This does not restrict the expressiveness of the language, as the user can use a fixed point combinator. It does however simplify the semantics, as recursive **let** expressions are problematic when speculated. We omit literals and primitive operators for the sake of brevity. Adding them introduces no extra complications.

Exceptions and errors (Section 2.4) are handed exactly as described in [32], namely by treating an exception as a *value* (and not as a control operator).

5.2 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, $\longrightarrow$, takes the form:

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

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

- Γ represents the heap. It is a function mapping names to expressions.
- E is the expression currently being evaluated.
- s is a stack of *continuations* containing work to be done, each taking one of the following forms (c.f. [11]):

$x \nearrow E$	Speculation	return to E, binding x
$\{P_i\}_0^n$	Case	choose a pattern P_i
$\bullet\ x$	Application	use arg x
$\#x$	Update	of thunk bound to x

  We write $D(s)$ to denote the number of speculation frames on the stack s. We refer to this as the *speculation depth*.

- Σ is the *speculation configuration*. Σ maps a **let** binder x to a natural number n. This number says how deeply we can be speculating and still be allowed to create a new speculation for that **let**. If the **let** is lazy, then n will be 0. Online profiling may change the mapping Σ while the program is running.

The transition rules for $\longrightarrow$ are given in Figure 4. The first nine are absolutely conventional [35], while the last three are the interesting ones for optimistic evaluation. Rule *(LAZY)* is used if the depth limit $\Sigma[x]$ is less than the current speculation depth $D(s)$. The rule simply builds a thunk in the heap Γ, with address x', and binds x to x'. Rule *(SPEC1)* is used if $\Sigma[a]$ is greater than the current speculation depth $D(s)$; it pushes a return frame and begins evaluation of the right-hand side. On return, *(SPEC2)* decrements d and binds x to (the address of) the computed value.

At a series of *sample points*, the online profiler runs. It can do two things: first, it can change the 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 5. 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 speculative-let continuation (**let** $x = \bullet$ **in** E) has been removed from the stack, and indeed our implementation stops at that point.

One merit of having an operational semantics is that it allows us to formalise the profiling semantics described in Section 3. We have elaborated the semantics of Figure 4 to describe profiling and cost attribution, but space prevents us showing it here.

6. PERFORMANCE

We measured the effect of optimistic evaluation on several programs. Most of the programs are taken from the `real` subset of the NoFib [28] suite, with a few taken from the `spectral` subset. These programs are all reasonably sized, realistic programs. GHC is 132,000 lines, and the mean of the other benchmarks is 1,326 lines. Programs were selected before any performance results had been obtained for them.

$$
\begin{array}{lrcl}
\textit{(!SPEC)} & \Gamma; E; x \uparrow E' : s & \rightsquigarrow & \Gamma[x' \to E[x'/x]]; E'; s \quad \text{where } x' \text{ is new.} \\
\textit{(!CASE)} & \Gamma; E; \{P_i\}_0^n : s & \rightsquigarrow & \Gamma; \textbf{case } E \textbf{ of } \{P_i\}_0^n; s \\
\textit{(!APP)} & \Gamma; E; (\bullet\ x) : s & \rightsquigarrow & \Gamma; E\ x; s \\
\textit{(!UPD)} & \Gamma; E; \#x : s & \rightsquigarrow & \Gamma[x \mapsto E]; x; s
\end{array}
$$

Figure 5: Operational Semantics : Abortion

We ran each program with our modified compiler, and with the GHC compiler that our implementation forked from. The results for normal GHC were done with all optimisations enabled. Tests were performed on a 750MHz Pentium III with 256Mbytes of memory. The implementation benchmarked here does not remember speculation configurations between runs. To reduce the effect of this cold start, we arranged for each benchmark to run for around 20 seconds. We will now summarise the results of our tests. For full results, please refer to Appendix A.

