Faster laziness using dynamic pointer tagging

Simon Marlow
Microsoft Research
simonmar@microsoft.com

Alexey Rodriguez Yakushev
University of Utrecht, The Netherlands
alexey@cs.uu.nl

Simon Peyton Jones
Microsoft Research
simonpj@microsoft.com

Abstract

In the light of evidence that Haskell programs compiled by GHC exhibit large numbers of mispredicted branches on modern processors, we re-examine the "tagless" aspect of the STG-machine that GHC uses as its evaluation model.

We propose two tagging strategies: a simple strategy called semi-tagging that seeks to avoid one common source of unpredictable indirect jumps, and a more complex strategy called dynamic pointer-tagging that uses the spare low bits in a pointer to encode information about the pointed-to object. Both of these strategies have been implemented and exhaustively measured in the context of a production compiler, GHC, and the paper contains detailed descriptions of the implementations. Our measurements demonstrate significant performance improvements (14% for dynamic pointer-tagging with only a 2% increase in code size), and we further demonstrate that much of the improvement can be attributed to the elimination of mispredicted branch instructions.

As part of our investigations we also discovered that one optimisation in the STG-machine, vectored-returns, is no longer worthwhile and we explain why.

1. Introduction

The Glasgow Haskell Compiler (GHC) is the most widely-used compiler for a lazy functional language. Since its inception, GHC has used the Spineless Tagless G-Machine (STG-machine) (Peyton Jones 1992) as its core execution model; for over 10 years this model remained largely unchanged, before eval/apply was adopted as an alternative to push/enter as the mechanism for function calls (Marlow and Peyton Jones 2004). In this paper we re-examine another aspect of the original STG-machine, namely the concept of "taglessness".

The *tagless* aspect of the STG-machine refers to the way in which a heap closure is evaluated. For example, consider

```
f x y = case x of (a,b) -> a+y
```

In a lazy language, before taking `x` apart the compiler must ensure that it is evaluated. So it generates code to push onto the stack a continuation to compute `a+y`, and jumps to the *entry code* for `x`. The first field of every heap closure is its entry code, and jumping to this code is called *entering* the closure. The entry code for an unevaluated closure will evaluate itself and return the value to the continuation; an already-evaluated closure will return immediately.

This scheme is attractive because the code to evaluate a closure is simple and uniform: any closure can be evaluated simply by entering it. But this uniformity comes at the expense of performing indirect jumps, one to enter the closure and another to return to the evaluation site. These indirect jumps are particularly expensive on a modern processor architecture, because they fox the branch-prediction hardware, leading to a stall of 10 or more cycles depending on the length of the pipeline.

If the closure is unevaluated, then we really do have to take an indirect jump to its entry code. However, if it happens to be evaluated already, then a conditional jump might execute much faster. In this paper we describe a number of approaches that exploit this possibility. We have implemented these schemes and show that they deliver substantial performance improvements (10-14%) in GHC, a mature optimising compiler for Haskell. Specifically, our contributions are these:

- We give the first accurate measurements for the dynamic behaviour of `case` expressions in lazy programs, across a range of benchmark programs. First, we measure how often a closure being scrutinised is already evaluated and, hence how often we can expect to avoid the enter-and-return sequence (Section 4). Second, we measure the distribution of data type sizes, which turns out to be important (Section 6).
- We describe and implement *semi-tagging*, a simple approach that avoids the enter-and-return associated with scrutinising an already-evaluated closure. Section 5 describes a full implementation and gives measurements of its effectiveness. As well as bottom-line execution times, we measure its effect on branch misprediction on a modern processor architecture. The improvements are substantial: execution time falls by 7-9%, while the branch misprediction rate is halved.
- We describe and implement an improvement to semi-tagging, called *dynamic pointer tagging*. The idea is to use the spare low bits of a pointer to encode (a safe approximation to) information about the pointed-to closure, even though that information can change dynamically. This technique appears to be novel. Again we describe the details of the implementation, which are less straightforward than for semi-tagging. Our measurements of its effectiveness are promising: a further 4-6% increase in performance (depending on processor architecture).
- We study the interaction of both these techniques with an existing optimisation called *vectored returns*. Vectored returns is an attractive optimisation that has been implemented in GHC for a decade. However, it carries a significant cost in terms of imple-

mentation complexity, and our new tagging schemes reduce its potential benefits. In Section 7.2 we present measurements that show that vectored returns are no longer attractive on today's processors, so the optimisation, and its attendant complexities, can be safely retired.

A major contribution of the paper is that our implementation and measurements are made in the context of a widely-used, highly-optimising compiler for Haskell, namely GHC; and our measurements are made against a large suite of benchmarks. Because GHC is a mature compiler, all the easy wins are part of the baseline, so a 10-15% improvement on real programs is a dramatic result.

The paper contains quite a bit of nitty-gritty detail, but that too is part of our contribution. Real compilers use many optimisation techniques, whose interaction is hard to foresee. We are able to report on concrete experience of implementing various tagging schemes, including their implementation costs as well as performance effects.

2. Compiling case expressions

In this section we review the relevant parts of the execution model and compilation scheme for the STG-machine implementation as it is in GHC today. Our particular focus is the compilation scheme for `case` expressions, such as the one in the body of `map`:

```
map :: (a->b) -> [a] -> [b]
map f xs = case xs of
                 []     -> []
                 (x:xs') -> f x : map f xs'
```

Informally, the semantics of `case` are as follows:

- The variable `xs` is evaluated to weak-head normal form (WHNF).
- The appropriate alternative is chosen according to the outermost constructor of the value of `xs`.
- If the pattern has variables, then these are bound to the fields of the constructor (in this example , `x` and `xs'` are bound in the second alternative).
- The appropriate right hand side is evaluated in this extended environment.

The precise semantics are given in the original STG-machine paper (Peyton Jones 1992).

Operationally, in the STG-machine implementation in GHC, the `case` expression is compiled to code that *enters* the variable being scrutinised, after pushing on the stack a return address, or continuation, for the alternatives. All heap objects have the uniform representation shown in Figure 1. The first word of every object is an *info pointer*, which points both to the *entry code* for the object, and to the preceding *info table* for the object. The info table describes the layout of the object to the garbage collector. It also contains a *object type field* which classifies the object as: a data constructor, a thunk, a function closure, or one of a variety of other object types. In the case of a data constructor, a further field in the info table gives the *tag* of the constructor; each constructor in a data type has a unique tag, starting at 0.

Because every heap object has associated entry code, we use the term "closure" for any heap object. The entry code for a closure always has the following meaning: it is executed with the register `R1` pointing to the closure, and it returns to the topmost stack frame having evaluated the closure pointed to by `R1`, returning the result in `R1`.

