Efficient Communication and Collection with Compact Normal Forms

Edward Z. Yang[1] Giovanni Campagna[1] Ömer Ağacan[2] Ahmed El-Hassany[2]
Abhishek Kulkarni[2] Ryan Newton[2]

Stanford University[1], Indiana University[2]
{ezyang, gcampagn}@cs.stanford.edu {oagacan, ahassany, adkulkar, rrnewton}@indiana.edu

Abstract

In distributed applications, the transmission of non-contiguous data structures is greatly slowed down by the need to serialize them into a buffer before sending. We describe Compact Normal Forms, an API that allows programmers to explicitly place immutable heap objects into regions, which can both be accessed like ordinary data as well as efficiently transmitted over the network. The process of placing objects into compact regions (essentially a copy) is faster than any serializer and can be amortized over a series of functional updates to the data structure in question. We implement this scheme in the Glasgow Haskell Compiler and show that even with the space expansion attendant with memory-oriented data structure representations, we achieve between $\times 2$ and $\times 4$ speedups on fast local networks with sufficiently large data structures.

Categories and Subject Descriptors D.3.2 [*Concurrent, Distributed, and Parallel Languages*]

General Terms Languages, Performance

1. Introduction

In networked and distributed applications it is important to quickly transmit data structures from one node to another. However, this desire is often in tension with the usual properties of high-level languages:

- Memory-safe languages such as Haskell or Java support rich, irregular data structures occupying an unbounded number of non-contiguous heap locations.
- Network interface cards (NICs) perform best when the data to be sent resides in a single contiguous memory region, *ideally* pinned to physical memory for direct memory access (DMA).

Thus, while efficiently sending byte arrays does not pose a problem for high-level languages, more complex data structures require a serialization step which translates the structure into a contiguous buffer that is then sent over the network. This serialization process is a source of overhead and can be the limiting factor when an application runs over a fast network.

In response to this problem, there have been several attempts to engineer runtime support enabling high-level languages to send heap representations directly over the network: e.g. in Java [8], or even in distributed Haskell implementations [19]. However, these approaches rarely manage to achieve *zero-copy* data transmission, and complications abound with mutable and higher order data.

In this paper, we propose a new point in the design space: we argue it's worth adopting the same network representation as the native in-memory representation, despite the cost in portability and message size. We show that even when message size increases by a factor of four, on a fast local network—like those found in distributed computation settings—end-to-end performance can still be improved by a factor of two.

The benefit of all this is that the problem of fast network transfer reduces to the problem of arranging for heap data to live in contiguous regions. While *region type systems* [2, 12, 29] could address this problem, we implement a simpler solution which requires no changes to the type system of Haskell: let programmers explicitly place immutable data into *compact regions*. Objects in these regions are laid out in the same way as ordinary objects, they can be accessed in the same way from ordinary Haskell code and updated in the standard manner of purely functional data structures (the new nodes appended to the compact region). Furthermore, as this data in question is immutable and has no outgoing pointers, we side step the normal memory management problems associated with subdividing the heap (as in generational and distributed collectors). Finally, given any heap object we can quickly test for membership in a compact region, from which we can also deduce whether it is fully evaluated, a question which is often asked in a lazy language like Haskell.

If fast network transmission between trusted processes was the only reason for CNFs, they might not seem so appealing, but adding CNF to Haskell *also* solves two other, seemingly unrelated problems:

- *Permanent data residency.* In long-running programs, there may be some large data structures which never become garbage. With a standard generational garbage collector, these data structures must still be fully traversed upon a major GC, adding major overhead. In these cases, it is useful to *promote* such data to an immortal generation which is *never* traced.
- *Repeated deepseq.* Even setting aside serialization, there are other reasons to *fully evaluate* data, even in a lazy language. For example, in a parallel computation settings, it is important to ensure that computational work is not accidentally offloaded onto the wrong thread by transmission of a thunk.

This *hyperstrict* programming in Haskell is done with the `NFData` type class, which permits a function to deeply evaluate a structure to normal form. However, `deepseq (deepseq x)` demonstrates a problem with the approach. The second `deepseq` should cost $O(1)$, as the data is already in normal form. However, as there is no tracking of normal forms either in the type system or the runtime, Haskell's `NFData` methods must perform *repeated traversals of data*, which can easily lead to accidental increases in the asymptotic complexity of an algorithm.

Once data is compacted in a CNF, repeated deepseq becomes $O(1)$, and the garbage collector likewise never needs to trace the data again. More generally, we make the following contributions:

- We propose a basic API for CNFs, specify what invariants it enforces and formally describe some of its sharing properties. Surprisingly, our API is *referentially transparent*, although to allow for greater control over performance it lives in the `IO` monad.
- We implement CNFs by modifying the Glasgow Haskell Compiler (GHC) and runtime and compare CNF to accepted, high-performance serialization approaches for Haskell. We demonstrate while that Haskell serialization is competitive with well-optimized alternatives (e.g. the Oracle Java virtual machine), the CNF approach is radically faster. Further, we quantify how this serialization advantage translates into faster message passing or remote procedure calls (Section 5.7), including when used in conjunction with remote direct memory access.
- We show that CNF can also improve garbage collection: both in reducing GC time and scaling to large heaps (Section 5.5). CNFs offer a middle ground that enables some application control of heap storage without compromising type safety or requiring major type-system changes.

While the specific low level techniques applied in this technique are not novel, we hope to show that with this work, distributed functional programming can become *much more efficient than it has been*. This is especially apt, as in recent years there has been extensive work on distributed Haskell frameworks [10, 20], which depend on slow serialization passes to send data. By leveraging CNF, we can give functional programming with immutable data both speed *and* safety in high-performance distributed programming.

2. Motivation: Serialization and its Discontents

Consider the simple problem of building and then sending a tree value to another process:

```
sendBytes sock (serialize (buildTree x))
```

In general, *serializing* the tree, that is, translating it into some well-defined format for network communication, is unavoidable, since the receiving process may be written in a completely different language, by a completely different set of people, in which case a portable interchange format is necessitated.

However, there are some situations where endpoints may be more closely related. If we are sending the tree to another thread in the same process, no serialization is necessary at all: just send the reference! Even in a distributed computation setting, it is relatively common for every process on the network to be running the *same binary*. We can summarize the possible situations by considering who we are sending to:

1. Another thread in the same process;
2. Another process in the network, trusted to be running the *same binary*;
3. A trusted endpoint in the network, which may not run the same binary; or perhaps
4. An untrusted endpoint across the network.

Most serialization and networking libraries are designed for the worst case—scenario 4—and thus miss out on substantial opportunities in cases 2 and 3. In Haskell, for example, the best option today is to use a binary serialization library such as `binary` or `cereal`. These libraries are very efficient examples of their kind, but by their nature they spend substantial time packing structures into an array of bytes and then unpacking them again on the other side.

Why should we care about scenarios 2 and 3? While scenario 4 covers general applications interacting with the internet, these middle scenarios represent applications running inside of supercomputers and data-centers composed of many nodes. In scenario 2, and possibly scenario 3, we can consider sending a representation that can be used *immediately* on the other side, *without* deserialization. High-performance networking hardware that provides *remote direct memory access* (RDMA), makes this scenario even more appealing, as it can directly place objects in remote heaps for later consumption *without* the involvement of remote processors. Thus, we have this principle:

PRINCIPLE 1. *To minimize serialization time, in-memory representation and network representation should be the same.*

Even if we are willing to accept this principle, however, there are still some difficulties.

2.1 Problem 1: Contiguous in-memory representation

By default, data allocated to the heap in a garbage collected language will *not* be in a contiguous region: it will be interspersed with various other temporary data. One direct solution to this problem might be to replace `(serialize (buildTree x))` from the earlier example code with an alternate version designed to produce a contiguous version of the tree, which could be immediately consumed by `sendBytes`:

```
sendBytes chan (buildTreeToRegion x)
```

The first problem with this approach is that its anti-modular if `buildTree` must be changed to yield `buildTreeToRegion`. The producer code may be produced by a library *not* under the control of the programmer invoking `sendBytes`—thus it is unreasonable to expect that the producer code be modified to suit the consumer. Nor is it reasonable to expect a program analysis to identify `buildTree` as producing network-bound data, because it is impossible to determine, in general (at all allocation sites) what the ultimate destination of each value will be. Besides, most high-level languages don't have the capability to region-allocate, even if we were willing to change the producer code.

