An Efficient Representation for Lazy Constructors using 64-bit Pointers

Georgios Fourtounis   Nikolaos Papaspyrou

School of Electrical and Computer Engineering
National Technical University of Athens, Greece
{gfour,nickie}@softlab.ntua.gr

Abstract

Pointers in the AMD64 architecture contain unused space, a feature often exploited by modern programming language implementations. We use this property in a defunctionalizing compiler for a subset of Haskell, generating fast programs having a compact memory representation of their runtime structures. We demonstrate that, in most cases, the compact representation is faster, uses less memory and has better cache characteristics. Our prototype shows competitive performance when compared to GHC with full optimizations on.

Categories and Subject Descriptors D.3.4 [*Processors*]: Code generation, Run-time environments; D.3.2 [*Language Classifications*]: Applicative (functional) languages

Keywords lazy functional programming; tagged pointers; AMD64; compact data representations

1. Introduction

Programs written in high-level languages are frequently praised for expressiveness and independence from hardware details. However, this comes at a price: hiding low-level details from the programmer means that language implementations are responsible for obtaining a good performance.

This is especially true for Haskell [28], a non-strict functional programming language, whose semantics is very different from the principles of today's mainstream hardware. Although Haskell's execution model has been adapted to run on such hardware [27, 34], there is still room for improvement, as hardware evolves and different issues dominate performance.

A significant problem in modern hardware is the processor-memory performance gap [11, 48]. Processors are too fast compared to memory, making programs stall while waiting for memory to respond to read/write accesses. An attempt to solve this disparity is caches, which are fast intermediate memories that sit between the processor and memory and try to minimize latencies by keeping recently-used data. Caches however are small in size and therefore programs must have good cache locality, a factor connected to code and data size [31, 49]. The memory gap puts special pressure on Haskell implementations, since the language is based on immutable data structures and programs spend a lot of time in memory allocation and garbage collection, exhibiting mediocre cache behaviour [38].

We present an implementation for a subset of Haskell that uses a compact memory representation, taking advantage of redundancy in pointers in AMD64 [36], currently one of the most widespread computer architectures. Our representation tries to minimize memory use by clustering runtime structures [17], trading extra memory usage for bit-field manipulations.

In the rest of this paper, we first describe the AMD64 features that we use together with our runtime model (Section 2). Then we show how we used these features in our implementation (Section 3) and discuss how garbage collection cooperates with our compact memory representation (Section 4). To evaluate our technique, we benchmark our prototype compiler against our previous (less compact) implementation and also against a fully optimizing GHC, the de facto standard Haskell compiler (Section 5). Finally, we examine related implementation techniques (Section 6) and conclude with a few remarks and directions for future work (Section 7).

2. Background

We present here the details of the AMD64 architecture that we use in our implementation, as well as our runtime model.

2.1 Redundancy in AMD64 Pointers

Pointers in the AMD64 architecture have the following properties:

1. They are aligned at 8 bytes [36, p. 12], their last 3 bits always being zero.
2. They may only be used to handle 48-bit addresses [36, Section 3.3.2]. The high 16 bits are assumed to be equal to the most significant used bit [4, Section 1.1.3].

Every pointer then uses only 45 significant bits (termed from now on the *pointer body*), leaving 19 bits per pointer (or 29.69%) that carry no information and can be reused. The space occupied by a pointer can now be seen as in Figure 1, where the shaded space has known contents and is therefore free to reuse, as its contents can be reconstructed on demand.

Accessing different fields of this space can be done with bitwise operations, such as logical shifts and boolean operations. For example, if bits 48-63 are assumed to contain a 16-bit unsigned integer (its type in C being `uint16_t` [45]), then the following C macro would take a pointer `p` and apply a logical shift operation to give back the integer value:

```
#define INT16(p) ((uint16_t)((uintptr_t) p >> 48))
```

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

FHPC '14, September 4, 2014, Gothenburg, Sweeden.
Copyright © 2014 ACM 978-1-4503-3040-4/14/09...$15.00.
http://dx.doi.org/10.1145/2636228.2636232

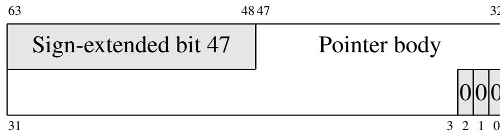

Figure 1. Space in a pointer in the AMD64 architecture.

In the rest of this paper, we will use C code like the above to represent such operations, in the style of Gudeman [25], assuming that this code is compiled by a C compiler for the AMD64 architecture. We will also use the standard C types `uintptr_t` and `intptr_t` [45] to view pointers as unsigned or signed 64-bit integers and do unsigned or sign-extended shifts correspondingly.

2.2 Lazy Activation Records

Our work is based on GIC, a prototype compiler from a subset of Haskell to C [21]. Our compiler is based on defunctionalization [43] from higher-order Haskell to a first-order subset of Haskell. Defunctionalization replaces all closures with constructors, for example the following higher-order program:

```
result     = double (add 2) 10
double f x = f (f x)
add    a b = a + b
```

is transformed internally to the following first-order program:

```
data Closure = Add Int

result       = double (Add 2) 10
double f x   = apply f (apply f x)
add    a b   = a + b

apply  c z   = case c of
                 Add a0 -> add a0 z
```

The first-order lazy language is implemented using *lazy activation records (LARs)* for function and constructor calls [21]. These are activation records that hold the following information:

- *prev*: The access link [2] to the lazy activation record of the calling function.
- *vals*: The parameters passed to the function or constructor, represented as unevaluated thunks.
- *nested*: A display structure [2] containing pointers to the contents of visible constructors. This is used by pattern matching inside functions, to lazily evaluate constructor components.

For example, assume the following function `f`, that takes two parameters and contains one pattern-matching expression:

```
f x y = case x of
          []   -> [1]
          a:as -> [a + y]
```

A LAR allocated for a call to `f` will contain the access link pointer, two thunks and one nested entry, as pictured in Figure 2.