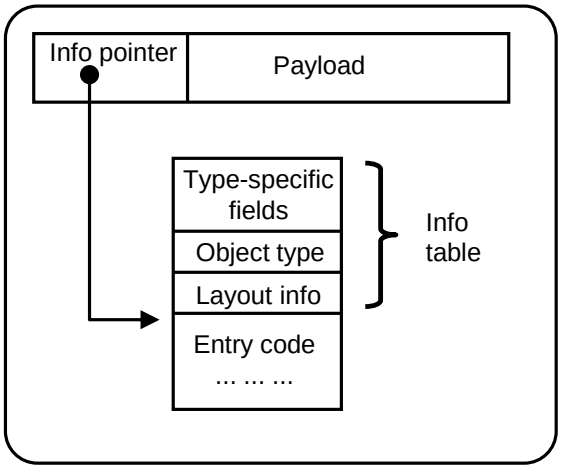

Figure 1. A heap object

GHC uses C-- (Peyton Jones et al. 1999) as its back-end intermediate language. C-- can be thought of as a high-level assembly language with nested expressions and named variables. GHC does not currently use the procedural abstractions of C--, instead it generates C-- code that directly manages an explicit stack. STG is translated into C-- before being turned into either C or native code. We will therefore present our generated code examples in C--.

Here, then, is the C-- code generated for the `map` example above, prior to the work described in this paper:

```
map_entry()  // On entry, f is in R2, xs in R3
{  // Save f on the stack, because it is
   // live in the continuation
   Sp[-4] = R2
   // Set R1 to point to the closure to evaluate
   R1 = R3;
   // Put the continuation on the stack
   Sp[-8] = map_ret;
   // Adjust the stack pointer
   Sp = Sp - 8;
   // Jump to the entry code for the closure in R1
   jump R1[0];
}

data { word32[] = {...} }  // Info table for map_ret
map_ret()
{  // R1 now points to the value of xs
   info = R1[0];
   tag = info[-8];
   if (tag == 0) jump tag_0;
   // Extract the fields:
   x   = R1[4];
   xs' = R1[8];
   // Recover the free variables from stack
   f = Sp[4];
   // Code for (f x : map f xs) follows
   ...
 tag_0:
   // Load [] into R1, adjust stack, and return
   R1 = Nil_closure;
   Sp = Sp + 8;
   jump Sp[0];
```

```
}
```

The C-- procedure `map_entry` corresponds to the `map` function itself, and `map_ret` is the continuation generated for the `case` by the compiler. Note that `map_ret` is directly preceded by some data; this is its info table. Return addresses have info tables just as heap closures do; this makes stack frames have almost exactly the same layout as heap closures. The info table for a return address describes the layout of the stack frame for the garbage collector, just as the info table for a heap closure describes the layout of the closure.

The code of `map_ret` begins by scrutinising the value returned, in order to distinguish between the two cases. Each constructor in a datatype has a distinct *tag*, which is stored in the info table of the constructor. In the case of lists, `[]` has tag zero, and `(:)` has tag one.

`R1`, `R2`, and `R3`, like the stack pointer `Sp`, are "registers" of the STG-machine. They may be implemented by a real machine register, or mapped to a memory location: the choice is left to the back-end implementation. (In practice, since `R1` is used so often by the abstract machine, it is essential for good performance that it is mapped to a real machine register by the back-end).

In C-- a memory access always uses byte addressing, this differs from C, where the pointer type dictates the addressing granularity. In this paper, we assume that all code examples are for a machine with a word size of four bytes. This means that most of the memory accesses are done with offsets that are multiples of four, except for some of the examples in Section 6.

3. Benchmark methodology

Before embarking on the discussion of our optimisation techniques, we take a small detour to explain our benchmark methodology. This will enable us to present performance measurements for various techniques in the following sections.

We benchmark Haskell code using the `nofib` benchmark suite (Partain 1992), which contains 91 Haskell programs of various sizes, ranging from small micro-benchmarks (`tak`, `rfib`) to large programs solving "real" problems (e.g. LZH compression, hidden line removal, and a Prolog interpreter). We make no apology for including the micro-benchmarks: in practice even the larger programs often have small inner loops, and the micro-benchmarks are useful for highlighting extreme cases in sharp relief.

This paper contains several tables of results taken from runs of the `nofib` benchmark suite, e.g. Figure 13. In these tables we do not attempt to list every single result; rather we present a selection of representative programs, and aggregate minimum/maximum and geometric means *taken over the whole suite*.

For each benchmark, we can take a range of measurements:

- **Runtime**: wall-clock running time averaged over 10 runs on a quiet machine
- **Allocations**: the amount of memory allocated by the program over its lifetime
- **Code size**: the size of the binary in bytes

We made most of our measurements on two separate machines:

- An Intel Xeon, 2.4GHz, using the IA32 instruction set,
- An AMD Opteron 250, using the x86-64 instruction set with 64-bit pointers.

Program	Evaluated scrutinee (%)
anna	65.1
cacheprof	72.8
constraints	54.5
fulsom	41.5
integrate	67.6
mandel	73.9
simple	78.6
sphere	72.8
typecheck	56.5
wang	41.9
(81 more)	...
Min	0.20
Max	99.00
Average	61.86

Figure 2. Percentage of scrutinees already evaluated

Additionally, we have extended GHC with support for reading the CPU performance counters provided by many modern CPUs. These enable us to take accurate measurements of CPU events such as raw cycles and instructions executed, cache misses, branch mispredictions, and so on. We performed our CPU-counter measurements on the AMD Opteron system only.

We added hooks to the GHC runtime system that start and stop the CPU performance counters around the execution of Haskell code. We were thereby able to count events that occur during the execution of the Haskell code (the "mutator") separately from those that occur in the garbage collector, which is important since we're primarily interested in what is happening in the program itself. None of the techniques in this paper affect the amount of memory allocated by the program, and hence the amount of work the garbage collector has to do.

4. Measuring the dynamic behaviour of case expressions

The target for our optimisation is this:

> the code path taken when a `case` expression that scrutinises a single variable finds that the closure referenced by the variable is already evaluated.

For these cases, we can hope to generate a "fast path" that avoids indirect jumps (and the associated branch mispredictions), by using a more tag-ful approach.

Note that not all `case` expressions scrutinise a single variable; the STG language allows `case` expressions that scrutinise an arbitrary expression; for example:

```
case (f x) of
   []   -> ...
   y:ys -> ...
```

However, only `case` expressions that scrutinise a single variable can possibly take advantage of the fast path, so we restrict our attention to this subclass of `case` expressions. For the remainder of the paper, we will use the term "`case` expression" to mean "`case` expression that scrutinises a single variable".

In order to determine whether the optimisation is likely to be worthwhile, we measured the number of `case` expressions executed by each program in the `nofib` suite, and the proportion of those that found the closure to be already evaluated. Our results are given in Figure 2. The bottom line is that on average, 62% of closures being entered by a case expression are constructors that return directly

to the case continuation. This figure indicates that the already-evaluated scenario is common, and hence worthy of attention.

5. Scheme 1: semi-tagging

In this section we examine a straightforward scheme to avoid enter-and-return sequences in closure evaluation. This scheme produces code that does not enter closures that are already evaluated. The basic idea is to test the type of the closure before entering it: if the closure is a constructor, then we continue directly with the code to examine the constructor, otherwise we enter the closure as normal to evaluate it.