A region-based type system with sufficient polymorphism could solve the modularity problem: a function identifies what region the returned value should be allocated into. But, while there have been languages that have this capability and expose it to users [12], widely used functional and object oriented languages do not. In fact, even MLKit [29]—which implements SML using regions and region-inference—does *not* expose region variables and `letregion` to the programmer. Thus they cannot write `buildTreeToRegion` and cannot guarantee that the result of `builtTree` ends up as the sole occupant of a distinct region.

Due to these drawbacks, we instead propose much simpler scheme: to simply *copy* the relevant data into the contiguous region. The key principle:

PRINCIPLE 2. *It is OK to copy, as long as the copy is amortized across all sends.*

In fact, when a copying garbage collector would be used, live data structures would have been copied anyway. We can do the copy once, and then avoid any further copies (by the garbage collector or otherwise.)

2.2 Problem 2: Safety

Once your data is in a contiguous, compact region, one would hope that it would simply be possible to send the entire region (without any checking) when attempting to send a pointer to an object in the region.

However, such an operation is only safe if the region, in fact, contains *all* of the reachable objects from a pointer. If this has been guaranteed (e.g., because a copy operates transitively on reachable objects), there is yet another hazard: if *mutation* is permitted on objects in the compact region, then a pointer could be mutated to point out of the region.

In fact, an analogous hazard presents itself with garbage collection: if a compact region has outbound regions, it is necessary to trace it in order to determine if it is keeping any other objects alive. However, if there are no outgoing pointers and the data is immutable, then it is impossible for a compact region to keep objects outside of it alive, and it is not necessary to trace its contents. To summarize:

PRINCIPLE 3. *Immutable data with no-outgoing pointers is highly desirable, from both a network transmission and a garbage collection standpoint.*

3. Compact Normal Form

Our goal with CNFs is to organize heap objects into regions, which can then be transmitted over the network or treated uniformly during garbage collection. Concretely, we do this by representing a pointer to an object in a compact region with the abstract type `Compact a`. Given a `Compact a`, a pointer to the actual object can be extracted using `getCompact`:

```
newtype Compact a
getCompact :: Compact a → a
```

How do we create a `Compact a`? Any such operation would need to take a value, fully evaluate it, and copy the result into a contiguous region. We represent the types which can be evaluated and copied in this way using a type class `Compactable`, similar to an existing Haskell type class `NFData` which indicates that a type can be evaluated to normal form. Most common types are compactable, e.g. `Bool` or `Maybe a` (if `a` is `Compactable`), but mutable types such as `IORef a` are not.

```
class NFData a ⇒ Compactable a
```

We might then try to define a function with this type:

```
newCompact :: Compactable a ⇒ a → IO (Compact a)
```

This function creates a new region and copies the fully evaluated `a` into it. Suppose, however, that we want to apply a functional update to this tree: with only `newCompact`, there would be no way to reuse data already living in a compact region for the new compact. Thus, `newCompact` should be decomposed into two functions:

```
mkCompact     :: IO (Compact ())
appendCompact :: Compactable a
              ⇒ a → Compact b → IO (Compact a)
```

`appendCompact`, like `newCompact`, fully evaluates `a`; however, it copies the result into the *same* compact region as `Compact b`. Additionally, it short-circuits the evaluation/copying process if a subgraph is already in the target compact region. (The actual heap object `Compact b` points to is otherwise ignored.) `mkCompact` then simply creates a new region and returns a dummy pointer `Compact ()` to the region, for use in `appendCompact`.

While one could quibble with the *particular* interface provided, the above interface is sufficient for all compactions—but we still need support for sending `Compact a` values over the network, e.g.:

```
sendCompact :: Socket → Compact a → IO ()
```

as in this example:

```
do c ← newCompact (buildTree x)
   sendCompact sock c
```

Referential transparency Despite the use of `IO`, `mkCompact` and `appendCompact` *are* referentially transparent, since we could implement non-monadic versions of these functions in the following way (where `deepseq` is a method in `NFData` which evaluates its first argument to normal form when the second argument is forced):

```
newtype Compact a = Compact a
mkCompact = Compact ()
appendCompact x _ = deepseq x (Compact x)
```

However, because compact normal forms are primarily motivated by performance, it does matter quite a bit due to operational reasons what data is shared, what compact regions are allocated, and which region data was copied into. Thus, these functions live in the `IO` monad[1].

There do exist some useful functions manipulating compacts which are *not* referentially transparent. For example, a function which tests if an arbitrary value lives in a compact region must live in the `IO` monad, because it can be used to observe sharing:

```
isCompact :: a → IO (Maybe (Compact a))
```

3.1 Region invariants

The `Compactable` type class enforces some important safety invariants on data which lives in a compact region:

- **No outgoing pointers**. Objects are copied completely into the compact region, so there are never any outgoing pointers. This is useful when transmitting a region over the network, as we know that if we send an entire region, it is self-contained. It also means, during garbage collection, it's not necessary to trace the inside of a region to determine liveness of other objects on the heap. Compacted objects are essentially a single array-of-bits heap object.
- **Immutability**. No mutable objects are permitted to be put in a compact region. This helps enforce the invariant of no outgoing pointers, and also means that data in a region can be copied with impunity.
- **No thunks**. Thunks are evaluated prior to being copied into a region; this means the CNF will not change layout, be mutated, or expand as a result of accessing its contents, and that we do not attempt to send closures over the network.

Haskell has especially good support for immutable data, which makes these restrictions reasonable for compact regions. While many languages now host libraries of purely functional, persistent data structures, in Haskell these are used heavily in virtually every program, and we can reasonably expect most structures will be `Compactable`.

Skipping tracing of the insides of compact regions has one implication: *if a single object in a compact region is live, all objects in a compact region are live*. This approximation can result in wasted

[1] Or the ST monad—ST would allow local control over sequencing and number of compacts, but of course at the point of exit (`runST`) the entire computation could be duplicated by the compiler, in principle.

space, as objects which become dead cannot be reclaimed. However, there a few major benefits to this approximation. First, long-lived data structures can be placed in a compact region to exclude them from garbage collection. Second, the avoidance of garbage collection means that, even in a system with copying garbage collection, the heap addresses of objects in the region are *stable* and do not change. Thus, compact regions serve as an alternate way of providing FFI access to Haskell objects. Finally, a "garbage collection" can be requested simply by copying the data into a new compact region, in the same manner a copying garbage collector proceeds.

3.2 Sharing

Because every compact region represents a contiguous region of objects, any given object can only belong to at most one compact region. This constraint has implications on the *sharing* behavior of this interface. Here are two examples which highlight this situation:

Sharing already compact subgraphs Consider this program:

```
do c ← mkCompact
   r1 ← appendCompact [3,2,1] c
   r2 ← appendCompact (4:r1) c
   -- Are 'tail r2' and r1 shared?
```

In the second `appendCompact`, we are adding the list `[4,3,2,1]`. However, the sublist `[3,2,1]` is already in the same compact region: thus, it *can* be shared.

However, suppose `r1` is in a different compact, as here:

```
do c1 ← mkCompact
   r1 ← appendCompact [3,2,1] c1
   c2 ← mkCompact
   r2 ← appendCompact (4:r1) c2
   -- Are 'tail r2' and r1 shared?
```

In this case, sharing would violate the compact region invariant. Instead, we must recopy `r1` into the new compact region. The copying behavior here makes it clear why, semantically, it doesn't make sense to allow mutable data in compact regions.

Sharing after append Consider the following program, where `t` is a thunk whose type is compactable:

```
do c ← mkCompact
   r ← appendCompact t c
   -- Are t and r shared?
```

The process of appending `t` to `c` caused it to be fully evaluated; furthermore, `r` refers to the fully evaluated version of this data structure which lives in the compact region. Is `t` updated to also point to this data structure?