The thunk is the basic machinery of lazy evaluation, wrapping code that should only be evaluated when needed (and whose value must be remembered, in case it is needed again). A thunk contains three fields:

- *flag*: A flag that shows if the thunk has been evaluated.
- *code*: A pointer to the code that evaluates the thunk.
- *value*: A memoized value, if the thunk has been evaluated.

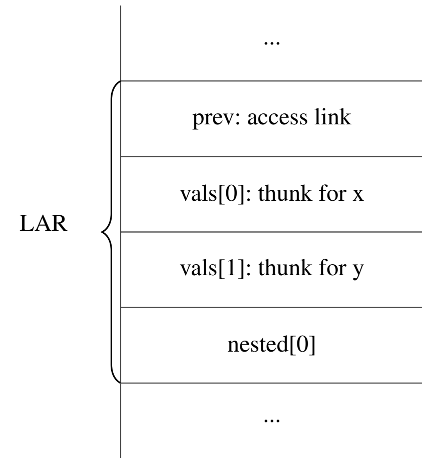

Figure 2. A lazy activation record (LAR) of a function taking two parameters and doing one level of pattern-matching.

The memoized value is a lazy constructor, which has a head tag (e.g., `Cons` or `Nil` for a classic list data type) and a pointer to a list of lazy arguments. In our implementation, a constructor is simply a function that returns a pair of a constructor tag and a pointer to the LAR containing its own arguments.

Note. For simplicity, in the rest of this paper we will use "function" to refer to both user-defined functions and constructor functions and "LAR" for lazy activation records created for function and constructor calls.

3. Single-Word Thunks

As described in §2.2, a thunk should have three fields: (a) a flag to indicate if it has been evaluated, (b) a code pointer, and (c) a memoized value. In this section we describe how to fit all of them in a single machine word.

We first observe that the code pointer is useful only when the thunk has not been evaluated yet; the opposite is true for the memoized value, which exists only when the thunk has already been evaluated. Therefore, we can reuse the same machine word for both the code pointer and the memoized value, if we keep the last word bit as the thunk flag that determines the state of the thunk.

The code pointer fits in one machine word by definition and its three low bits are assumed to be zero.[1] The memoized value is a lazy constructor, i.e., a pair of a constructor tag and a LAR pointer. If we assume that each data type has at most 2^{16} different constructors, we can embed the tag in the unused 16 high bits of the LAR pointer, representing the lazy constructor using just the machine word of the pointer. Since we use the space of a pointer, the three low bits will also be zero. As the three low bits have known content in both cases (code pointer and memoized value), the last bit can then be reused as the thunk flag.

We now have a compact representation but we still have two spare bits (1-2). We exploit this and use bit 1 as a *value flag* to differentiate between normal constructors and enumeration constructors (such as `True` or `42`); the latter need no pointer to arguments and can then use the unused space of the pointer body as well. We call these values *primitive* and use them to represent 62-bit integers, booleans, floating-point numbers, and other enumerations. Because Haskell is statically typed, the value flag should not be necessary: all operations are always applied to suitable values during runtime.

[1] This property can be forced, e.g. in GCC, by using the option `-falign-functions`.

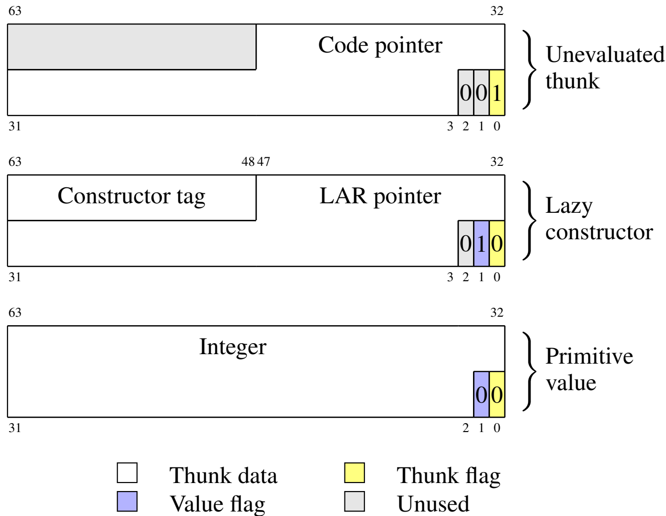

Figure 3. Space in a 64-bit thunk.

However we need the flag for the garbage collector described in the following section, which has to examine thunks to find pointers.

Figure 3 shows the layout of unevaluated and evaluated thunks; the latter case is both for lazy constructors and primitive values. There is still some unused space (bit 2 in constructor values and bits 1-2 in unevaluated thunks). We also reserve the pattern `110` of the last three bits for future use.

Handling unevaluated thunks. The following macro can check if a thunk has already been evaluated, by testing the thunk flag:

```
#define IS_VAL(t) (((uintptr_t) t & 1) == 0)
```

If the flag is set, then the `CODE` macro can be used to mask out the flag and give back the code pointer that must be called to evaluate the thunk:

```
#define CODE(t) ((uintptr_t) t & ~1)
```

If the thunk flag is cleared, then bits 2-63 contain an already computed value, which is either a lazy constructor or a primitive value, and the macros of the following paragraphs can be used to access its contents. As explained before, the type of the value is known statically, without testing bit 1.