Version 1. The simplest modification to the code generator is to add this test directly before the enter:

```
map_entry()
{
   // Set up continuation as before:
   Sp[-4] = R2
   R1 = R3;
   Sp[-8] = map_ret;
   Sp = Sp - 8;

   // New code follows:
   // grab the info pointer from the closure
   info = R1[0];
   // grab the type field from the info table
   type = info[-12];
   // if it's a constructor, jump to the continuation
   if (type <= MAX_CONSTR_TYPE) jump map_ret;
   jump info;
}

data { word32[] = {...} }  // Info table for map_ret
map_ret()
{
   // exactly as before:
   info = R1[0];
   tag = info[-8];
   if (tag == 0) jump tag_0;
   ... code for (f x : map f xs) ...
 tag_0:
   ... code for [] ...
}
```

We read the info pointer, and then read the field of the info table that contains the type of the closure (remember that the info table is laid out backwards in memory directly before the info pointer, so this offset is negative). The type of the closure can then be examined, to determine whether it is a constructor; for various reasons several different object types all represent constructors, but we arrange that constructor types occupy the lowest-numbered values, so a single comparison suffices (here `MAX_CONSTR_TYPE` is the highest-valued constructor type). If the closure is a constructor, then we jump directly to the continuation. The continuation code extracts the tag from the constructor and performs comparisons to select the appropriate branch code.

Version 2. In the event that the closure was a constructor, the above scheme is slightly sub-optimal: we load the info pointer twice. An alternative scheme is as follows:

```
map_entry()
{
   Sp[-4] = R2
   R1 = R3;
   Sp[-8] = map_ret;
   Sp = Sp - 8;
   jump map_ret;
}

data { word32[] = {...} }  // Info table for map_ret
map_ret()
{
   info = R1[0];
   type = info[-12]
   if (type <= MAX_CONSTR_TYPE) jump info;
   tag = info[-8];
   if (tag == 0) jump tag_0;
   ... code for (f x : map f xs) ...
 tag_0:
   ... code for [] ...
}
```

All the comparisons are done in the continuation. This means an extra comparison in the event that we have to enter and return, but the code size is overall slightly smaller. We found this version to be marginally better.

Version 3. There is one further improvement that we could make:

```
map_entry()
{
   Sp[-4] = R2
   R1 = R3;
   Sp[-8] = map_ret;
   Sp = Sp - 8;
   jump map_cont;
}

map_cont()
{
   info = R1[0];
   type = info[-12]
   if (type <= MAX_CONSTR_TYPE) jump info;
   tag = info[-8];
   if (tag == 0) jump tag_0;
   ... code for (f x : map f xs) ...
 tag_0:
   ... code for [] ...
}

data { word32[] = {...} }  // Info table for map_ret
map_ret()
{
   jump map_cont;
}
```

Now `map_entry` can fall through directly to `map_cont`, because `map_cont` has no info table. This requires a fall-through optimisation that our back-end does not currently implement, so we have not tested this version. We expect it to be a marginal improvement, since in total it eliminates one (direct) jump in the already-evaluated case.

Implementing version 2 above required about 100 lines of modifications to the STG-to-C-- code generator inside GHC.

5.1 Results

In Figures 3 and 4 we show the effect of semi-tagging on code size and run-time, on the AMD Opteron and Intel Xeon respectively.

	With semi-tagging (Δ%)	
Program	Code size	Runtime
anna	+2.7	-5.3
cacheprof	+1.7	-10.4
constraints	+1.5	-11.6
fulsom	+2.5	+4.9
integrate	+1.6	-2.1
mandel	+1.7	-17.9
simple	+3.1	-9.8
sphere	+2.2	0.18
typecheck	+1.3	-18.1
wang	+1.8	+9.3
(81 more)	...	...
Min	+0.9	-33.7
Max	+3.1	+9.3
Geometric Mean	+1.5	-7.2

Figure 3. Semi-tagging performance (AMD Opteron, 64-bit)

	With semi-tagging (Δ%)	
Program	Code size	Runtime
anna	+3.9	-10.4
cacheprof	+2.4	-9.7
constraints	+2.1	-3.1
fulsom	+3.6	+1.3
integrate	+2.4	-13.0
mandel	+2.5	-17.0
simple	+4.8	-23.4
sphere	+3.2	-15.0
typecheck	+1.8	-17.6
wang	+2.6	-4.9
(81 more)	...	...
Min	+1.3	-37.2
Max	+4.8	+5.8
Geometric Mean	+2.2	-9.4

Figure 4. Semi-tagging performance (Intel P4, 32-bit)

	Baseline	With semi-tagging		
Program	Branch miss rate (%)	Branches executed (Δ%)	Branch misses (Δ%)	Branch miss rate (%)
anna	25.45	+20.5	-46.6	11.30
cacheprof	13.21	+10.7	-50.4	5.90
constraints	17.68	+17.6	-36.5	9.51
fulsom	23.29	+25.3	-21.4	14.61
integrate	18.20	+14.9	-35.9	10.14
mandel	22.50	+10.4	-47.0	10.80
simple	29.00	+13.3	-55.3	11.45
sphere	22.26	+13.2	-48.5	10.12
typecheck	25.90	+20.9	-45.1	11.74
wang	19.20	+35.6	-34.8	9.23
(81 more)	...	...	...	...
Min	0.20	+0.0	-95.0	0.20
Max	34.53	+39.1	+3.4	24.31
Average	17.43	+14.8	-46.3	8.80

Figure 5. Branch mispredictions (AMD Opteron, 64-bit)

We can see that while adding semi-tagging increases code size slightly, it has a dramatic impact on performance: 7.2% faster on the Opteron, and 9.4% faster on the Xeon.

We conjectured that the difference is due mainly to the fact that the Xeon has a longer pipeline, so branch mispredictions are more costly. To substantiate this guess, we used the CPU counters to measure the total number of branches executed, and the number of mispredicted branches during the execution of Haskell code (excluding the garbage collector). The results are shown in Figure 5.

The first column shows the branch misprediction rate for the baseline compiler; that is, what percentage of all branches are mispredicted, excluding the garbage collector. The next column, "Branches executed", shows the percentage increase in the total number of branch executed when moving from the baseline compiler to the semi-tagging compiler. As we would expect, semi-tagging increases the number of branches by around 15%, because of the extra conditional tests before entering a closure.

The next column "branch misses" shows the percentage increase in the total number of mispredicted branches. The effect is dramatic: even though the there are more branches in total, the total number of mispredictions in a program run is halved! The final column shows the branch misprediction rate of the semi-tagging compiler, which is also around half of the baseline value shown in the first column. (Note: the "Average" figures for the branch misprediction rates are *arithmetic* means, whereas those for the percentage increase in branch instructions and mispredictions are, of course, still *geometric* means.)

These results correlate well with the changes in performance that we saw above. For example, we can perform a back-of-an-envelope calculation for one of the example programs: `cryptarithm1` executed an average of 1.07G fewer cycles in the mutator with semi-tagging, and had an average of 83M fewer mispredicted branches. A mispredicted branch costs 10-12 cycles on this architecture (Fog 2006), so the difference in mispredicted branches accounts for at least 80% of the difference in cycles executed for this program. The results are similar for other programs.