In some cases, it is not possible to achieve this sharing: if `t` is a reference to a fully evaluated structure in different compact, it *must* be copied to the new compact region. Additionally, if `t` had already been fully evaluated, it's not possible to "modify" the result to point to the version in the new compact region. Thus, to make sharing behavior more predictable and indifferent to evaluation order, we decided `t` should never be updated to point to the version of the data structure in the compact.

Semantics We can be completely precise about the sharing properties of this interface by describing a big-step semantics for our combinators in the style of Launchbury's *natural semantics* [18]. To keep things concrete, we work with the specific intermediate language used by GHC called STG [15], which also supports data constructors. The syntax STG plus our combinators is described in Figure 1, with metavariables f and x representing variables, K representing constructors, and a representing either a literal or variable. STG is an untyped lambda calculus which has the same restriction as Launchbury natural semantics that all arguments a to

$$
\begin{array}{lcll}
e & ::= & & \\
 & | & \mathsf{lit} & \text{Literal} \\
 & | & f\ \overline{a_i}^{\,i} & \text{Application} \\
 & | & x & \text{Thunk} \\
 & | & K\ \overline{a_i}^{\,i} & \text{Constructor} \\
 & | & \mathbf{case}\ e\ \mathbf{of}\ \overline{K_i\ \overline{a}\ \rightarrow e_i}^{\,\mathbf{i}} & \text{Pattern match} \\
 & | & \mathbf{let}\ x = rhs\ \mathbf{in}\ e & \text{Let binding} \\
 & | & \mathbf{mkCompact} & \\
 & | & \mathbf{appendCompact}\ x\ y & \\
 & & & \\
rhs & ::= & & \text{Right-hand sides} \\
 & | & \lambda\ \overline{x_i}^{\,i}\ .e & \text{Function} \\
 & | & \ulcorner e \urcorner & \text{Thunk} \\
 & | & K\ \overline{a_i}^{\,i} & \text{Constructor}
\end{array}
$$

Figure 1: Syntax for simplified STG

function (and constructor) applications must either be a literal or a variable. This makes it easy to model the heap as a graph (with variables representing pointers); thus, sharing behavior can be described.

The basic transition in a big-step semantics is $\Gamma : e \Downarrow \Gamma' : a$: an expression e with heap Γ reduces to a value or literal with new heap Γ'. The semantics for the standard constructs in STG are completely standard, so we omit them; however, there is one important difference about Γ: a heap may also contain *labelled* bindings $x \overset{c}{\mapsto} v$, indicating the value in question lives in a compact region c. (Values in the normal heap implicitly have a special label ϵ). With this addition, the rules for the two combinators are then quite simple:

$$\frac{c\ \mathbf{fresh} \qquad x\ \mathbf{fresh}}{\Gamma : \mathbf{mkCompact} \Downarrow \Gamma[x \overset{c}{\mapsto} ()] : x}$$

$$\frac{\Gamma : x \Downarrow \Delta : x' \qquad x' \overset{c}{\mapsto} rhs\ \mathbf{in}\ \Gamma' \qquad \Delta : y \Downarrow_c^{\mathsf{rnf}} \Theta : y'}{\Gamma : \mathbf{appendCompact}\ x\ y \Downarrow \Theta : y'}$$

The rule for `appendCompact` hides some complexity, as it needs to recursively evaluate a data structure to normal form. We can express this process with a specialized evaluation rule $\Gamma : e \Downarrow_c^{\mathsf{rnf}} \Gamma' : a$, which indicates e should be fully evaluated and the result copied into the compact region c, where a points to the root of the copied result. There are only three rules for `rnf`:

$$\frac{\Gamma : e \Downarrow \Gamma' : z' \qquad \Gamma' : z' \Downarrow_c^{\mathsf{rnf}} \Gamma'' : z''}{\Gamma : e \Downarrow_c^{\mathsf{rnf}} \Gamma'' : z''}\ \textsc{Eval}$$

$$\frac{x \overset{c}{\mapsto} v\ \mathbf{in}\ \Gamma}{\Gamma : x \Downarrow_c^{\mathsf{rnf}} \Gamma : x}\ \textsc{ShortCut}$$

$$\frac{x \overset{c'}{\mapsto} K\ \overline{y_i}^{\,i}\ \mathbf{in}\ \Gamma_0 \qquad \overline{\Gamma_i : y_i \Downarrow_c^{\mathsf{rnf}} \Gamma_{i+1} : z_i}^{\,i}}{\Gamma_0 : x \Downarrow_c^{\mathsf{rnf}} \Gamma_n[z \overset{c}{\mapsto} K\ \overline{z_i}^{\,i}] : z}\ \textsc{ConRecurse}$$

First, if x is not fully evaluated, we evaluate it first using the standard reduction rules (Eval). Otherwise, if we are attempting to `rnf` a variable into c (ShortCut), but it already lives in that region, then nothing more is to be done. Otherwise, x already points to a constructor in weak head normal form but in a different region c' (ConRecurse), so we recursively `rnf` the arguments to

the constructor, and then allocate the constructor into the compact region c.

4. Implementation

4.1 The GHC runtime system

We first review some details of GHC runtime system. Readers already familiar with GHC's internals can safely skip to the next subsection.

Block-structured heap In GHC, the heap is divided in blocks of contiguous memory in multiples of 4KB. [21] The smallest block size is 4KB, but larger blocks can be allocated to hold objects which are larger than 4KB. Blocks are chained together in order to form regions of the heap, e.g. the generations associated with generational garbage collection.

In memory, blocks are part of aligned *megablocks* of one megabyte in size. These megablocks are the unit of allocation from the OS, and the first few blocks in each megablock are reserved for the *block descriptors*, fixed size structures containing metadata for one block in the same megablock. Because of this organization it is possible to switch between a block and a block descriptor using simple pointer arithmetic. Block descriptors contain information such as how large a block is (in case it holds an object larger than four kilobytes) and what portion of the block is in use.

This block structure gives the GHC runtime system the property that given an arbitrary pointer into the heap, it is possible in some cases to verify in constant time in what object it lives, and that property is exploited by our implementation to efficiently test if an object already lives in a compact region.

Layout of objects in the heap Since the in-memory representation of objects is what will be transmitted on the network, it is worth explaining how GHC lays out objects in memory. Objects are represented by a machine size *info pointer* followed by the payload of the object (numeric data and pointers to other object, in an order which depends on the object type).

The info pointer points to an *info table*, a static piece of data and code that uniquely identifies the representation of the object and the GC layout. In case of functions, thunks and stack continuations, it holds also the actual executable code, while for ADTs it contains an identifier for the constructor which is used to discriminate different objects in `case` expressions.

It is important to note that info tables are stored in the text segment of the executable and never change or move for the lifetime of the process. Moreover, in case of static linking, or dynamic linking without address space layout randomization, they are also consistent between different runs of the same binary. This means that no adjustment to info pointers is necessary when the same binary is used.

4.2 Compact regions

Conceptually, a compact region is a mutable object in which other objects can be added using the `appendCompact` operation. Operationally, a region is represented as a chain of blocks (hopefully one block long!) Each block of a compact region has a metadata header (in addition to the block descriptor associated with the block), which contains a pointer to the next and to the first block in the chain. Additionally, the first block of a compact region contains a tag which identifies the machine from which the data originated (the purpose of which is explained later in the section).

It is interesting to observe therefore that a compact region can be thought of as a heap object in itself: it can be treated as a linked list of opaque bytestrings which do not have to be traced. At the same time, the compact region internally contains other objects which can be directly addressed from outside.

Garbage collection Usually in a garbage collected language, it is unsafe to address component parts of an object known to the GC, because there is no way for the GC to identify the container of the component and mark it as reachable as well.

Nevertheless, for compacts this property is achievable: given an arbitrary address in the heap, we can find the associated block descriptor and use the information stored there to verify in constant time if the pointer refers to an object in a compact storage. If it does, we mark the entirety of the compact region as alive, and don't bother tracing its contents.

Furthermore, because we have this efficient test for membership in a specific compact, it can be employed to test if an object is already a member of a compact, or even if it is just in normal form (so a `deepseq` can be omitted).

4.3 Appending data to a compact region

As we've described, the process of appending data to a compact region is essentially a copy, short-circuiting when we encounter data which already lives in the compact region. However, we can avoid needing to perform this copy recursively by applying the same trick as in Cheney copying collection: the compact region also serves as the queue of pending objects which must be scanned and copied.