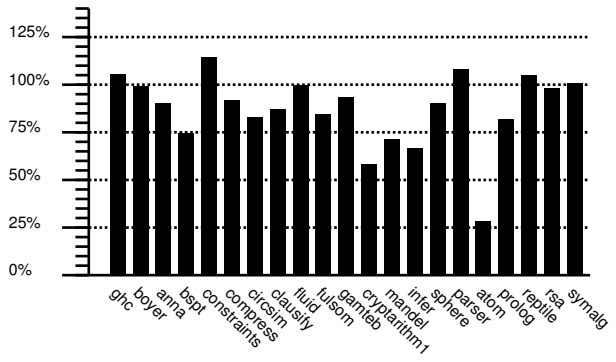

Figure 6: Benchmark Run-Time relative to Normal GHC

6.1 Execution Time

Figure 6 shows the effect that optimistic evaluation has on run time. These results are very encouraging. The average speedup is just over 15%, some programs speed up dramatically and no program slows down by more than 15%. As one would expect, the results depend on the nature of the program. If a program has a strict inner loop that the strictness analyser solves, then we have little room for improvement. Similarly, if the inner loop is inherently lazy, then then there is nothing we can do to improve things, and indeed the extra overhead of having a branch on every **let** will slow things down. In the case of `rsa` speculation had virtually no effect as `rsa` spends almost all of its time inside a library written in `C`.

These results reflect a single fixed set of tuning parameters, such as the thunk-cost threshold (Section 3.1), which we chose based on manual experimentation. Changing these parameters can improve the run-time of a particular program significantly (up to 25% or so), but only at the cost of worsening another. Whether a more sophisticated profiling strategy could achieve the minimal run-time for every program remains to be seen.

6.2 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 [24]. 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 =
  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!

The "residency" column of Appendix A shows the effect on maximum heap size on our benchmark set, with a mean of 85% of the normal-GHC figure. Not surprisingly, the improvement is much smaller than that of `count` — perhaps because these real programs have already had their space leaks cured — and some programs use more space than before. Nevertheless, optimistic evaluation seems to reduce the prevalence of unexpectedly-bad space behaviour, a very common problem for Haskell programmers, and that is a welcome step forward.

6.3 Code Size

Code size increases significantly (34% on average). This is due to the need to generate both lazy and strict versions of expressions. 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.

6.4 Where the Performance comes from

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

Figure 7 shows the performance of several simplified variants of our system relative to our full optimistic implementation. Figure 8 shows the performance of several altered versions of normal GHC relative to normal GHC.

Chunky evaluation. The "No Chunky" bars in Figure 7 show the effect on switching off chunky evaluation. When

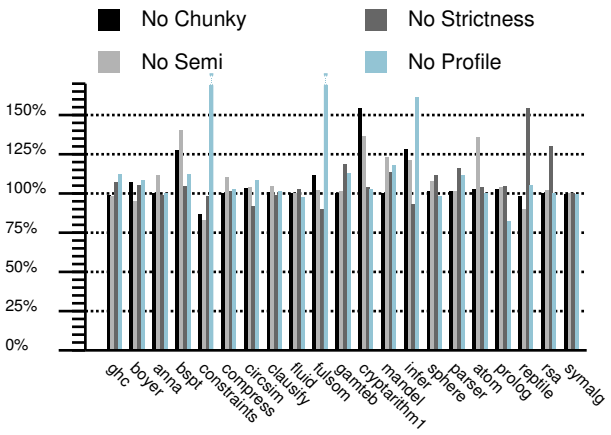

Figure 7: Run times of Simplified Implementations

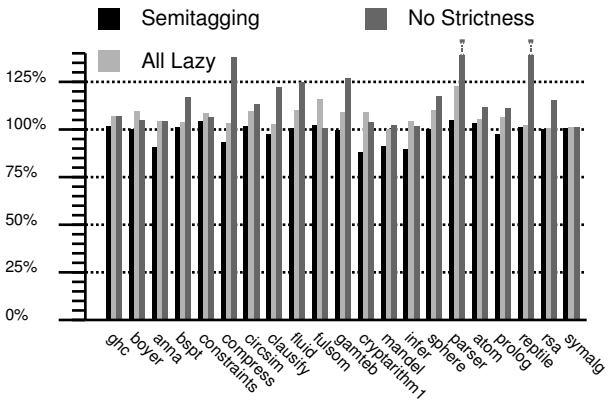

Figure 8: Effect of Changes to Normal GHC

chunky evaluation is turned off, a **let** expression can only be on or off, with no reduction in depth allowed for in-between cases. Some programs are largely unaffected by this, but others suffer significantly. `constraints` actually speeds up.

Semi-tagging. Our implementation of optimistic evaluation uses an optimisation 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* [30]: 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 "Semitagging" bars in Figure 8 show the effect of enabling semi-tagging only relative to *baseline* GHC (i.e. no optimistic evaluation). Under normal lazy evaluation, the scrutinee is often unevaluated, so while semi-tagging is a win on average, the average speedup is only 2%.