5.2 Indirections

In the STG-machine, a suspended computation is represented by a *thunk*, which is a combination of an info pointer and the free variables of the suspended computation. When the thunk is evaluated, it overwrites itself with an *indirection* to the value, and return the value to the code that demanded it. Figure 6 gives a diagrammatic explanation of the process.

Indirections have a subtle effect on semi-tagging: a thunk may have been evaluated, but may still be represented by an indirection to the actual value. When a case expression examines the closure, it may find an indirection: in our simple semi-tagging scheme, the test for "is a constructor" will fail, and the case expression falls back to entering the closure. The entry code for an indirection simply returns the value it points to, but this sequence still incurs the enter-return penalty.

Is it common that a case expression encounters an indirection? The answer to this question depends on how often the garbage collector runs, because the garbage collector eliminates indirections, short-cutting them as part of its heap traversal. So the more often the garbage collector runs, the fewer indirections will be encountered during program execution.

We measured the number of times an indirection was encountered by a case expression as a percentage of the total number of case expressions executed, for two garbage collector settings. In both cases we used 2 generations, but the first set of results is with the default setting of a 0.5Mb nursery (young generation), and the second set is for a 32Mb nursery. The size of the nursery is more significant than the overall size of the heap, because each minor garbage collection will remove all the indirections in the nursery, so increasing the size of the nursery will directly increase the length of time that an indirection is visible. We know from previous experiments that most

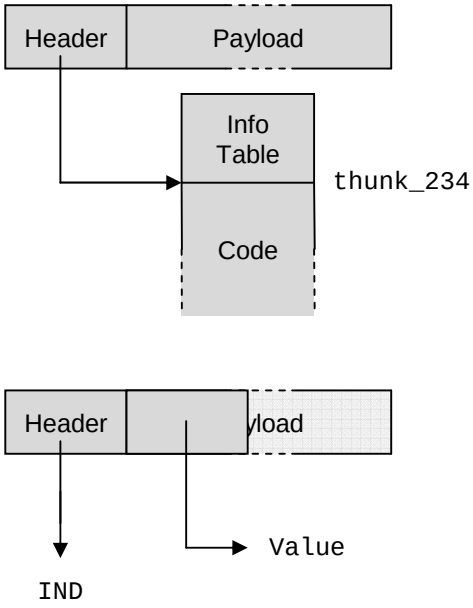

Figure 6. Update of a thunk

	Indirections/Cases (%)	
Program	0.5Mb	32Mb
anna	24.1	47.1
cacheprof	5.0	9.0
constraints	22.3	24.4
fulsom	13.3	21.1
integrate	4.0	4.4
mandel	11.9	16.9
simple	8.5	9.4
sphere	18.1	23.5
typecheck	27.1	29.2
wang	13.6	23.3
(81 more)	...	...
Min	0.00	0.00
Max	37.80	74.40
Average	10.43	18.04

Figure 7. Indirections encountered for different young generation sizes

thunks are updated soon after creation (Sansom and Jones 1993), and so most indirections will be in the nursery.

The results, in Figure 7, show that as the nursery size increases there is a definite increase in the number of indirections we encounter: on average 8% more when changing from the default heap setting of an 0.5Mb nursery to a 32Mb nursery. Some programs show a more dramatic difference: for example one of our micro-benchmarks, `wheel-sieve2` (not shown in the table), goes from 1.8% with a 0.5Mb nursery to 74% with a 32Mb nursery. For most programs, the difference is much less.

In Section 4, when we counted the number of times a case expression encountered an evaluated closure, we did not count indirections in the class of evaluated closures. So combining the results, we find that as we increase the nursery size from 0.5Mb to 32Mb, the number of evaluated closures that are encountered by a case expression falls by 8%, from 62% (measured in Section 4) to 54%. So although indirections seem to be highly significant in at least a few programs, it is still safe to say that on the whole a majority of case expressions directly scrutinise a constructor.

One could recover this lost 8% by instead treating an indirection as an evaluated closure, and adding an extra alternative in the case-analysis code that distinguishes between constructors; in the case of an indirection, we would simply follow the indirection and start the case analysis again. This would no doubt increase code size by a few percent and might give a modest improvement in performance. We have not tried this variant, although it would also be straightforward to make this a compile-time selectable option.

6. Scheme 2: pointer-tagging

In the semi-tagging scheme, we determined the evaluatedness and the tag of a closure by inspecting the closure's info table. Next we take this idea one step further, by encoding this information in *pointers to the closure*. The clear win from doing this is that we avoid dereferencing the info pointer and polluting the cache with the info table and, if the closure has no free variables (think of `True`, `False`, and `[]`, for example), then we can even avoid dereferencing the pointer at all.

How do we encode the type and tag in the pointer? The method we use is to take advantage of the unused least-significant bits of a pointer: pointers to closures are always aligned to the word size of the machine, so on a 32-bit architecture the two least-significant bits of a pointer are always zero, and on a 64-bit architecture there are three zero-bits. We want to encode two kinds of information in these bits:

- Whether or not the pointer is to a constructor.
- For constructors, the tag of the constructor.

First some terminology: we call a datatype *compact* if the number of constructors is three or fewer on a 32-bit platform, or seven or fewer on a 64-bit platform. The tag of a compact datatype will fit in the tag bits of a pointer.

For a compact datatype, the tag-bit encoding we use on a 32-bit architecture is:

unevaluated	evaluated constructor		
or unknown	tag 0	tag 1	tag 2
`00`	`01`	`10`	`11`

(extended in the obvious way for a 64-bit architecture). For a non-compact datatype, the encoding is:

unevaluated or unknown	evaluated
`00`	`01`

It is always safe to use zero for the tag bits, since that makes no commitment about the closure type. As we shall see, we cannot guarantee to tag *every* pointer to a constructor, so this approximate representation is necessary. In contrast, though, a non-zero tag must never lie, because we will compile code for a `case` expression that relies on its veracity.

We can only encode tags up to 3 on a 32-bit architecture, which leads to the following question: what fraction of case-analysis acts on data types with three or fewer constructors? To find out, we performed measurements across our benchmark suite of the percentage of case expressions executed that scrutinised datatypes of various family sizes. The results are given in Figure 8. As we can see

Family Size	Distribution (%)	Cumulative (%)
1	42.5	42.5
2	52.4	94.9
3	1.2	96.1
4	0.5	96.6
5	1.7	98.3
6	0.9	99.2
7	0.0	99.2
> 7	0.8	100.0

Figure 8. Constructors encountered in a case expression, classified by datatype family size

from the table, if we can encode tags up to 3, then we cover more than 95% of case expressions, and tags up to 7 gives us 99%. These results are unsurprising: we know that datatypes like lists, `Bool`, `Int`, and `Char` are very common, and these types all have just one or two constructors. However, it is good to have solid data to back up these intuitive claims.

Note that even if the datatype has more constructors than we can encode in the tag bits, we can still encode evaluatedness in the tag bits, which results in a cheaper test for evaluatedness than in the semi-tagging scheme.

We have implemented this dynamic pointer-tagging scheme in GHC. The following sections describe the full details of the implementation and give performance measurements.