If copying would cause the block to overflow, a new block is allocated and appended to the chain, and copying then proceeds in the next block. The size of the appended block is a tunable parameter; in our current implementation, it is the same size as the previous block.

Preserving sharing while copying Suppose that we are copying a list into a compact region, where every element of the list points to the *same* object: the elements are all shared. In a normal copying garbage collector, the first time the object is copied to the new region, the original location would be replaced with a *forwarding* pointer, which indicates that the object had already been copied to some location.

However, in our case, we can't apply this trick, because there may be other threads of execution running with access to the original object. Initially, we attempted to preserve sharing in this situation by using a hash table, tracking the association of old objects and their copies in the compact region. Unfortunately, this incurred a significant slow-down (between $\times 1.5$ and $\times 2$).

Thus, our default implementation does *not* preserve sharing for objects which are not already in a compact region. (Indeed, this fact is implied by the semantics we have given.) Thanks to the fact that only immutable data is copied in this way, this duplication is semantically invisible to the application code, although memory requirements can in the worst case become exponential, and structures containing cycles cannot be handled by this API.

While it may seem like this is a major problem, we can still preserve sharing for data structures whose shared components already live in a compact region. In this case, when we take a data structure already in a compact region, apply some functional update to it, and append the result to it, the shared components of the new data structure continue to be shared. We believe internal sharing which does not arise from this process is less common, especially in data which is to be sent over the network.

Trusted Compactable instances The `Compactable` type class serves two purposes: first, it describes how to evaluate data to normal form while short-circuiting data which is already in normal form (the existing type class `NFData` always traverses the entirety of an object), and second, it enforces the safety invariant that no mutable objects be placed in a compact region.

Unfortunately, because `Compactable` type classes are user definable, a bad instance could lead in the type checker accepting a

copy of an impermissible type. Currently, our implementation additionally does a runtime check to ensure the object fulfills the invariants. Ideally, however, a `Compactable` would only be furnished via trusted instances provided by GHC, in a similar fashion to the existing `Typeable`. [17]

Mutable non-pointer objects One design choice to be made is whether or not objects with mutable non-pointer fields should be permitted in compact regions. One such object is the unboxed array, which backs useful data types such as `Data.Text`, which represents Unicode strings. However, if a mutable object can be placed in a compact region, it's possible to observe the difference between copies of the type in question: if you mutate one, the change will not show up in the copies. Fortunately, the type system distinguishes mutable and immutable unboxed arrays (even if the runtime does not), so provided no `Compactable` instances is provided for the mutable types referential transparency can be preserved.

4.4 Network communication

Once your data is in a compact region, you can use any standard techniques for sending buffers over the network. However, there are some complications, especially regarding *pointers* which are in the compact region, so for the sake of explicitness (and to help explain the experimental setups in the next section), we describe the details here.

Serialization A compact region is simply a list of blocks: thus, the serialization of a compact region is each block (and its length), as well as a pointer to the root of the data structure that is the root. The API we provide is agnostic to the *particular* network transport to be used:

```
data SerializedCompact a = S {
  blockList :: [(Ptr a, Word)],
  root :: Ptr a
}

withCompactPtrs :: Compact a
                → (SerializedCompact a → IO b)
                → IO b
importCompact :: SerializedCompact a
              → (Ptr b → Word → IO ())
              → IO (Compact a)
```

The two functions operate in pair: `withCompactPtrs` accepts a function `SerializedCompact a →IO b` that should write the data described by the `SerializedCompact` to the communication channel. Conversely, `importCompact` takes care of reserving space in the heap for the compact region using the `SerializedCompact` (transmitted out of band as simple address/size pairs), then calls the provided function `Ptr b →Word →IO ()` for each piece of reserved space: this function receives the data and places it at this address.

One property of this design is that the `SerializedCompact`, containing the addresses of blocks on the originating machine, must be sent in full through an out of band channel. This is to give a chance to the runtime system to allocate the blocks on the receiving machine at the right addresses from the start, which is necessary to allow full zero-copy transfer in a RDMA scenario.

Pointer adjustment If the data associated with a compact region is not loaded into the same address as its original address, it is necessary to offset all of the internal pointers so that they point to the new address of the data in question. This procedure can be skipped if the sender is trusted and the compact region is loaded to its original address.

To ensure that we will be able to load compact regions into the correct address space, we observe the address space in a 64-bit architecture (or even a 48 bit one like x86_64) is fairly large, more than the application will usually need. Therefore, our idea is to divide it into n *chunks* (in our case, 256 chunks of 128 GiB each) and assign each chunk to a specific machine/process combination.

Memory in these chunks is separated by the normal heap and is used only for compact storage, which means that every machine can have an approximate view of the contents of its assigned chunk in all other machines. This is enough to greatly reduce the number of collisions when attempting a directed allocation.

Unfortunately, this scheme is not guaranteed to work, as memory can be reused on the sender before it is reclaimed also on the receiver, triggering a collision and a linear pointer adjustment. An alternate design is to never reuse address space, simply unmapping the address for old compacts when they become no longer reachable.

Interoperation with different binaries As mentioned above, info tables for each object in a compact region are static and well-defined for a given binary. This allows us to ignore the info pointers inside the compact data, provided that the data originates from another instance of the same executable on a compatible machine. We verify this with an MD5 checksum of the binary and all loaded shared libraries, which is included in the payload of every compact sent on the wire and verified upon importing.

If this verification fails, the import code has to adjust the info pointers of all objects contained in the imported storage. One option to reconstruct the info pointers would be to send the info tables together with the data. Unfortunately, the info tables are fairly large objects, due to alignment and the presence of executable code, which makes this option not viable in practice. Additionally, the executable code can potentially make references to other pieces of code in the runtime system.

Instead, we observed that every info table is identified by a dynamic linker symbol which is accessible to the runtime. Thus, we extended the compact storage format to contain a map from all info table addresses to the symbol names, to be sent on the wire with the data. This map is employed to obtain symbol names for the transmitted info table pointers, which can then be linked against their true locations in the new binary.

Because this mapping incurs some overhead, we allowed the programmer to chose if he's willing to pay this cost for some more safety. On the other hand, we can cache the mapping on the receiver side, so if the types and data constructors of the values sent do not change, the setup cost needs to be paid only for the first message sent.

5. Evaluation

In this section, we characterize the performance of compact normal forms by looking both at serialization/deserialization and memory footprint costs, as well as end-to-end numbers involving network transmission, garbage collections and a key-value store case-study. The details of the experiments are in the upcoming subsections, but we first spend some time to describe our experimental setup.

In some experiments, we compare against the latest versions of the Haskell `binary` and `cereal`. We also compared against the builtin Java serialization engine (`java.io.Serializable`) shipped with Java HotSpot version 1.8.0_31, as a sanity check to ensure Haskell has reasonable performance to start with—we are not merely making slow programs less slow, nor are we addressing a Haskell specific problem.

There are a variety of different types which could use to serialize and deserialize. In our experiments, we used two variants of balanced binary trees with different pointer/total size ratios, varying sizes in power of two. In particular:

- `bintree` is a binary tree with a single unboxed integer in leaves. This variant has high pointer/total size ratio, and thus represents a worst case scenario for transmitting compact normal forms.

```
-- Pointer-heavy data with more pointers than scalars.
-- Representative of boxed, linked datatype in Haskell,
-- such as lists.
data BinTree = Tree BinTree BinTree
             | Leaf !Int

-- Small-struct data, increasing to a handful of scalars.
-- Representative of custom datatypes for numeric
-- and computationally intensive problems.
data PointTree
    = PTree PointTree PointTree
    | PLeaf { x :: !Int64
            , y :: !Int64
            , z :: !Int64
            , mass :: !Int64
            }

-- Small-array data, with small, unboxed strings.
data TweetMetaData =
     TweetMetaData { hashtags :: ![Text]
                   , user_id  :: !Int64
                   , urls     :: ![Text]
                   }
```

Figure 2: Our three representative data types for studying data transfer and storage. We do not cover large unboxed array data, because these types are already handled well by existing memory management and network transfer strategies.