However, *under optimistic evaluation most scrutinees are evaluated*, and semi-tagging gives a consistent improvement. The "No Semi" bars in Figure 7 are given relative to the full optimistic evaluator, and show that switching off semi-tagging almost 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.

Strictness Analysis. The "No Strictness" bars show the effect of turning off GHC's strictness analysis. Figure 8 shows that strictness analysis is usually a very big win for normal GHC, while in Figure 7 we see that the effect of switching off strictness analysis in an optimistically-evaluated implementation is far smaller. Hence, in the absence of strictness analysis, the win from optimistic evaluation would be far greater than the ones we report in Figure 6.

Profiling. The "No Profile" bars in Figure 7 show the effect of disabling online profiling. Most programs are largely unaffected, but a few programs such as `constraints` and `fulsom` slow down massively. Programs like these justify our work on profiling (Section 3); our goal is to give acceptable performance in *all* cases, without occasional massive and unpredictable slow-downs.

Overheads. The "All lazy" bars in Figure 8 show what happens when we pay all the costs of optimistic evaluation, but get none of the benefits. In this experiment, we set Σ to map every **let** to 0, so that all **let**s are done lazily. Comparing this to Normal GHC, the baseline for this graph, shows the overheads that the profiler and the switchable-let mechanism impose on normal evaluation; they are always less than 25%.

7. RELATED WORK

7.1 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. (When we switch it off, the speedups from optimistic evaluation are much greater.) The other promising static analysis is Faxén's *cheap eagerness analysis* [7], 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 [8], 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 online 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.)

There exist many other static analyses that can improve the performance of lazy programs. In particular, the GRIN project [3] takes a different spin on static analysis. A whole program analysis is used to discover for each `case` expression the set of all thunk expressions that might be being evaluated at that point. The bodies of these expressions are then inlined at the usage sites, avoiding much of the cost of lazy evaluation. Such transformations do not however prevent lazy space leaks.

7.2 Eager Haskell

Eager Haskell [20, 19] 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 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 slows by only 15%. 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 cases 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.

7.3 Speculative Parallelism

The parallel programming community has been making use of speculation to exploit parallel processors for a long time [5]. 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 [12, 27]. Mattson [22, 23] speculatively evaluates lazy expressions in parallel. *Local Speculation* [23, 6] does some speculations on the local processor when it would otherwise be waiting, reducing the minimum size for a useful speculation. Huntback [17] 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 cancelled 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 [14]. 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 [38, 34]. In some ways this is closer to our work: call-by-need is a bit 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 [25]. A very successful variant of this idea is Cilk [10], 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.

7.4 Other Work

Stingy Evaluation [39] is an evaluation strategy designed to reduce space leaks such as the one described in Section 6.2. When evaluating a **let** expression, or during garbage collection, 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. Stingy evaluation was implemented in the LML [1] compiler.

Branch Prediction [36] 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].

Online Profiling is used in many existing language implementations, including several implementations for the Java [18] language, such as HotSpot [37] and Jalapeno [4]. One of the first implementations to use such techniques was for Self [15]. These systems use similar techniques to optimistic evaluation, but do not apply them to laziness.

Feedback Directed Optimisation [9] is a widely used technique in static compilers. A program is run in a special profiling mode, recording statistics about the behaviour of the program. These statistics are then used by the compiler to make good optimisation 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.

8. 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 14%, 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:

Heap Usage Profiling. As shown in Section 6.2, optimistic evaluation often speeds programs up by preventing them from filling the heap up with thunks. However our current system does not take heap behaviour 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 behaviour 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 5, preliminary work suggests that we can indeed prove that the online profiler will eventually find all **let** expressions that waste work, and hence 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 [19] are the first to explore the rich territory of choosing dynamically between these two extremes.

The implementation described in this paper is freely available in the GHC CVS, and readers are encouraged to download it and have a look at it themselves. We plan to include Optimistic Evaluation in a release version of GHC soon.

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, Andy Pitts, Norman Ramsey, John Reppy, and Richard Sharp, who provided useful suggestions.

9. REFERENCES

[1] L. Augustsson and T. Johnsson. The chalmers lazy-ML compiler. *The Computer Journal*, 32(2):127–141, Apr. 1989.