6.1 Implementation

Compared to semi-tagging, pointer-tagging is a much more invasive change: it breaks a pervasive assumption, namely that a closure pointer can be dereferenced, and hence requires that we identify every place in the runtime system where closure pointers are dereferenced, and every place in the compiler that generates code to do so.

There are four parts to the implementation:

- Initialising tag bits for constructor pointers
- Using the tag bits in a case expression
- Clearing tag bits when dereferencing closure pointers
- Propagating tag bits: the garbage collector

We deal with these separately.

6.1.1 Initialising tag bits for constructor pointers

Pointer-tagging of dynamic constructors Dynamic constructors are allocated in the heap at run-time. With pointer-tagging, the expression `length [x]`[1]. compiles to

```
f_entry
{
  Hp[-8] = cons_info;
  Hp[-4] = x;
  Hp[ 0] = Nil_closure + 1;
  R2 = Hp - 8 + 2;
  jump length_info;
}
```

[1] `[x]` is Haskell's notation for a singleton list containing `x`, i.e. `x:[]`

`Hp` is the heap-allocation pointer, held in a global register. It points to the last occupied word of heap; the free space begins at the next lower address. We construct a cons cell starting at `Hp - 8` containing two fields: `x` and `Nil_closure` respectively. The `Nil_closure` symbol refers to `[]`: since this is a nullary constructor we need only a single instance of it at runtime, so GHC compiles the single instance of a nullary constructor statically into the module that defines the type. When we refer to a nullary constructor, we must of course tag the pointer; hence `Nil_closure+1`.

Now, when passing the address of the newly constructed cons cell to `length`, we should tag the pointer. This is done in the assignment to `R2`: the value `2` is the tag we are adding (one plus the tag of cons, which is one). Note how the tag assignment is merged with the heap offset, so no extra code is generated: this is a common pattern, as it turns out.

Pointer-tagging of static constructors A static constructor instance is one whose field values are known statically at compile-time. There is no code generated for these constructors, and they are not allocated in the heap; instead the constructor is generated at compile time and placed in the static data of the program. Referring to a static constructor is done by symbol name. For instance, the expression `length ['a']` is compiled into the following:

```
data lit1_closure {
  word32[] = { char_info, 97 }
}

data lit2_closure {
  word32[] = { cons_info, lit1_closure+1, Nil_closure+1 }
}

f_entry
{
  R2 = lit2_closure+2;
  ... adjust Sp ...
  jump length_info;
}
```

There are two static constructor instances here: `lit1_closure` represents the literal 'a', which is a boxed value of type `Char`, and `lit2_closure` is the value `['a']`, a single cons cell with `lit1_closure` as the head and `[]` as the tail. Just as in the dynamic case above, we ensure that the pointers to static constructors are tagged, by simple arithmetic on their addresses: we add 1 to the occurrence of `lit1_closure` (the and add 2 to the occurrence of `lit2_closure` (cons has tag 1).

If the static constructor binding is in a separately-compiled module, correctly tagging its references requires the compiler to propagate information about the constructor binding from the defining module to the usage site. GHC does not currently propagate this information between modules, so we are unable to tag references to static constructors in other modules. If, for example, `lit2_closure` was defined in a different module than `f_entry`, we would set `R2` to `lit2_closure` instead of `lit_closure+2`; but doing so is perfectly fine because a tag of zero is always safe.

6.1.2 Using the tag bits in a case expression

A case expression enters the closure for an inspected variable only if the tag bits are zero:

```
map_entry
{
   ...
```

```
   Sp[-4] = R2
   R1 = R3;
   Sp[-8] = map_ret;
   Sp = Sp - 8;

   // if it's a constructor, jump to the continuation
   if (R1 & TAG_MASK != 0) jump map_ret;
   jump R1[0];
}
```

The continuation code varies depending on the datatype size. If the datatype is compact, then the code can test and branch using the tag in the pointer:

```
data { word32[] = { ... } }
map_ret
{
   tag = R1 & TAG_MASK;
   if (tag == 1) jump nil_alt;
     // else, tag == 2
     // Code for (x:xs') alternative
     // Extract the fields and free vars
     x   = R1[4 - 2];
     xs' = R1[8 - 2];
     f   = Sp[4];
     ... code for (f x : map f xs') ...

 nil_alt:
   R1 = Nil_closure + 1;
   Sp = Sp + 8;
   jump Sp[0];
}
```

This code makes use of an invariant about return continuations, namely that *the pointer returned in R1 is always tagged*. It is not difficult to ensure this invariant always holds: pointers to freshly-built constructors are always tagged (Section 6.1.1), and when returning an existing constructor we simply tag the pointer before returning (the code after `nil_alt` is a good example).

When unpacking the fields of the constructor in a case alternative, we must remember to account for the tag bits in the pointer. For example, in the cons alternative above, we fetch `x` and `xs'` using offsets that take account of the tag bits now known to be in `R1`. As in the case of allocation, this adjustment costs nothing — we simply adjust the field offset.

It would be possible to apply the fall-through optimisation described in Version 3 of Section 5, although currently we have not implemented this.

For a non-compact datatype the case continuation looks like the semi-tagging one. However the info table address must be extracted with care because now the closure pointer is tagged. Remember that non-compact datatype closures are tagged with one, so a simple offset adjustment suffices:

```
data { word32[] = { ... } }
map_ret
{
   info = R1[-1];
   tag = info[-8];
  ... do branch selection using tag ...
}
```

6.1.3 Dereferencing closure pointers

Accessing the closure from a closure pointer requires clearing any tag bits in the pointer; we also call this operation "untagging". In the runtime, this is done with a macro:

```
  ... ptr points to closure ...
  value = UNTAG(ptr)[offset];
```

the macro `UNTAG(ptr)` clears the tag bits of the pointer using bit-wise conjunction: `ptr & ~TAG_MASK`. There were relatively few places where we had to add untagging in the runtime:

- the garbage collector,
- a macro `ENTER()` for evaluating arbitrary closures,
- the entry code for indirections,
- pre-compiled code for some common closure types,
- the external API for inspecting heap objects,
- the heap profiling subsystem,
- debugging and sanity checking code.

In the code generator, untagging of a pointer is necessary only in two places: the code for a case expression, which we have already covered, and when entering a variable in a tail-call position. For an example of the latter case, consider the expression

```
case x of
  (a,b) -> a
```

Here we need to generate code to tail-call `a`. The simplest scheme is to untag and enter `a`. There are two ways in which this scheme can be optimised:

- If the type of `a` tells us that it is a function, then we have no untagging to do, but we still have to enter the closure. We say more about tagging pointers to functions in Section 6.3.
- Instead of untagging and entering, we could test the tag bits, and if the closure is evaluated then we could return it immediately, and otherwise enter it as normal. This can be thought of as a variant of the tagged case expression, where we are evaluating a variable but don't statically know the address of the continuation. We tried this variant, and found it to result in a small code size increase (0.3%) with no measurable performance benefit, so we rejected this option in favour of the simpler untag and enter sequence for now.