Lazy constructors. A lazy constructor contains two fields: a constructor tag and a pointer to the LAR containing the constructor components as thunks. The constructor tag can be accessed by simple shifting:

```
#define CONSTR(p) ((uintptr_t) p >> 48)
```

The constructor pointer must be reconstructed from bits 3-47. As shown in Figure 1, to reconstruct a pointer from its pointer body, the three low bits must be cleared and the highest-bit in the pointer body must be sign-extended to the left. The following `CPTR` macro shows this operation, where the `(intptr_t)` cast makes the right shift signed:

```
#define CPTR(p) (((intptr_t) p << 16 >> 16) & ~7)
```

Primitive values. These are 62-bit signed integers, residing in bits 2-63 and can be read and written by the following macros:

```
#define PVAL_R(p) ((intptr_t) p >> 2)
#define PVAL_W(i) ((intptr_t) i << 2)
```

The cost of this representation is that for every built-in operation, each argument must be shifted 2 bits to the right, the operation performed, and the computed value must be shifted 2 bits to the left again, to be a valid thunk. However, we can do better for some operations, since the lower bits are always 0, for example, two positive integers can be added without shifting as follows [25]:

```
#define SUM(p1,p2) ((intptr_t) p1 + (intptr_t) p2)
```

Why use this layout of the low bits? As shown in the previous paragraph, the representation of primitive integers has an advantage, since the low bits are 0 and some operations can be simplified. Another choice would be to have the low bits of constructors be 0, so that dereferencing the constructor LAR pointer would not have to use the `~7` mask. We could also have the three low bits be 0 for unevaluated thunks, so that `CODE` would not need the `~1` mask. In practice, we did not see any performance difference between these three representations.

4. Garbage Collection

Although our code generator produces C, we cannot directly use a general-purpose conservative collector, such as Boehm-Demers-Weiser [10], because the information embedded in pointers masks them beyond recognition.[2] We need a garbage collector that understands the representation of thunks and lazy activation records. In this section we will describe the basic interface that our representation presents to a custom semi-space garbage collector [26].

There is only one way to allocate memory in our implementation: creating a LAR for a function or constructor call. Our garbage collector is then simple: the roots are pointers to lazy activation records found in registers or in the stack; collection proceeds recursively to all lazy activation records reachable from them. Pointers to other LARs can be found (a) in the *prev* link, (b) in evaluated thunks containing constructors, and (c) in the *nested* fields.

As described in §2.2, lazy activation records of different functions have different layouts: the activation record of a function taking $a \geq 0$ arguments and containing $n \geq 0$ levels of nested pattern matching clauses has a thunks and n *nested* fields. This information must be stored in each LAR, so that the garbage collector knows its layout. Since functions may take no arguments or do no pattern-matching, the only part of an activation record that is always there is the access link pointer *prev*. It is then convenient to use its high 16 bits to encode a and n as two adjacent 8-bit quantities. (This effectively restricts our implementation to support functions taking up to 255 arguments and containing pattern-matching clauses nested at most 255 levels deep, which seem to be reasonable restrictions in practice.) Bit 0 is also used by the garbage collector and will be described later in this section.

The resulting layout of the *prev* pointer is shown in the top of Figure 4. In the following code, `PTRMASK` is the mask that extracts the pointer body and `ARINFO` initializes the *prev* pointer during LAR construction:

```
#define PTRMASK           0x0000fffffffffff8
#define ARINFO(a,n,prev) (((uintptr_t) a << 56) \
                        | ((uintptr_t) n << 48) \
                 | ((uintptr_t) prev & PTRMASK))
```

The two fields for arity and nesting can be extracted with these macros:

```
#define AR_a(p) ((uintptr_t) p >> 56)
#define AR_n(p) (((uintptr_t) p >> 48) & 0xff)
```

[2] We can modify such a collector to identify our pointers, but it is not clear how well such a strategy would work in practice in a conservative setting.

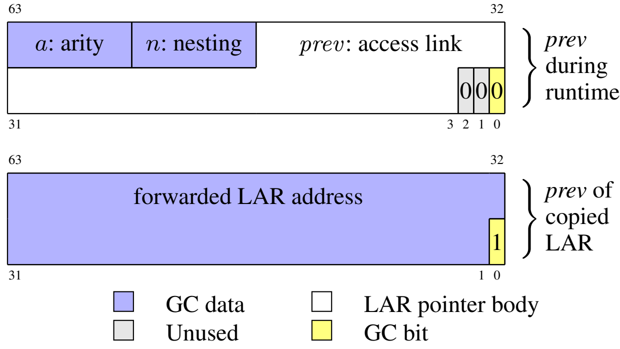

Figure 4. Embedding information for garbage collection in the *prev* field.

The low three bits of the *prev* pointer are initially cleared by PTRMASK and the garbage collector bit is always cleared when the collector is not running. Consequently, when the program is not doing a collection, *prev* can be accessed during runtime without masking the low bits:

```
#define AR_prev(p) ((intptr_t) p << 16 >> 16)
```

A semi-space garbage collector needs space to store the *forwarded address* [26] of an already copied object — in practice, implementations usually repurpose some space on the collected object for this field. In our case, we reuse the *prev* pointer to store the forwarded address of the LAR and set bit 0 of *prev* to indicate if the LAR is forwarded so that we can take its new address (macros IS_FORWARDED and FORWARDED_ADDR):

```
#define IS_FORWARDED(lar)   ((uintptr_t) lar->prev & 1)
#define FORWARDED_ADDR(lar) ((uintptr_t) lar->prev & ~1)
```

The resulting representation is shown in the bottom of Figure 4.

5. Benchmarks

We evaluate our prototype in two ways: (a) we measure its performance against GHC, to ensure that it is efficient, and (b) we compare the compact representation with a less compact thunk representation, used in a previous version of our prototype.

5.1 Evaluation against GHC

To evaluate the performance of our prototype, we run a set of standard Haskell benchmarks compiled by both GIC and GHC and compare running times. Of course, the benchmarks compare not only the efficiency of the memory representation proposed in this paper. They depend heavily on the efficiency of the complete GIC compiler, which is based on defunctionalization [20] and the intensional transformation [21].

GHC is the state-of-the-art compiler for Haskell, incorporating decades of research on optimizations and on implementation techniques for lazy languages. On the contrary, our compiler[3] is at an early development stage, missing many features; for example it does not do any optimizing code transformations, such as fusion or inlining. It does however two simple optimizations based on static analysis: (i) a sharing analysis [19] that evaluates unshared thunks using a call-by-name strategy, and (ii) an analysis that allocates LARs on the stack for functions that return values that do not let their environment escape (such as integers) [6].