[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] U. Boquist. *Code Optimisation Techniques for Lazy Functional Languages*. PhD thesis, Chalmers University of Technology, Sweden, April 1999.

[4] 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.

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

[6] 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.

[7] 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.

[8] K.-F. Faxén. Dynamic cheap eagerness. In *Proceedings of the 2001 Workshop on Implementing Functional Languages*. Springer Verlag, 2001.

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

[10] 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.

[11] 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.

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

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

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

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

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

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

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

[19] J.-W. Maessen. Eager Haskell: Resource-bounded execution yields efficient iteration. In *The Haskell Workshop, Pittsburgh*, 2002.

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

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

[22] 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.

[23] J. S. Mattson Jr and W. G. Griswold. Local Speculative Evaluation for Distributed Graph Reduction. In Hammond and O'Donnell [13], pages 185–192.

[24] 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.

[25] 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.

[26] 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.

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

[28] 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.

[29] 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.

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

[31] S. Peyton Jones and W. Partain. Measuring the effectiveness of a simple strictness analyser. In Hammond and O'Donnell [13], pages 201–220.

[32] 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.

[33] 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.

[34] 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.

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

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

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

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

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

[40] 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		Optimistic vs Normal GHC				Vs Normal GHC			Vs Optimistic			
test name	code lines	run time	code size	run time	code size	heap allocs	resi-dency	semi-tagging	all lazy	no stranal	no semi-tagging	no chunky	no profile	no stranal
ghc	132,042	18.27	18,201k	105%	158%	89%	105%	101%	106%	107%	98%	98%	112%	107%
boyer	1,018	17.39	754k	99%	124%	94%	91%	100%	109%	105%	95%	106%	108%	105%
anna	9,561	15.22	2,150k	90%	158%	90%	90%	90%	104%	104%	111%	100%	100%	98%
bspt	2,153	13.32	878k	74%	131%	82%	58%	101%	103%	116%	140%	127%	112%	104%
constraints	271	29.39	706k	114%	126%	95%	152%	104%	108%	106%	82%	86%	3484%	98%
compress	764	15.70	554k	91%	117%	98%	99%	93%	103%	138%	110%	100%	102%	101%
circsim	668	18.86	763k	83%	135%	78%	81%	101%	109%	112%	104%	103%	108%	91%
clausify	182	20.78	701k	86%	129%	58%	81%	97%	102%	122%	104%	100%	101%	98%
fluid	2,401	12.95	1,337k	99%	146%	79%	46%	100%	109%	124%	100%	100%	97%	102%
fulsom	1,444	14.17	1,172k	84%	157%	69%	23%	102%	115%	100%	101%	111%	176%	90%
gamteb	701	15.15	922k	93%	134%	88%	83%	99%	108%	126%	101%	100%	113%	118%
cryptarithm1	164	8.3	455k	58%	113%	70%	122%	87%	109%	103%	136%	154%	102%	103%
mandel	498	18.2	848k	71%	131%	59%	99%	90%	100%	102%	122%	100%	118%	113%
infer	591	14.55	864k	66%	134%	104%	143%	89%	104%	101%	121%	128%	161%	93%
sphere	472	11.98	899k	90%	137%	78%	187%	99%	109%	117%	107%	101%	98%	111%
parser	1,379	14.5	817k	107%	134%	84%	74%	104%	122%	150%	101%	101%	111%	115%
atom	187	22.39	828k	28%	131%	46%	40%	102%	105%	111%	135%	102%	100%	103%
prolog	641	20.61	646k	82%	124%	88%	113%	97%	106%	111%	103%	102%	82%	104%
reptile	1,522	17.98	966k	104%	133%	97%	78%	100%	101%	157%	90%	98%	105%	154%
rsa	74	17.79	718k	98%	126%	99%	106%	100%	100%	115%	101%	100%	100%	130%
symalg	1,146	17.85	1,037k	100%	135%	100%	100%	100%	101%	100%	100%	99%	99%	100%
minimum	74	8.3	455k	28%	113%	46%	23%	87%	100%	100%	82%	86%	82%	90%
maximum	132,042	29.39	18,201k	114%	158%	104%	187%	104%	122%	157%	140%	154%	3484%	154%
geometric mean				84%	134%	82%	85%	98%	106%	115%	107%	105%	128%	106%
arithmetic mean	7,518	16.88	1,725k											