6.1.4 Propagating tag bits: the garbage collector

The garbage collector has a key role to play in the pointer-tagging scheme. It is the garbage collector's job to remove indirections in the heap, and as part of doing so it turns an untagged pointer (the pointer to the indirection) into a tagged pointer which points directly to the constructor. The more often the GC runs, the fewer indirections there will be (as we saw in Section 5.2) and the more tagged pointers will be encountered.

Is it possible that there can ever be an untagged pointer to a constructor with no intervening indirection? Whenever we create a constructor in the heap, the code generator guarantees to only refer to it using a tagged pointer, so it is never the case that a pointer to a constructor on the heap can be untagged. However, this leaves one way in which untagged pointers to constructors can arise:

Program	Tagged/Evaluated (%)
anna	98.4
cacheprof	99.6
constraints	99.1
fulsom	93.2
integrate	76.6
mandel	80.3
simple	90.6
sphere	94.5
typecheck	100.0
wang	100.0
(81 more)	...
Min	73.30
Max	100.00
Average	97.34

Figure 9. Percentage of evaluated scrutinees that were tagged

	With semi-tagging (Δ%)	
Program	Code size	Runtime
anna	+3.8	-17.3
cacheprof	+2.6	-18.0
constraints	+2.4	-14.4
fulsom	+3.5	-3.1
integrate	+2.6	-8.7
mandel	+2.7	-24.8
simple	+3.6	-22.6
sphere	+3.0	0.14
typecheck	+2.2	-23.8
wang	+2.7	+2.1
(81 more)	...	...
Min	+1.4	-37.3
Max	+4.0	+10.6
Geometric Mean	+2.4	-13.7

Figure 10. Pointer-tagging performance (AMD Opteron, 64-bit)

we don't guarantee to tag pointers to *static* constructors (see Section 6.1.1). We have measured how often a case expression encounters a pointer to a constructor that isn't tagged. The results are given in Figure 9; on average, 97% of pointers to constructors were properly tagged.

The garbage collector could add the tag bits to pointers to static constructors, which would have the effect of making it even more rare to encounter an untagged pointer to a constructor, but in the version of the system we measured for this paper it currently doesn't.

Static constructors are most commonly encountered in the form of *dictionaries*, which are part of the encoding of Haskell's typeclass overloading, so programs that make heavy use of overloading would be more likely to suffer from untagged pointers.

6.2 Results

Pointer-tagging is more efficient than the semi-tagging scheme. From Figures 10 and 11 we can see that the runtimes have been reduced by 13.7% and 12.8% on the AMD Opteron and Intel Xeon respectively, in exchange for an increase in binary size of a little over 2%.

Compared to semi-tagging, our pointer-tagging implementation is winning by 5.5% on the AMD and 3.4% on the Intel, with code size increased by less than 1%.

Pointer-tagging prevents roughly the same amount of enter-and-return sequences that semi-tagging does. So we do not expect a

	With semi-tagging (Δ%)	
Program	Code size	Runtime
anna	+4.2	-15.0
cacheprof	+2.9	-14.9
constraints	+2.6	-4.7
fulsom	+3.9	-5.9
integrate	+2.9	-16.7
mandel	+2.9	-19.9
simple	+4.4	-26.9
sphere	+3.4	-22.8
typecheck	+2.4	-31.4
wang	+3.0	-11.1
(81 more)	...	...
Min	+1.5	-44.4
Max	+4.4	+11.4
Geometric Mean	+2.6	-12.8

Figure 11. Pointer-tagging performance (Intel P4, 32-bit)

	L1 D-cache		L2 D-cache
Program	accesses	misses	misses
anna	-23.4	-13.3	-28.4
cacheprof	-15.9	-5.8	-8.1
constraints	-12.8	-4.8	-5.4
fulsom	-8.9	-17.2	-53.3
integrate	-7.7	-2.2	+18.6
mandel	-8.4	-1.9	-29.4
simple	-11.8	-3.9	-4.1
sphere	-13.0	-19.5	-73.2
typecheck	-20.4	-8.5	-30.8
wang	-13.4	-2.0	-10.4
(81 more)	...	...	
Min	-31.5	-19.5	-73.2
Max	+0.6	+0.5	+81.6
Geometric Mean	-13.9	-4.5	-20.1

Figure 12. Cache behaviour for pointer-tagging vs. the baseline

change in the number of mispredicted branches from tagging pointers. Our measurements confirm this, the rate of branch mispredictions is similar to those obtained from the semi-tagging scheme. For this reason we have omitted branch-prediction measurements.

The enhanced performance of pointer-tagging over semi-tagging comes from reduced cache activity. In most case expressions branch selection can be determined from the pointer tag alone. This is in contrast with the semi-tagging situation, where the info table is loaded into the cache for the sole purpose of extracting the constructor tag. We can see how pointer-tagging affects the cache behaviour of the program in Figure 12. Accesses to both levels of cache and accesses to main memory all decrease, on average.

6.3 Should we tag pointers to functions?

Thus far our attention has focused on pointers to *constructors*, but any functional language has another important class of values, namely *functions*. If we tag constructor-value pointers to indicate their evaluatedness (and perhapas tag), could we also tag function-value pointers to indicate their evaluatedness (and perhaps more besides)?

GHC already uses an eval/apply evaluation model for function application (Marlow and Peyton Jones 2004), which behaves rather like semi-tagging. When an unknown function is applied, the function and its arguments are passed to a pre-compiled code sequence in the runtime system. This "generic apply" code sequence first checks to see whether the function closure is evaluated. If not, it

evaluates the thunk. If so, it checks whether the arity of the function (held in its info table) is correct for making the call.

Little would be gained from encoding just the evaluatedness in the function-value pointer. But if we could encode the arity of the function as well, then *we could avoid ever accessing the info table*, which would improve the cache behaviour (c.f. Section 6.2). The tag bits in a function pointer would therefore have the following meanings: 0 means unevaluated or evaluated with unknown arity, and 1–3 (or 1–7 on a 64-bit machine) indicates an evaluated function with the given arity. Calls to evaluated functions with the correct arity represent the majority (over 90%) of calls to unknown functions, and over 99% of calls to unknown functions pass 3 or fewer arguments (Marlow and Peyton Jones 2004), so we have enough tag bits to cover the majority of cases.

We could go further. Rather than always making an out-of-line call to the generic apply sequence for an unknown call, we could compile an inline test of the tag bits: if the tag bits indicate an evaluated function with the correct arity, then jump immediately to the function, otherwise fall back to the generic apply. This would avoid a jump in the common case that the call is to a function with the correct arity, in exchange for a little extra code.

There might appear to be a difficulty in tagging pointers to static function closures in separate modules, in the same way that we had difficulty with tagging pointers to static constructors in other modules (Section 6.1). However, GHC already passes information about the arity of functions between compilation units to facilitate making fast calls to statically-known functions, and we could use this same information to correctly tag pointers to static function closures[2].

We have a preliminary implementation of the scheme described above, and initial measurements indicate that it improves performance by a modest 1-2% over pointer-tagging. We plan to further investigate this scheme in future work.

7. Constructor returns

So far we have focused attention exclusively on *entering* the scrutinee of a `case` expression. But if the scrutinee is entered, it will take a *second* indirect jump when its evaluation is complete, to return to the case continuation.