The experiments were performed on a machine with four Intel Xeon® E7340 CPUs (2.40 GHz), having a total of 16 cores,

[3] Available from `https://github.com/gfour/gic`.

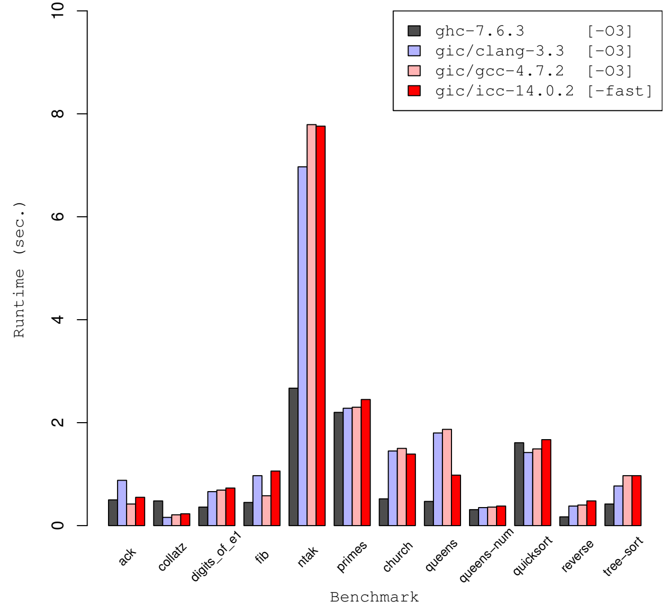

Figure 5. Execution time of benchmarks compiled with GHC and GIC (smaller is better).

with 4 MB cache and 16 GB RAM, running Debian GNU/Linux 7.3. We used GHC 7.6.3 as the reference Haskell compiler and LLVM 3.3/Clang as the C back-end of our compiler. We also tested GCC and the Intel C Compiler on our benchmarks, to account for compiler-specific performance quirks. We ran GHC with all optimizations turned on (-O3), using its default back-end. Explicit type signatures were inserted in all benchmarks, to remove any unneeded polymorphism overhead that GHC might introduce, especially when handling integers. We compiled the code generated by GIC with all optimizations turned on as well, in all three C compilers.

Figure 5 shows a comparison of the execution time for several benchmark programs, compiled with GHC and GIC. We see that (a) `collatz` performs better when compiled with GIC, (b) `primes`, `queens-num` and `quicksort` have roughly the same performance, and (c) the rest (`ack`, `digits_of_e1`, `fib`, `ntak`, `church`, `queens`, `reverse`, `tree-sort`) run faster with GHC. We also find that Clang, GCC, and the Intel C Compiler do not differ much regarding the performance of the selected benchmarks, although there are cases where the first (`ntak`, `tree-sort`), the second (`fib`), or the third (`ack`, `queens`) performs better.

Figure 6 shows the cache behavior of the test programs, measured using the Cachegrind tool [38]. We see here that (a) `collatz`, `digits_of_e1` and `queens-num` exhibit fewer cache misses when compiled with GIC, (b) `ntak`, `primes`, and `queens` show similar cache misses, and (c) `ack`, `church`, `fib`, `quick-sort`, `reverse`, and `tree-sort` show fewer misses when compiled with GHC.

The `church` benchmark makes extensive use of closures, stressing our implementation of defunctionalization (as Danvy and Nielsen indicate, Church encoding is in fact the inverse of defunctionalization [18]). Our worse performance than GHC may be the result of missing optimizations or it could mean that our defunctionalization is not mature for closure-heavy programs. The latter can be addressed if specialized optimizations are used, such

	GHC					GIC, previous representation					GIC, compact representation				
	I1	LLi	D1	LLd	LL	I1	LLi	D1	LLd	LL	I1	LLi	D1	LLd	LL
ack	0.0	0.0	6.1	2.1	0.3	0.0	0.0	10.1	0.0	0.0	0.0	0.0	10.8	0.0	0.0
church	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
collatz	0.0	0.0	0.7	0.0	0.0	0.0	0.0	0.3	0.2	0.1	0.0	0.0	0.1	0.1	0.0
digits_of_e1	0.0	0.0	3.2	0.0	0.0	0.0	0.0	2.4	1.8	0.6	0.0	0.0	1.3	0.8	0.2
fib	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
ntak	0.0	0.0	1.6	0.0	0.0	0.0	0.0	2.4	1.9	0.8	0.0	0.0	1.0	0.9	0.2
primes	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.1	0.1	0.0	0.0	0.0	0.1	0.1	0.0
queens	0.0	0.0	0.3	0.0	0.0	0.0	0.0	0.8	0.6	0.2	0.0	0.0	0.4	0.3	0.0
queens-num	0.0	0.0	0.9	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
quick-sort	0.0	0.0	4.2	0.5	0.1	0.0	0.0	11.2	3.9	1.6	0.0	0.0	8.2	1.7	0.5
reverse	0.0	0.0	8.2	0.0	0.0	0.0	0.0	23.4	8.0	3.3	0.0	0.0	15.0	3.6	1.0
tree-sort	0.0	0.0	7.3	0.0	0.0	0.0	0.0	14.4	3.4	1.4	0.0	0.0	7.9	1.6	0.4

Figure 6. Cache miss rates reported by Cachegrind (%). I1: first-level instruction cache. LLi: last-level instruction cache. D1: first-level data cache. LLd: last-level data cache. LL: last-level combined cache. Zeroes are shown as greyed out values.

as the flow-directed analysis of the defunctionalizing compiler MLton [15]. However, defunctionalization is a technique orthogonal to the compact representation proposed in this paper; we chose to include this benchmark for completeness, as higher-order programs are an essential part of the Haskell style of programming.

We also found that the fast arithmetic operators mentioned in Section 3 (such as `SUM`) did not make a noticeable difference in practice for these benchmarks.