- `pointtree` is a binary tree with four unboxed integers in leaves, increasing the data density.

Additionally, we also analyzed a third data type, composed of URLs, hashtags and user IDs for all posts in Twitter in the month of November 2012[22, 23].

Our experiments were done on a 16-node Dell PowerEdge R720 cluster. Each node is equipped with two 2.6GHz Intel Xeon E5-2670 processors with 8-cores each (16 cores in total), and 32GB memory each. For the network benchmarks over sockets, we used the 10G Ethernet network connected to a Dell PowerConnect 8024F switch. Nodes run Ubuntu Linux 12.04.5 with kernel version 3.2.0.

5.1 Serialization and deserialization costs

Our first evaluation compares the cost of *copying* data into a compact region with various serialization methods, as well as the resulting *space* usage of the serialized versions. In some sense, just comparing serialization is a worst case comparison: while the serialized form of a data structure is not generally useful for ordinary manipulation, compact normal forms can be accessed like normal data structures, so the cost of copying could be amortized.

In Figure 3, we see a log-log plot comparing serialization times for binary trees which store an integer at each node. We can see that at low tree sizes, constant factors dominate the creation of compact normal forms; however, at larger sizes copying is an order of magnitude faster than serializing. The line for Compact/Share, which refers to the implementation of compact which uses a hash table to preserve internal sharing, demonstrates the overhead of using a hash table to preserve out-of-line sharing. The graph for `pointtree` was comparable, and for Twitter the serialization overhead was consistently $\times 11$ for binary and between $\times 9$ and $\times 18$ for Java.

5.2 Memory overhead

In Table 1, we report the sizes of the various serialized representations of large versions of our data types; these ratios are representative of the asymptotic difference.

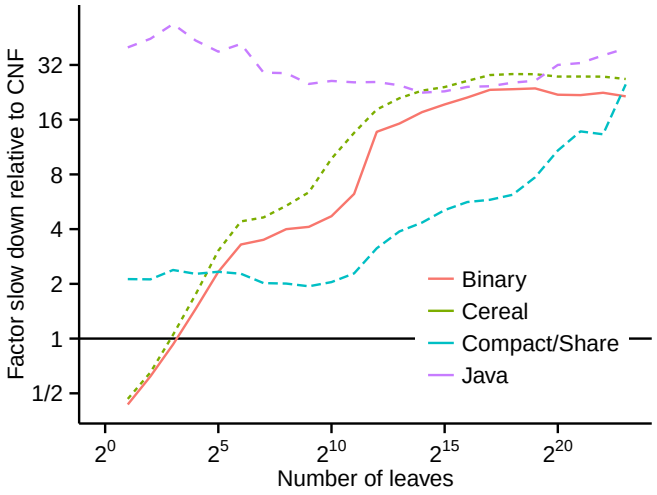


Figure 3: Relative improvement for serializing a `bintree` of size 2^N with CNFs versus other methods. Both x and y scales are logarithmic; bigger is better for CNF (and worse for the serializer being compared.) Compact/Share refers to the implementation of compact regions which preserves internal sharing.

Method	Type	Value Size	MBytes	Ratio
Compact	bintree	2^{23} leaves	320	1.00
Binary			80	0.25
Cereal			80	0.25
Java			160	0.50
Compact	pointtree	2^{23} leaves	512.01	1.00
Binary			272	0.53
Cereal			272	0.53
Java			400	0.78
Compact	twitter	1024MB	3527.97	1.00
Binary			897.25	0.25
Cereal			897.25	0.25
Java			978.15	0.28

Table 1: Serialized sizes of the selected datatypes using different methods.

We see that in the worst case, the native in-memory representation can represent a $\times 4$ space blow-up. This is because a serialization usually elides pointers by inlining data into the stream; furthermore tags for values are encoded in bytes rather than words. However, as the raw data increases, our ratios do get better. Interestingly, the Twitter data achieves a relatively poor ratio: this is in part because most of the strings in this data are quite small.

The difference in memory size sets the stage for the next set of experiments on network transfer latency.

5.3 Heap-to-Heap Network Transfer

Given that the size of data to be transmitted increases, the real question is whether or not the end-to-end performance of transmitting a heap object from one heap to another is improved by use of a compact normal form. With a fast network, we expect to have some slack: on a 1 Gbit connection, an extra 240 megabytes for a 2^{23} size binary tree costs us an extra 2.01 seconds; if serializing takes 6.92 seconds, we can easily make up for the slack (and things are better as more bandwidth is available).

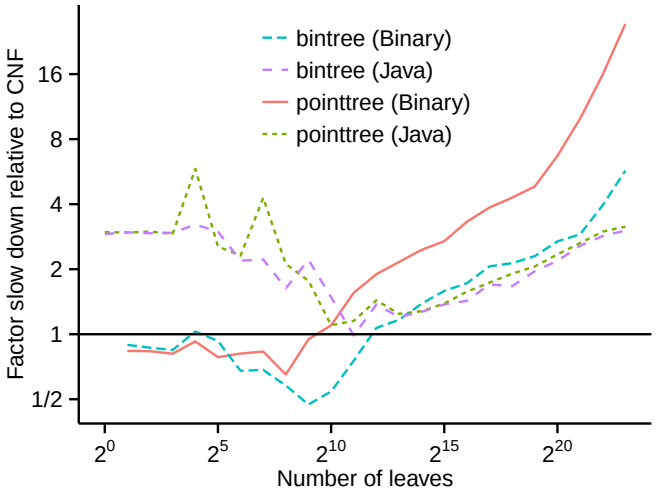


Figure 4: Relative improvement for median end-to-end latency for socket communication with CNFs versus serialization by Haskell binary and Java, for two different data structures bintree and pointtree. Both x and y scales are logarithmic; bigger is better for CNF (and worse for the serializer being compared.) At small sizes, constant factors of CNFs dominate.

Type	Size	Compact	Binary	Java
bintree	2^{23} leaves	3.180 s	18.18 s	9.595 s
	2^{20} leaves	382.4 ms	1.028 s	837 ms
	2^{17} leaves	59.93 ms	109.1 ms	90 ms
pointtree	2^{23} leaves	4.978 s	136.1 s	15.71 s
	2^{20} leaves	624.0 ms	4.181 s	1.461 s
	2^{17} leaves	81.31 ms	354.0 ms	141 ms

Table 2: Median end-to-end latency for socket communciation with CNFs versus serialization by Haskell binary and Java, for the different data structures bintree and pointtree.

Figure 4 shows the relative improvement for end-to-end latency compact normal forms achieve relative to existing solutions for binary and Java. We see that for low tree sizes, constant factors and the overall round trip time of the network dominate; however, as data gets larger serialization cost dominates and our network performance improves.

5.4 Persistence: Memory-mapped Files

While communicating messages between machines is the main use case we've discussed, it's also important to send messages through time, rather than space, by writing them to disk. In particular, not all on-disk storage is meant for archival purposes—sometimes its transient, for caching purposes or communicating data between phases of an application. In Map-Reduce jobs, data is written out between rounds. Or in rendering pipelines used by movie studies, all geometry and character data is generated and written to disk from an earlier phase of the pipeline, and then repeatedly shaded in a later stage of the pipeline. For these use cases, storing in Compact format directly on disk is a feasible alternative.

Here we consider a scenario where we want to process the twitter data set discussed previously. The original data-set is stored on-disk in JSON format, so the natural way to process it would be to read that JSON. For this purpose, the standard approach in Haskell

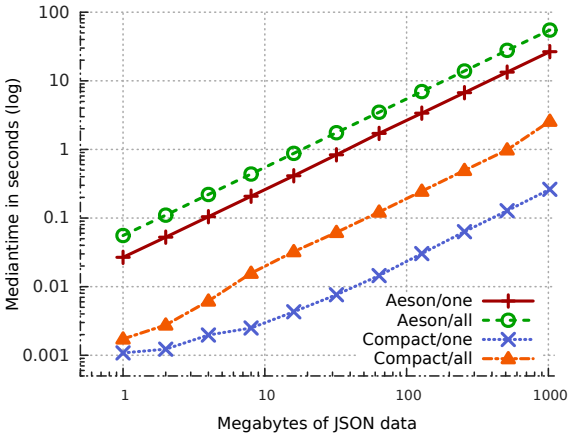


Figure 5: Time spent to load N megabytes of Twitter metadata to access respectively one item at random or process all items sequentially, when loading the JSON directly with Aeson compared to loading a preprocessed Compact file from disk.

would use the efficient Aeson library[2]. We use `Data.Aeson.TH` to derive instances which parse the on-disk format to the in-memory format shown in Figure 2.