In this section we discuss two issues relating to this latter control transfer.

7.1 Using call/return instructions

As we mentioned in Section 2, GHC generates code that manages the Haskell stack entirely separately from the system-supported C stack. As a result, a `case` expression must explicitly push a return address, or continuation, onto the Haskell stack; and the "return" takes the form of an indirect jump to this address. There is a lost opportunity here, because every processor has built-in `CALL` and `RET` instructions that help the branch-prediction hardware make good predictions: a `RET` instruction conveys much more information than an arbitrary indirect jump.

Nevertheless, for several tiresome reasons, GHC cannot readily make use of these instructions:

- The Haskell stack is allocated in the heap. GHC generates code to check for stack overflow, and relocates the stack if necessary. In this way GHC can support zillions of little stacks (one per thread), each of which may be only a few hundred bytes long. However, operating systems typically take signals on the user stack, and do no limit checking. It is often possible to arrange that signals are executed on a separate stack, however.
- The code for a case continuation is normally preceded by an info table that describes its stack frame layout. This arrangement is convenient because the stack frame looks just like a heap closure, which we described in Section 2. The garbage collector can now use the info table to distinguish the pointers from non-pointers in the stack frame closure. This changes if the scrutinee is evaluated using a `CALL` instruction: when the called procedure is done, it `RET`urns to the instruction right after the call. This means that the info table can no longer be placed before a continuation. Thus the possible benefits of a `CALL`/`RET` scheme must outweigh the performance penalty of abandoning the current (efficient) info table layout.

A tentative solution would be to use RET for constructor returns, but do JMP and PUSH rather than doing CALL. This turns out to be unsatisfactory: RET prediction uses a buffer internal to the processor, which is used to maintain return targets. CALL instructions push return addresses into this buffer, but this is not the case for JMP+PUSH. So this tentative solution will still suffer from mispredicted RET instructions.

In conclusion it is not easy to switch to `CALL`/`RET` instructions, and it is not clear that Haskell programs would run faster as a result.

	No vectored returns (Δ%)		
Program	Code size	Run time	mutator L2 misses
anna	-6.4	-12.1	-51.5
cacheprof	-3.7	-2.9	-37.9
constraints	-2.7	+8.9	+111.5
fulsom	-2.9	-6.8	-51.6
integrate	-2.7	-0.8	-8.4
mandel	-2.6	-3.2	-82.8
simple	-2.2	-1.6	-3.2
sphere	-2.6	-9.2	-42.3
typecheck	-2.7	-2.7	-57.4
wang	-2.8	-13.0	-1.2
(81 more)	...	...	...
Min	-6.4	-13.0	-89.0
Max	-0.7	+8.9	+111.5
Geometric Mean	-2.3	-1.5	-22.7

Figure 13. Turning off vectored returns (AMD Opteron)

7.2 Vectored returns

Fifteen years ago, the original paper about the STG machine described an attractive optimisation to the compilation of `case` expressions, called *vectored returns* (Peyton Jones 1992, Section 10.4). Consider a boolean `case` expression:

```
case x of
  True  -> e1
  False -> e2
```

As we have described it so far, we push a return address and enter `x`; the code at the return address tests the (tag of the) returned value, to choose between `e1` and `e2`. But suppose instead we pushed *two* return addresses or, rather, a pointer to a vector of two return addresses. Now the constructor `True` could return directly to the first, while the constructor `False` could return directly to the second.

[2] Note that the type of a function doesn't convey its arity: for example, a function with arity 1 might return a function of arity 2, but its type would suggest an arity of 3.

The return still consists of a single indirect branch, but no test need be performed on return, so *there is a net saving of one conditional branch*.

GHC has embodied this idea for a decade. The "return address" on the stack still points to an info table (so that the garbage collector can walk the stack), and the vector simply forms part of the info table. The choice between direct and vectored return is made on a type-by-type basis. For data types with many constructors, the vector table would be large and might contain many duplicate entries, so GHC uses a threshold scheme: vectored returns were used for data types with 8 constructors or fewer.

The picture changes somewhat with our tagging schemes. Now `x` is tested before being entered, and if it is evaluated the appropriate alternative is selected. In these cases the use of a vectored return is irrelevant, so its benefits (if any) would be reduced. This observation provoked us to re-evaluate the effectiveness of the whole vectored-return idea.

In principle, vectored returns still sound like a win: the total number of instructions executed should be reduced by vectored returns, while the number of indirect branches should be unchanged. However, vectored returns impose several less obvious costs, on both code size and run time:

Run time. Vectored returns are not cache-friendly. A vectored return makes a *data* access to the info table of a return continuation. During normal execution (garbage collection aside), the info table of a return continuation would never otherwise enter the data cache. These vector-table accesses therefore increase the data-cache load, and this effect turns out to be significant. Figure 13 shows the results of our measurements of mutator-only level-2 data cache miss rates for our benchmark programs. Switching off vectored returns improves the miss-rate by 20%!

Code size. On 64-bit processors, each table entry is 8 bytes, and the corresponding test-and-branch code is typically smaller than the size of the vector. Even on a 32-bit machine this can sometimes be the case.

Code size. The info table for a return address has an optional field used to store information about static references for the garbage collector. In the absence of a vector table this field can often be omitted, but if the vector table is present then this optional field must always be present so that the vector table is in a predictable location. This causes some extra code-size overhead for vectored returns.

Code size, run time. If we want to generate position-independent code, which we do, the vector has to contain *offsets* relative to the vector table rather than direct pointers to the alternatives. This increases the instruction count for a vectored return (currently 4 instructions on x86 processors, compared to one instruction for a direct return).

Complexity. Vectored returns significantly complicate some other aspects of the compiler. In particular, certain stack frames (e.g. the code for polymorphic `seq`) need to be able to handle *any* return convention, vectored or otherwise. These standard stack frames therefore need a fully-populated return vector in addition to the direct-return code.

When we made our measurements we found, to our surprise, that even *before* adding semi-tagging vectored returns had a net negative impact on performance! (The previous measurements, made some years ago, showed a net benefit of around 5%.) Figure 13 shows a comparison between the performance of GHC-compiled programs with vectored returns for types with up to 8 constructors (the default), and the same programs compiled with a fixed direct-return convention. Turning off vectored returns both reduces code size and improves performance marginally.

These results are for a 64-bit machine using 8-byte vector table entries. On a 32-bit machine the code size reduction is -1.3% while the running time difference is -1.7%.

Our conclusion is simple: vectored returns carry no net benefit, and can safely be retired, with welcome savings in the complexity budget of both compiler and runtime system.

All the other measurements in this paper are relative to the highest-performing baseline available, the one with vectored returns switched off. Thus, all the gains reported in this paper are genuine gains due to tagging alone.

8. Related work

Many programming language implementations (SML/NJ, Ocaml, SmallTalk) use tag bits to differentiate pointers from non-pointers. The garbage collector uses the tags to find the pointers when it is marking/copying the live heap. Although it uses the same low-order-bits encoding, this technique is almost unrelated to ours. Our tag bits never indicate a pointer/non-pointer distinction, nor do we suffer from the loss of some bits of integer precision. Furthermore, our pointer-tagging scheme describes something about the object pointed to, *even though the latter may change dynamically*. That is why the zero tag always means "unknonwn"; and it is why the garbage collector cooperates to propagate tags from objects into the pointers that reference them.