In general, the benchmarks show potential, as our prototype is rather simple (7908 lines of Haskell and C)[4] and lacks most optimizations implemented by GHC. For the small programs at hand, this compact representation for thunks and activation records exhibits program running times in the same order of magnitude as the state-of-the-art GHC.

5.2 Evaluating the representation

We compare the compact representation with the previous version of our implementation, in which thunks were less compact. The previous representation was simpler, using three words for each thunk: (a) the code pointer, (b) the constructor tag or an integer, and (c) the context pointer (or zero, if the thunk is an evaluated integer). The code pointer doubled as a thunk flag, using the `NULL` pointer to indicate that the thunk had already been evaluated. Lazy activation records also kept information for garbage collection, in a separate word.

We compiled the test programs using both representations and benchmarked them, comparing running time, memory use, and cache behavior.

The first measurement shown in Figure 7 is the runtime comparison, where the compact representation performs better for most programs except `primes` and `queens-num`. We believe that the worse performance in the two benchmarks indicates that repeated mathematical operations (such as the primality and the n-queens tests) may be expensive in a very compact representation.

The measurements in Figure 6 display the cache behavior of the two representations: here the compact representation causes smaller miss rates in the simulated cache hierarchy of Cachegrind, with the exception of `ack`, which has slightly more first-level data-cache misses. Some programs show a small improvement (e.g. `primes`), while others show a substantial advantage (`quick-sort`, `reverse`, `tree-sort`).

An expected outcome of our technique was that programs using the compact representation needed less space than the ones using the previous representation. This is shown in the table of Figure 8,

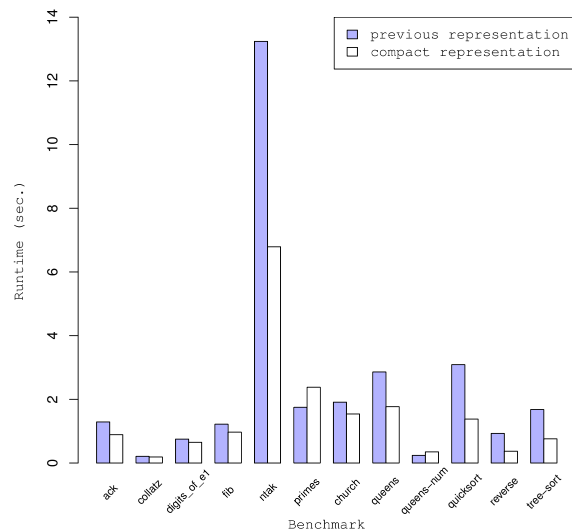

Figure 7. Runtime comparison between the two representations.

which shows how much heap space was used by the programs, assuming no garbage collection has taken place yet. We do not show `ack`, `fib`, and `primes`, since they only use the stack. In general, there is an average 68.2% reduction in heap usage; this more than triple reduction makes sense, since most functions of the test programs have few arguments and the savings on the LAR metadata are also significant. Stack measurements are not shown here; all programs ran with a maximum of 262,144 bytes of stack, so similar reduction applies in this smaller scale, as the same data (LARs containing thunks) are stored in the stack.

The results show that in most cases the compact representation is better than the previous one: it needs one third of the memory, it is usually faster, and has better cache characteristics. Also, the small memory usage makes our technique suitable for restricted memory environments, such as embedded systems.

6. Related Work

Hiding information in the low bits of aligned pointers (*pointer tagging*) is an old and tested technique used in language implementa-

[4] The line count was generated using David A. Wheeler's `SLOCCount`.

Heap usage in bytes			
Benchmark	Previous	Compact	Reduction
collatz,	41,360,250	13,200,080	68.1%
digits_of_e1	602,368,070	192,246,560	68.1%
ntak	11,516,398,810	3,636,757,520	68.4%
church	2,198,570	672,690	69.4%
queens	1,503,132,770	498,798,920	66.8%
queens-num	8,140	2,600	68.1%
quicksort	2,788,938,330	875,448,910	68.6%
reverse	685,380,260	216,432,080	68.4%
tree-sort	1,652,780,690	525,870,220	68.2%

Figure 8. Memory usage in the two representations, without garbage collection.

tions [25]. For example, Smalltalk embedded 15-bit signed integers (instances of the *SmallInteger* class) in 16-bit aligned pointers [23] and ML constructors were implemented using tagged pointers [12]. Another more recent example is the Erlang/OTP runtime system which uses an elaborate tag scheme to represent many objects as single words (or even half-words in 64-bit systems) [42].

Implementations of lazy programming languages have also used tagged pointers [29]. In particular, GHC uses aligned pointer tagging [34] to distinguish data constructors for types with few constructor tags. Our technique can be seen as a generalization, permitting the use of data types with many constructors. GHC is based on the STG machine [27], which assumes that thunks are more heavyweight than ours, containing an *info pointer* to useful metadata (such as information for debugging and parallel execution). In our representation we do not support debugging metadata. To support concurrent thunk evaluation, a solution would to be to add an extra thunk field containing a lock or equivalent synchronization mechanism. However, we could still keep our compact one-word thunks if we used a test-and-set instruction on the highest thunk bit (bringing constructor tags down to 15 bits and primitive values down to 61 bits), effectively using that bit to implement a spin lock. One or more bits can be used to implement more complex synchronization, such as the black-hole technique used in GHC [35], in which case the runtime system should keep extra memory for book-keeping (such as queues for suspended computations).

The prevalence of the AMD64 architecture with its 48-bit addressing has pushed other implementations of statically typed programming languages to utilize the spare high bits in pointers. For example, high-bit tagging is used in the 64-bit Objective-C runtime in recent releases of the Mac OS X and iOS operating systems [7].

NaN-boxing is another technique that embeds information inside floating-point numbers, exploiting redundancy in the IEEE-754 representation [25]. It has also become popular, being incorporated in recent implementations of dynamic languages such as Lua [41], JavaScript [8], and Ruby [37].