The first scenario we consider requires reading *full* dataset through memory, in particular we count how many occurrences of the "cat" hashtag occur in the dataset, while we vary the size of the dataset read from 1MB to 1024MB. "Aeson/all" in Figure 5 shows the result. Reading the full gigabyte takes substantial time—55 seconds. "Compact/all" shows an alternative strategy. We cache a Compact representation on disk, using a format where each block is a separate file. We can then mmap these blocks directly into RAM upon loading, and allow the OS to perform demand paging whenever we access the data. At the full 1GB size, this approach is 21.3× faster than using Aeson to load the data.[3]

Finally, we also consider a *sparse* data access strategy. What if we want to read a specific tweet from the *middle* of the data set? This scenario measured in the "/one" variants of Figure 5. Here, we still map the entire Compact into memory. But the OS only needs to load data for the specific segments we access, no matter where they fall. As a result Compact/one still increases linearly (time for system calls to map $O(N)$ blocks), but the gap widens substantially between it and Aeson/one. The traditional parsing approach must parse half of the data set to reach the middle, resulting in 26.6 seconds to access a tweet in the middle of the 1GB dataset, rather than 0.26 seconds for Compact.

5.5 Garbage Collection Performance

One of the stated benefits of compact normal forms is that objects in a compact region do not have to be traced. Unfortunately, we cannot in general give an expected wall clock improvement, since the specific benefit in an application depends on what data is converted to live in a compact region. Additionally, not all data is suitable for placement in a compact region: if a data structure is rapidly changing compact regions will waste a lot of memory storing dead data.

To give a sense of what kinds of improvements you might see, we constructed a few synthetic benchmarks based on patterns

[2] `https://hackage.haskell.org/package/aeson`

[3] We were not able to disable disk-caching on the evaluation platform (requiring root access), but we report the median result of 5 trials for all data points.

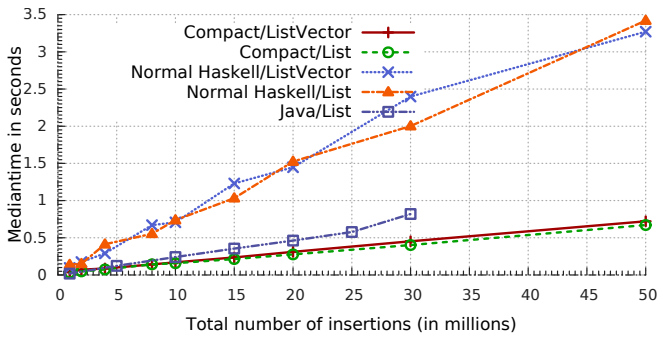


Figure 6: Median time for 16 threads to complete each $N/16$ insertions in 16 lists, where the lists are owned by the threads separately or are referenced by a shared `Vector (IORef [a])`. We can see that in normal Haskell times are influenced by GC pauses, which are greatly reduced for Compacts, despite the need to copy upfront. Java is included as a comparison, to show that Compact can improve performance even against a well tuned fast parallel GC.

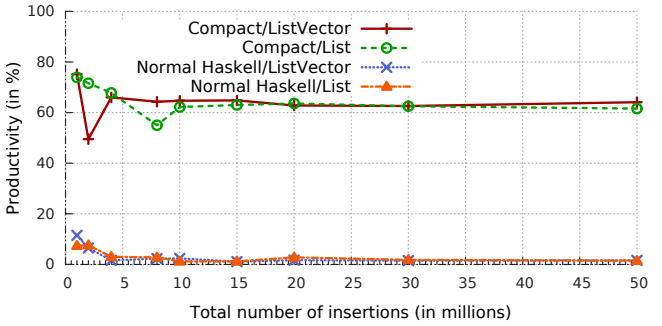


Figure 7: Percentage of CPU time spent in the mutator (as opposed to GC) for 16 threads to complete each $N/16$ insertions in 16 lists, showing the increasing effect of tracing long lived data. Despite using a generational algorithm, the effect of major GCs is so prominent in normal Haskell that only a small fraction of time is spent in the real computation.

we've seen in workloads where garbage collector performance is influential:

- p threads concurrently allocate a list of elements into a compact region. This is a baseline showing the *best-case* improvement, since no elements become dead when a new cell is consed onto a list.
- p threads concurrently allocate a list of elements, but rooted in a single vector. This is meant to illustrate an example where adding a compact region could help a lot, since GHC's existing parallel garbage collector scales poorly when the initial roots are not distributed across threads.

In all of these experiments, the data allocated by each thread is kept live until the end of the test run, simulating *immortal* data which is allocated but never freed.

In Figure 6 we can see the improvement in median running time for these two experiments when the operations happen for a list that lives in a compact region as opposed to the normal heap, while in Figure 7 we can observe the influence of GC in the overall time, which is greatly reduced in the compact case, allowing a more efficient use of resources.

One observation from these experiments is that it is important that the most or all of the existing compact data structure is reused by the mutator — otherwise, the excessive copies into the compact region of soon to be unused data become predominant in the total cost.

Additionally, the double allocation factor introduces memory pressure that triggers more garbage collections: while GC is faster in presence of compact regions, minor collections have to trace the new temporary objects that are allocated prior to copying into the compact region, and that is an added cost if the objects are short lived.

One way to overcome this limitation is to copy the data into a new compact region after a certain number of updates, just like a copying GC would do, such that the amount of unused values is always limited. In our current implementation this is a manual process and relies on the programmer to know the space complexity of the data structure being updated as well as the access patterns from the application (possibly with the help of profiling), but future work could explore efficient heuristics to automate this.

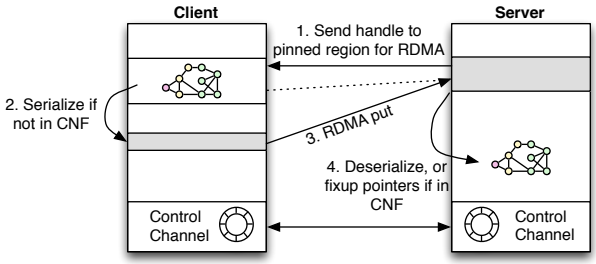
Conversely, it may be interesting to observe that because the GC does not trace the internals of compacts, the GC pauses are less dependent on the layout of the data in memory and how it was computed, making them not only shorter but also more predictable for environments with latency constraints.

5.6 Zero-copy Network Transfer using RDMA

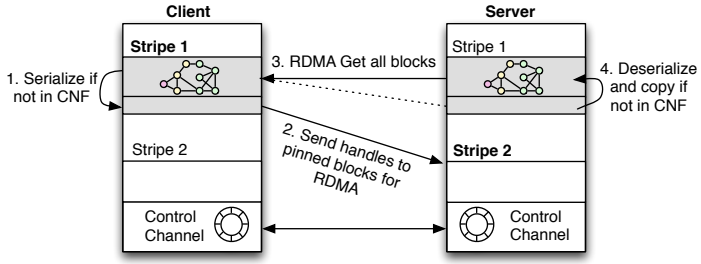
High-performance computing environments as well as large data centers typically comprise of a cluster of tightly-coupled machines networked using low-latency, high-throughput, switched fabrics such as Infiniband or high-speed Ethernet. Remote Direct Memory Access (RDMA) enables a source machine to remotely access a destination machine's memory without any active participation from the latter. In essence, RDMA decouples *data movement* from *synchronization* in communication between hosts. RDMA-enabled network hardware is set up to access a remote processor's memory without involving the operating system on either end. This eliminates synchronization overheads and multiple redundant copies, achieving the lowest possible latency for data movement.