CMUCL uses a 3-bit tagging scheme to distinguish various types of pointer (MacLachlan 2003). The tagging scheme is global (as opposed to type-dependent like ours). Scheme 48 also uses a 2-bit type tag on the pointer (Kelsey and Rees 1994). Neither of these systems need to support lazy evaluation, so there is no need for tagging closure pointers with laziness information. However pointer tags could, for example, encode whether a `cons` or `nil` object is being pointed-to, just like the dynamic pointer-tagging scheme does. Maybe this encoding is not used because the available tag-bits are already reserved for dynamic type information. The lack of static type information leaves little room for CMUCL and Scheme 48 to encode more information in the pointer tag.

A possible variant on pointer-tagging would be to divide memory into segments where closures with the same constructor tag reside. In this way high bits of pointers can be used as constructor tags, and this frees up the lowest bits to encode other semantic information. The Big Bag of Pages (BiBoP) (Steele 1977) gives this scheme additional flexibility: the tag bits do not directly encode closure information, but rather, they index into a table that provides it. Both of these variants introduce complexities into the storage manager and code generation however, because now constructors must be allocated in different heaps, using different heap pointers.

The closest directly-related work is an unpublished paper by Hammond (Hammond 1993), which describes several tagging schemes and measures the effectiveness of each using hand-written code to simulate the output of a compiler. The main focus of Hammond's paper is a scheme he calls "semi-tagging"[3], in which the least-significant bits of the *info pointer* in a closure are used to indicate evaluated vs. unevaluated closures. This doesn't correspond exactly to either our semi-tagging or dynamic-pointer-tagging schemes, but there are reasons to believe that it would fall between the two, both in terms of complexity and performance:

[3] We took the liberty of re-using the name

- It is likely to be slightly more efficient to check the tag bits on the info pointer, as per Hammond's semi-tagging, than to inspect the info table as in our semi-tagging, since reading the info table is an extra memory operation (albeit one which is likely to be cached, because we only read the info tables of constructors and there are relatively few of those).
- It is not likely to be as efficient as dynamic-pointer-tagging, because loading the info pointer to check the tag bits may mean an extra memory operation.
- Tagging the least-significant bits of the info pointer adds non-localised complexity to the implementation, but perhaps not as much as dynamic pointer tagging, because the garbage collector would not need to propagate tag bits.

In the GRIN intermediate language (Boquist 1999) unevaluated closures are encoded by constructors added by the compilation process. Every thunk and function is represented by a constructor (with arguments if encoding an applied function). The code for forcing an unevaluated closure cannot just enter the closure because now it has no code associated with it, instead it can match the compiler-introduced constructor and jump to the thunk or function that is represented by it. This approach is somewhat tag-ful because it inspects the closure content for evaluation, and like dynamic pointer-tagging it avoids the indirect jump incurred by a tag-less approach.

Nethercote and Mycroft (Nethercote and Mycroft 2003) were the first to do low level measurements of GHC-compiled Haskell programs. They used the CPU counters from the the AMD Athlon processor to collect cache miss and branch prediction information. Their work states that branch misprediction stalls account up to 32% of execution time. Nethercote and Mycroft attributed these stalls to the indirect jumps that implement laziness in compiled code, and they suggested tagging pointers with an "evaluatedness" bit. Our work takes this idea a step further: the tag not only indicates "evaluatedness" but also caches information about the pointed-to closure, such as constructor tags and function arities.

9. Conclusion

Our conclusion is that it is time to take out "tagless" out of the Spineless Tagless G-Machine (STG-machine). Our performance figures before and after the optimisations confirm that the uniform "enter-to-evaluate" strategy is at the source of half of the branch misprediction events.

To solve this problem, we have proposed two schemes that do a more tag-ful evaluation of closures: semi-tagging and dynamic pointer tagging. Dynamic pointer-tagging wins on pure performance: a 13% performance imporovement compared to semi-tagging's 9% on the 32-bit processor (14% / 7% on the 64-bit processor). However, semi-tagging wins comprehensively if we consider complexity: the changes to the compiler were small and localised, whereas dynamic pointer-tagging changes the fundamental representation of pointers and hence requires non-localised changes to the compiler and runtime.

On balance we plan to adopt dynamic pointer-tagging in future versions of GHC. Although the changes are non-localised, they are still relatively small: approximately 600 lines to the whole system.

Acknowledgements

The second author was partially supported by Microsoft Research Cambridge for this work. Thanks are due to Anoop Iyer for his clarifications on branch prediction and to John van Schie for his modified version of Valgrind for experiments.

References

Urban Boquist. *Code Optimisation Techniques for Lazy Functional Languages*. PhD thesis, Chalmers University of Technology, April 1999. URL `http://www.cs.chalmers.se/~boquist/phd/phd.ps`.

Agner Fog. The microarchitecture of Intel and AMD CPUs: An optimization guide for assembly programmers and compiler makers. online manual, 2006. `http://www.agner.org/optimize/microarchitecture.pdf`.

K. Hammond. The spineless tagless G-machine — NOT. unpublished, 1993. URL `citeseer.ist.psu.edu/hammond93spineless.html`.

Richard A. Kelsey and Jonathan A. Rees. A tractable scheme implementation. *Lisp and Symbolic Computation*, (7(4)):315–335, 1994. URL `http://repository.readscheme.org/ftp/papers/vlisp-lasc/scheme48.ps.gz`.

Robert A. MacLachlan. Design of CMU Common Lisp. online manual, 2003. `http://common-lisp.net/project/cmucl/doc/CMUCL-design.pdf`.

Simon Marlow and Simon Peyton Jones. Making a fast curry: Push/enter vs. eval/apply for higher-order languages. In *ACM SIGPLAN International Conference on Functional Programming (ICFP'04)*, pages 4–15, Snowbird, Utah, September 2004. ACM.

Nicholas Nethercote and Alan Mycroft. Redux: A dynamic dataflow tracer. *Electr. Notes Theor. Comput. Sci.*, 89(2), 2003.

WD Partain. The `nofib` benchmark suite of Haskell programs. In J Launchbury and PM Sansom, editors, *Functional Programming, Glasgow 1992*, pages 195–202. 1992.

Simon Peyton Jones, Norman Ramsey, and Fermin Reig. `C--`: a portable assembly language that supports garbage collection. In G Nadathur, editor, *International Conference on Principles and Practice of Declarative Programming*, number 1702, pages 1–28, Berlin, September 1999. Springer.

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

Patrick M. Sansom and Simon L. Peyton Jones. Generational garbage collection for haskell. In *Functional Programming Languages and Computer Architecture*, pages 106–116, 1993. URL `citeseer.ist.psu.edu/sansom93generational.html`.

Guy Lewis Steele. Data representation in PDP-10 MACLISP. Technical Report AI Lab Memo AIM-420, MIT AI Lab, 1977.