An interesting application of NaN-boxing for Haskell was Caro's encoding that embeds 52-bit pointers inside 64-bit double-precision numbers in pH [13]. Compared to our representation, the encoding supports another subset of integers ($-2^{63} \ldots 2^{52} - 1$), in favor of direct representation of double-precision numbers. Our representation has enough space for unboxed single precision floating-point numbers; to support double-precision, a similar technique based on NaN-boxing should also be used. Caro's representation handles I-structures and M-structures, which have parallel semantics and are implemented with lenient evaluation [14]. The representation needs additional space, outside the 64 bits, for the thunk and synchronization flags; our representation starts from the sequential lazy case, with parallel evaluation as an orthogonal extension, keeping everything in a single 64-bit word. Caro also modified the technique to fit everything inside a 64-bit word, reducing addresses to 44 bits and supporting more integers ($-2^{63} \ldots 2^{63} - 1 - 2^{51}$) [14].

Appel noted that implementations of statically typed languages have all information necessary to eliminate runtime tags that inform the garbage collector [5]. As he studied a strict language, his remark is not about the thunk flag, which is a standard tag on lazy language implementations. His technique can be applied in our implementation too, if we eliminate the arity and nesting fields and the value flag in the thunks, in favor of statically generated LAR-specific type information. This information may be embedded as a LAR tag or kept as additional space. However, Appel also noted that this may not affect performance in practice.

Pointer compression is another technique that tries to reduce the space used by pointers, when the address space can be compressed to a smaller range, e.g. when data may only need a part of addressable memory [9, 30], or when different addresses share a common prefix [49]. The technique has been applied successfully in Java virtual machines [1, 39, 46, 47]. Our approach differs in that it can use the full address space, without any assumptions on the size or the use of the data structures of the program.

Runtime systems based on tagged pointers can be seen as software equivalents of language-specific tagged hardware architectures. Such architectures had been proposed in the past for the execution of functional programming languages (e.g., Lisp machines [24]) but were found to be unneeded for the non-dynamically-typed languages that were then available [22]. Our experiments show that AMD64 hardware, although radically different from these architectures, can be efficiently programmed using tag-bit techniques.

Finally, a by-product of our experiments is that C is a good back-end language for lazy implementations and buys a lot of off-the-shelf compiler technology; this is an observation also made recently by Liu *et al.* when developing the Intel Labs Haskell Research Compiler [33]. On the other hand, the AMD64 architecture offers convenient instructions such as `INSERTQ`/`EXTRQ` that operate on parts of 64-bit words [3] and can be used in a machine-code generator using our representation.

7. Concluding Remarks

We have presented an implementation of a lazy functional language, where redundancy in AMD64 pointers is exploited to store thunks as single 64-bit words. Programs compiled with our implementation are relatively fast, compared to the state-of-the-art compiler, and have a compact memory representation.

Our technique is not only applicable to modern AMD64 systems; future extensions of the architecture may change the addressing space to 52 bits [4, Section 1.1.2], still leaving unused space for the constructor tags. Our technique can also be used to implement lazy languages on 64-bit ARMv8 and SPARC processors, since their pointers show similar redundancy [32, 40]. It should also be noted that some versions of the Microsoft Windows operating system offer even more unused space in pointers, having just 44-bit addresses on AMD64 hardware [16].

Our representation can have an advantage on some parallel implementations of lazy languages. Simultaneously updating more than one bit fields in the same word is an atomic operation, eliminating the possibility of some data races in concurrent updates. We also believe that our technique can be adapted for use in compilers for strict functional programming languages.

Our prototype compiler currently targets a subset of Haskell, supporting monomorphic and (parametric) polymorphic programs, and GADTs. Its front-end is very simple, not supporting complex patterns in `case` clauses, `let` bindings (relying instead on a

lambda-lifter), or type classes. We are working on these features and we do not expect that adding them will affect our representation. In particular, type classes are implemented in Haskell using *dictionary* arguments, which can be seen as constructors [43]. Also, at present, our prototype does not support tail calls (except those possibly inserted by the C compiler), so there is much more potential for memory-efficiency in our compiler.

It must be noted that defunctionalization is not required for our compact representation to work but was merely used to simplify the implementation. As indicated in Figure 3, bit 2 of evaluated constructor thunks is unused and can be used to tag a distinct representation for closures. We believe that this alternative should be explored in practice; having 16-bit constructor tags means at most 2^{16} different closure constructors may appear in the whole program, which is a serious restriction when compiling big programs.

There is still redundancy in our representation. In particular, *prev* pointers have spare bits that can be used for example as extra flags by a more sophisticated garbage collector or by a JIT compiler such as Schilling's [44]. Code pointers have bits 1-2 and 48-63 unused, where information can be stored about unevaluated thunks. Pointers in the *nested* fields are only present when a function does pattern matching, so their unused space should be used for case-clause-specific runtime information. Finally, 16 bits for constructor tags may be too many in practice; some of this space can be reused to store some other constructor-specific information.

Since memory pressure and bandwidth limitations are very important in multi-core systems, we are working on extending our prototype with a parallel runtime, in order to evaluate our approach in a parallel setting.

Acknowledgments

This research has been co-financed by the European Union (European Social Fund — ESF) and Greek national funds through the Operational Program "Education and Lifelong Learning" of the National Strategic Reference Framework (NSRF) — Research Funding Program: THALES. Investing in knowledge society through the European Social Fund. Project code: MIS 380153.

References

[1] A.-R. Adl-Tabatabai, J. Bharadwaj, M. Cierniak, M. Eng, J. Fang, B. T. Lewis, B. R. Murphy, and J. M. Stichnoth. Improving 64-bit Java IPF performance by compressing heap references. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, CGO '04, pages 100–110, Washington, DC, USA, 2004. IEEE Computer Society. ISBN 0-7695-2102-9. DOI: `10.1109/CGO.2004.1281667`.

[2] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986.