The promise of fast, low-latency RDMA communication, however, is often thwarted by pragmatic issues such as explicit buffer management and synchronization, and the fact that RDMA APIs are low-level and verbose to program with. In contemporary RDMA networking hardware, a host application is required to *pin* the memory that it wants to expose for transfers. The operating system populates page table entries (PTE) associated with this pinned buffer such that all subsequent accesses to memory bypass the OS (the Network Interface Card (NIC) can directly DMA to or from the locked memory). Futher, a source machine requires a *handle* to the remote memory that it wants to access. Thus, there is often a rendezvous required between peers before they can communicate with each other.

Modern high-performance communication libraries offer several features built on top of the raw RDMA API to ease message passing over the network. Each peer reserves pre-pinned ring buffers for every other peer, which are used for transferring small messages. A peer maintains an approximate pointer into a eager ring buffer which is used as the index into remote memory. When a peer suspects that it might overflow the remote buffer, it reclaims space by synchronizing with the remote peer. Large messages are sent by sending a handle to the memory, and requesting the target to *get* the memory associated with the handle. In addition to raw remote memory access (RMA), message passing libraries also pro-



(a) Eager (push-based) RDMA protocol. Here the client wants to send a tree data structure to the server. This approach eliminates the initial rendezvous before communication at the expense of copying into a pre-pinned region on the server.



(b) Rendezvous (pull-based) RDMA protocol. This is the zero-copy case where the client sends metadata of the tree to the server. The server pulls data using remote read into a stripe that it has reserved for the client so that no pointer fixups are required.

Figure 8: The two RDMA data transfer schemes that were used for sending a tree from a client to the server.

vide an RPC mechanism for invoking handlers on the transfered remote data.

We have already discussed the interaction of CNFs with network communication, and demonstrated the purported performance improvements in Section 5.3. Here we consider true zero-copy transfer of heap objects between two networked machines. The two cases that we evaluated are shown in Figures 8a and 8b.

Consider a case where a client wants to send a pointer-based data structure to the server. With RDMA, the client needs to know where to *put* the data in the server's memory. In the approach demonstrated in Figure 8a that we refer to as the eager (push-based) protocol, the server sends a handle to a pinned region in its memory per a client's request. The client has to serialize the data structure into a contiguous memory region if the structure is not in CNF. The client puts into remote memory and notifies the server of completion. All of the protocol messages are exchanged over a control channel also implemented on top of RDMA using eager ring buffers. Finally, the server deserializes the received structure incurring an extra copy and the penalty of fixing up internal pointers if the structure is in CNF.

In the rendezvous (pull-based) zero-copy case shown in Figure 8b, both client and server applications use the striped allocation scheme described earlier. The client has a fixed symmetric memory region (stripe) corresponding to the client in it's virtual address space. The client sends the metadata of the structure (pointer to all of the blocks) that it wants to send to the server. In the normal case, this would mean pinning each node in the tree and sending its address to the server. Fortunately, for us, a Compact is internally represented as a list of large blocks, and thus incurs significantly lower metadata exchange overhead. The server finally gets all of the blocks directly into the right addresses eliminating the need for any extraneous copies. Essentially, with this scheme, we turn all of the RDMA puts into gets, and eliminate an additional round-trip between the client and server.

The RDMA benchmarks were run over the 40Gbpbs QDR Infiniband interconnect through a Mellanox ConnectX-3 EN HCA. For these experiments, we used the Common Communication Interface (CCI)[4] RDMA library over Infiniband. We varied the depth of the tree and its data type as in the previous sections. Both protocols discussed above were implemented and the median time of each phase: tree generation, serialization, communication, deserialization was measured. At higher tree depths, the metadata for each tree is several MBs, which bogs down the ring-buffer based control channel. We implemented the metadata exchange through a pre-pinned metadata segment (as discussed in the Eager scheme) for both protocols.

[4] `http://cci-forum.com/`

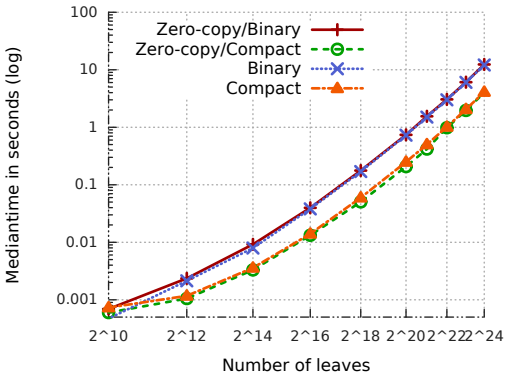


Figure 9: Median time it takes to send a `bintree` of varying tree depths from a client to the server using RDMA. At depth=26, it takes 48s to send a 4GB tree for `Binary` (for a throughput of 85MB/s), whereas it takes 15s for a 2.5GB `Compact` tree (for a throughput of 170MB/s).

As shown in Figure 9, we see upto 5x speedup with `Compact` over `Binary` due entirely to the elimination of serialization overhead even when `Compact` has to transfer upto 5 times more data. However, we found that the time for deserialization of `Binary` was lower than the time required to fixup pointers for `Compact`.

The volume of data transfer is more in the zero-copy case as clients need to exchange metadata with the server. Furthermore, the size of each message is restricted by the maximum block size in the `Compact`. However at larger message sizes, we still expect to see performance improvements for zero-copy over the push variant for two reasons: first, an extra copy is eliminated and secondly, for large messages the cost of deserialization trumps the cost of additional data transfer. Figure 10 confirms our hypothesis. For `Binary`, we mostly see a slowdown except for tree depths above 25 as the cost of deserialization is never amortized by the

5.7 Case study: In-memory key-value store

We implemented a simple in-memory key-value store in Haskell, using compact regions to store values. Remote clients make requests upon the server to fetch the value at a given key. One possible implementation might store the table in a of map from `Key`

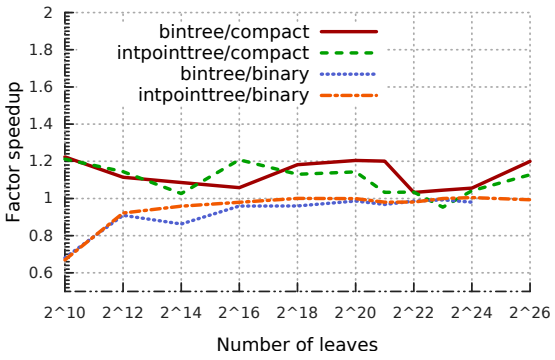


Figure 10: Factor speedup relative to the Eager (push-based) protocol discussed in Figure 8a that incurs a copy and/or pointer fixup overhead. Upto 20% speedup was observed for zero-copy rendezvous transers with `Compact` trees.

Keys	DB size	Binary	Compact
100	6.56 MB	17,081	69,570
1,000	65.6 MB	15,771	63,285
10,000	656 MB	15,295	57,008

Table 3: Requests handled by server for varying database sizes. The size corresponds to the space used by values in the Haskell heap.

`ByteString`, where the `ByteString` represents the serialized payload which should be sent in response. However, this is only workable if the server will *only* service fetch requests—the original values would be gone from memory, and could only be retrieved by deserializing the `ByteString` values. Of course, this deserialization is costly and doesn't support small, incremental updates to the values in question.

Alternatively, the implementor could choose to store `Map Key Val` directly, and opt to serialize on every fetch request. Leveraging lazy evaluation, it would be an elegant optimization to instead store `Map Key (Val,ByteString)`, where the `ByteString` field is a thunk and is computed (and memoized) only if it is fetched by a client. Yet this option has its own drawbacks. The ByteString still needs to be recomputed in whole for any small modification of `Val`, and, further, the entire in-memory store now uses up to twice as much resident memory!

Using Compacts can improve both these problems, while keeping GC pressure low. If we store a `Map Key (Compact Val)` then (1) the value is ready to send at any point, (2) we are able to incrementally add to the compact without recreating it, and (3) the values are always in a "live" state where they support random access at any point within the Compact.

To test this approach, we built a TCP based server running on our evaluation platform. We evaluate this server in terms of client requests per second, using fixed-size values on each request (pointtree of depth 10, size 65.6KB). We use 16 client processes, each launching requests in a sequential loop, to saturate the server with as many requests as the 10G Ethernet network supports. In Table 3, we show how varying the size of the in-memory database changes request handling throughput, by changing the behavior of the memory system. Here we compare our Compact-based server against the `Map Key Val` solution, again using the Binary package for serializing `Val`, showing an increased throughput across a range of in-memory key-value store sizes.