[3] AMD. AMD64 architecture programmer's manual volume 2: 128-bit and 256-bit media instructions, rev. 3.16. `http://developer.amd.com/wordpress/media/2012/10/26568_APM_v4.pdf`, 2012. Accessed: 31-01-2014.

[4] AMD. AMD64 architecture programmer's manual volume 2: System programming, rev. 3.23. `http://developer.amd.com/wordpress/media/2012/10/24593_APM_v21.pdf`, 2012. Accessed: 31-01-2014.

[5] A. W. Appel. Runtime tags aren't necessary. *LISP and Symbolic Computation*, 2(2):153–162, 1989. DOI: `10.1007/BF01811537`.

[6] G. Argo. Improving the three instruction machine. In *Proceedings of the Fourth International Conference on Functional Programming Languages and Computer Architecture*, FPCA '89, pages 100–115, New York, NY, USA, 1989. ACM. DOI: `10.1145/99370.99378`.

[7] M. Ash. Friday Q&A 2013-09-27: ARM64 and you (blog entry). `https://www.mikeash.com/pyblog/friday-qa-2013-09-27-arm64-and-you.html`. Accessed: 24-01-2014.

[8] L.-R. Babé. Firefox 4 performance. `https://hacks.mozilla.org/2011/03/firefox4-performance/`, 2011. Accessed: 31-01-2014.

[9] D. G. Bobrow. A note on hash linking. *Communications of the ACM*, 18(7):413–415, July 1975. DOI: `10.1145/360881.360920`.

[10] H. Boehm. A garbage collector for C and C++. `http://www.hpl.hp.com/personal/Hans_Boehm/gc/`, 2014. Accessed: 01-03-2014.

[11] S. Y. Borkar, H. Mulder, P. Dubey, S. S. Pawlowski, K. C. Kahn, D. J. Kuck, and J. R. Rattner. Platform 2015: Intel processor and platform evolution for the next decade. Whitepaper, Intel, 2005.

[12] L. Cardelli. Compiling a functional language. In *Proceedings of the 1984 ACM Symposium on LISP and Functional Programming*, LFP '84, pages 208–217, New York, NY, USA, 1984. ACM. DOI: `10.1145/800055.802037`.

[13] A. Caro. A novel 64 bit data representation for garbage collection and synchronizing memory. Technical Report Computation Structures Group Memo 396, Intel, April 1997. `http://csg.csail.mit.edu/pubs/memos/Memo-396/memo-396.pdf`.

[14] A. Caro. *Generating Multithreaded Code from Parallel Haskell for Symmetric Multiprocessors*. PhD thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, February 1999.

[15] H. Cejtin, S. Jagannathan, and S. Weeks. Flow-directed closure conversion for typed languages. In G. Smolka, editor, *Programming Languages and Systems*, volume 1782 of *Lecture Notes in Computer Science*, pages 56–71. Springer Berlin Heidelberg, 2000. DOI: `10.1007/3-540-46425-5_4`.

[16] D. Chernicoff. The corporate datacenter – it's ready for Windows Server x64 editions. `http://download.microsoft.com/download/4/4/7/44715635-144e-4726-8705-b8ec83c0a34d/The%20Corporate%20Datacenter-It%27s%20Ready%20for%20Windows%20Server%20x64%20Editions.pdf`, 2007. Accessed: 31-01-2014.

[17] T. Chilimbi, M. Hill, and J. Larus. Making pointer-based data structures cache conscious. *Computer*, 33(12):67–74, Dec 2000. DOI: `10.1109/2.889095`.

[18] O. Danvy and L. R. Nielsen. Defunctionalization at work. In *Proceedings of the 3rd ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming*, PPDP '01, pages 162–174, New York, NY, USA, 2001. ACM. DOI: `10.1145/773184.773202`.

[19] J. Fairbairn and S. Wray. Tim: A simple, lazy abstract machine to execute supercombinators. In G. Kahn, editor, *Functional Programming Languages and Computer Architecture*, volume 274 of *Lecture Notes in Computer Science*, pages 34–45. Springer Berlin Heidelberg, 1987. DOI: `10.1007/3-540-18317-5_3`.

[20] G. Fourtounis and N. S. Papaspyrou. Supporting Separate Compilation in a Defunctionalizing Compiler. In J. P. Leal, R. Rocha, and A. Simões, editors, *2nd Symposium on Languages, Applications and Technologies*, volume 29 of *OpenAccess Series in Informatics (OASIcs)*, pages 39–49, Dagstuhl, Germany, 2013. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. DOI: `10.4230/OASIcs.SLATE.2013.39`.

[21] G. Fourtounis, N. Papaspyrou, and P. Rondogiannis. The generalized intensional transformation for implementing lazy functional languages. In *Proceedings of the 15th International Symposium on Practical Aspects of Declarative Languages (PADL '13)*, volume 7752 of *Lecture Notes in Computer Science*, pages 157–172, Rome, Italy, Jan. 2013. Springer. DOI: `10.1007/978-3-642-45284-0_11`.

[22] E. F. Gehringer and J. L. Keedy. Tagged architecture: How compelling are its advantages? In *Proceedings of the 12th Annual International Symposium on Computer Architecture*, ISCA '85, pages 162–170, 1985. DOI: `10.1145/327070.327153`.

[23] A. Goldberg and D. Robson. *Smalltalk-80: The Language and Its Implementation*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1983.

[24] E. Goto, T. Soma, N. Inada, T. Ida, M. Idesawa, K. Hiraki, M. Suzuki, K. Shimizu, and B. Philipov. Design of a Lisp Machine – FLATS. In *Proceedings of the 1982 ACM Symposium on LISP and Functional Programming*, LFP '82, pages 208–215, New York, NY, USA, 1982. ACM. DOI: `10.1145/800068.802152`.

[25] D. Gudeman. Representing type information in dynamically typed languages. Technical Report TR 93-27, Department of Computer Science, The University of Arizona, October 1993.

[26] R. Jones, A. Hosking, and E. Moss. *The Garbage Collection Handbook: The Art of Automatic Memory Management*. Chapman & Hall/CRC, 1st edition, 2011.

[27] S. L. P. Jones. Implementing lazy functional languages on stock hardware: the Spineless Tagless G-machine – version 2.5. *Journal of Functional Programming*, 2:127–202, 1992.

[28] S. L. P. Jones, editor. *Haskell 98 Language and Libraries: The Revised Report*. `http://haskell.org/`, September 2002.

[29] K. Langendoen and P. H. Hartel. FCG: A code generator for lazy functional languages. In U. Kastens and P. Pfahler, editors, *Compiler Construction*, volume 641 of *Lecture Notes in Computer Science*, pages 278–296. Springer Berlin Heidelberg, 1992. DOI: `10.1007/3-540-55984-1_26`.

[30] C. Lattner and V. S. Adve. Transparent pointer compression for linked data structures. In *Proceedings of the 2005 Workshop on Memory System Performance*, MSP '05, pages 24–35, New York, NY, USA, 2005. ACM. DOI: `10.1145/1111583.1111587`.

[31] C. Lefurgy, E. Piccininni, and T. Mudge. Evaluation of a high performance code compression method. In *Proceedings of the 32nd Annual ACM/IEEE International Symposium on Microarchitecture*, MICRO 32, pages 93–102, Washington, DC, USA, 1999. IEEE Computer Society.

[32] A. Limited. Principles of ARM memory maps. `http://infocenter.arm.com/help/topic/com.arm.doc.den0001c/DEN0001C_principles_of_arm_memory_maps.pdf`, 2012. Accessed: 29-01-2014.

[33] H. Liu, N. Glew, L. Petersen, and T. A. Anderson. The Intel Labs Haskell Research Compiler. In *Proceedings of the 2013 ACM SIGPLAN Symposium on Haskell*, Haskell '13, pages 105–116, New York, NY, USA, 2013. ACM. DOI: `10.1145/2503778.2503779`.

[34] S. Marlow, A. R. Yakushev, and S. L. Peyton Jones. Faster laziness using dynamic pointer tagging. In *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming*, ICFP '07, pages 277–288, New York, NY, USA, 2007. ACM. DOI: `10.1145/1291151.1291194`.

[35] S. Marlow, S. Peyton Jones, and S. Singh. Runtime support for multicore Haskell. In *Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming*, ICFP '09, pages 65–78, New York, NY, USA, 2009. ACM. DOI: `10.1145/1596550.1596563`.

[36] M. Matz, J. Hubička, A. Jaeger, and M. Mitchell. AMD64 ABI draft 0.99.5, September 3, 2010. URL `www.x86-64.org/documentation/abi.pdf`. Accessed: 31-01-2014.

[37] Mruby developers. Full-length nan boxing on 64bit platforms, #1441. `https://github.com/mruby/mruby/pull/1441`, 2013. Accessed: 31-01-2014.

[38] N. Nethercote and A. Mycroft. The cache behaviour of large lazy functional programs on stock hardware. In *Proceedings of the 2002 Workshop on Memory System Performance*, MSP '02, pages 44–55, New York, NY, USA, 2002. ACM. DOI: `10.1145/773146.773044`.

[39] Oracle. Compressedoops. `https://wikis.oracle.com/display/HotSpotInternals/CompressedOops`, 2012. Accessed: 14-05-2014.

[40] Oracle. SPARC Assembly Language Reference Manual, 5.2: Address Sizes. `http://docs.oracle.com/cd/E26502_01/html/E28387/glyfs.html`. Accessed: 28-01-2014.

[41] M. Pall. LuaJIT 2.0 intellectual property disclosure and research opportunities. `http://lua-users.org/lists/lua-l/2009-11/msg00089.html`, 2009. Accessed: 31-01-2014.

[42] N. Papaspyrou and K. Sagonas. On preserving term sharing in the Erlang virtual machine. In *Proceedings of the Eleventh ACM SIGPLAN Workshop on Erlang Workshop*, Erlang '12, pages 11–20, New York, NY, USA, 2012. ACM. DOI: `10.1145/2364489.2364493`.

[43] F. Pottier and N. Gauthier. Polymorphic typed defunctionalization and concretization. *Higher-Order and Symbolic Computation*, 19: 125–162, 2006. DOI: `10.1007/s10990-006-8611-7`.

[44] T. Schilling. Challenges for a trace-based just-in-time compiler for Haskell. In A. Gill and J. Hage, editors, *Implementation and Application of Functional Languages*, volume 7257 of *Lecture Notes in Computer Science*, pages 51–68. Springer Berlin Heidelberg, 2012. DOI: `10.1007/978-3-642-34407-7_4`.

[45] The IEEE and The Open Group. The Open Group Base Specifications Issue 6, IEEE Std 1003.1, stdint.h – integer types. `http://pubs.opengroup.org/onlinepubs/000095399/basedefs/stdint.h.html`, 2004. Accessed: 28-01-2014.

[46] K. Venstermans, L. Eeckhout, and K. D. Bosschere. Object-relative addressing: Compressed pointers in 64-bit Java virtual machines. In E. Ernst, editor, *ECOOP*, volume 4609 of *Lecture Notes in Computer Science*, pages 79–100. Springer, 2007.

[47] K. Venstermans, L. Eeckhout, and K. De Bosschere. Java object header elimination for reduced memory consumption in 64-bit virtual machines. *ACM Transactions on Architecture and Code Optimization*, 4(3), Sept. 2007. DOI: `10.1145/1275937.1275941`.

[48] W. A. Wulf and S. A. McKee. Hitting the memory wall: Implications of the obvious. *SIGARCH Computer Architecture News*, 23(1):20–24, Mar. 1995. DOI: `10.1145/216585.216588`.

[49] Y. Zhang and R. Gupta. Data compression transformations for dynamically allocated data structures. In R. N. Horspool, editor, *CC*, volume 2304 of *Lecture Notes in Computer Science*, pages 14–28. Springer, 2002.