RRN: Metric: what is the expected latency of a request to (1) fetch a tree at a given key, and (2) perform an insert on a tree at a given key.

RRN: different-binary is important here because we won't be able to maintain the same

6. Related Work

The idea of reusing an in-memory representation for network and disk communication originates from the very beginning of computing. It was common for old file formats to involve direct copies of in-memory data structures [24, 25, 28], in order to avoid spending CPU cycles serializing and deserializing in a resource constrained environment. More recently, libraries like Cap'n Proto [30] advocate in-memory representations as a general purpose binary network interchange format.

These applications and libraries are almost universally implemented in langauges with manual memory management. However, there are some shared implementation considerations between these applications and compact normal forms. The literature on pointer swizzling [16, 31], for example, considers how pointers should be represented on disk. The idea of guaranteeing a structure is mapped to the same address occurs in many other systems. [4, 5, 27]

On the other hand, it is far less common to see this technique applied in garbage collected languages. One common mode of operation was to save the entire heap to disk in an image, so that it could be reloaded quickly; schemes like this were implemented in Chez Scheme and Smalltalk. Our goal was to allow manipulating *subsets* of the heap in question.

The one system we are aware of which organizes heap objects into regions in the same way is Expresso [8] for Java. Like our system, Expresso allocates heap objects into contiguous chunks of memory which can then be transmitted to other nodes. However, while there are similarities in the underlying implementations, our API is substantially safer: Expresso is implemented in a language which supports mutation on all objects, which means that there is no invariant guaranteeing that all pointers are internal to a compact block. Compact Normal Forms do have this invariant, which means we can optimize garbage collection and avoid dangling pointers. Some other work [6] send the literal format but don't try to maintain a contiguous representation; thus a traversal is still necessary as part of the serialization step.

There are a number of systems address related problems to ours, we now comment on them below.

Message passing in distributed computation There is a lot of prior work in systems for distributed computation. The Message Passing Interface (MPI) is the standard for message communication in many languages, and emphasizes avoiding copying data structures. However, MPI assumes that data lives in a contiguous buffer prior to sending: it is up to the high-level language to arrange for this to be the case.

Message passing implementations in high-level languages like Java and Haskell are more likely to have a serialization step. Accordingly, there has been some investigation on how to make this serialization fast: Java RMI [26], for example, improved serialization overhead by optimizing the serialization format in question. However, except in the cases mentioned in the previous section, most serialization in these languages doesn't manage to achieve zero-copy.

It is worth comparing our API to existing message passing APIs in distributed Haskell systems. Our approach is more in keeping with previous systems like Eden [3], where the system offers built-in support for serializing fully evaluated, non-closure data. Cloud Haskell [10], on the other hand attempts to support the transmis-

sion of higher-level functions with a combination of extra language techniques. Like Cloud Haskell, our system works best if identical Haskell code is distributed to all nodes, although we can accommodate (with performance loss) differing executables.

Regions and garbage collection It is folklore that in the absence of mutable data, generational garbage collection is very simple, as no *mutable set* must be maintained in order that a backwards pointer from the old generation to the new generation is handled properly. In this sense, a compact region is simply a generalization of generational garbage collection to have arbitrarily many tenured generations which are completely self-contained. This scheme bears similarity to distributed heaps such as that in Singularity [14], where each process has a local heap that can be garbage collected individually. Of course, the behavior of data in a compact region is much simpler than that of a general purpose heap.

The idea of collecting related data into regions of the heap has been explored in various systems, usually in order to improve data locality. [1, 7, 11] At the static end of the spectrum, *region* systems [12, 29] seek to organize dynamically allocated data into regions which can be freed based on static information, eliminating the need for a tracing garbage collector. MLKit [13] combines region inference with tracing garbage collection; their garbage collection algorithm for regions bears some similarities to ours; however, since we don't trace data living in a region, our algorithm is simpler at the cost of space wastage for objects in a compact region which become dead—a tradeoff which is also familiar to region systems.

7. Conclusions

In programming languages, abstraction is naturally sometimes at odds with performance, especially with regards to garbage collection versus manual memory management. In this paper, we have tried to show how compact regions can be a semantically simple primitive that still brings good performance benefits to the table. We believe this sort of region management may prove to be a practical compromise for managing heap layout, just as semi-explicit parallelism annotations have proven a useful compromise.

Acknowledgments

Edward Z. Yang is supported by the DoD through the NDSEG. Support for this work was also provided by NSF grant XPS-1337242.

References

[1] E. D. Berger, B. G. Zorn, and K. S. McKinley. Reconsidering custom memory allocation. *ACM SIGPLAN Notices*, 37:1, 2002.

[2] R. L. Bocchino Jr, V. S. Adve, D. Dig, S. V. Adve, S. Heumann, R. Komuravelli, J. Overbey, P. Simmons, H. Sung, and M. Vakilian. A type and effect system for deterministic parallel Java. *ACM Sigplan Notices*, 44(10):97–116, 2009.

[3] S. Breitinger, U. Klusik, and R. Loogen. From (sequential) Haskell to (parallel) Eden: An Implementation Point of View. *Symposium on Programming Language Implementation and Logic Programming - PLILP*, pages 318–334, 1998.

[4] J. Chase, F. Amador, E. Lazowska, H. Levy, and R. Littlefield. The Amber system: parallel programming on a network of multiprocessors. *ACM SIGOPS Operating Systems Review*, 23(December 1989):147–158, 1989.

[5] J. S. Chase, H. M. Levy, E. D. Lazowska, and M. Baker-Harvey. Lightweight shared objects in a 64-bit operating system, 1992.

[6] S. Chaumette, P. Grange, B. Métrot, and P. Vignéras. Implementing a High Performance Object Transfer Mechanism over JikesRVM. 2004.

[7] T. M. Chilimbi and J. R. Larus. Using generational garbage collection to implement cache-conscious data placement, 1999.

[8] L. Courtrai, Y. Maheo, and F. Raimbault. Expresso: a Library for Fast Java Objects Transfer. In *Myrinet User Group Conference (MUG)*, 2000.

[9] D. Coutts, R. Leshchinskiy, and D. Stewart. Stream fusion. from lists to streams to nothing at all. In *ICFP07*, 2007.

[10] J. Epstein, A. P. Black, and S. Peyton-Jones. Towards Haskell in the cloud. *ACM SIGPLAN Notices*, 46(Section 4):118, 2012.

[11] T. D. S. Gene Novark. Custom Object Layout for Garbage-Collected Languages. *Techreport*, 2006.

[12] D. Grossman, G. Morrisett, T. Jim, M. Hicks, Y. Wang, and J. Cheney. Region-based memory management in Cyclone. In *PLDI*, 2002.

[13] N. Hallenberg, M. Elsman, and M. Tofte. Combining region inference and garbage collection. In *PLDI*, volume 37, page 141, May 2002.

[14] G. C. Hunt and J. R. Larus. Singularity: rethinking the software stack. *SIGOPS OSR*, 2007.

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

[16] A. Kemper and D. Kossmann. Adaptable pointer swizzling strategies in object bases: design, realization, and quantitative analysis. *The VLDB Journal*, 4:519–566, 1995.

[17] R. Lämmel and S. P. Jones. Scrap your boilerplate with class: Extensible generic functions. *SIGPLAN Not.*, 40(9):204–215, Sept. 2005.

[18] J. Launchbury. A natural semantics for lazy evaluation. pages 144–154. ACM Press, 1993.

[19] R. Loogen, Y. Ortega-mallén, and R. Peña marí. Parallel functional programming in Eden. *J. Funct. Program.*, 15(3):431–475, May 2005.

[20] P. Maier and P. Trinder. Implementing a high-level distributed-memory parallel haskell in haskell. In *Implementation and Application of Functional Languages*, pages 35–50. Springer, 2012.

[21] S. Marlow, T. Harris, R. P. James, and S. Peyton Jones. Parallel generational-copying garbage collection with a block-structured heap. In *Proceedings of the 7th international symposium on Memory management*, ISMM '08, pages 11–20, New York, NY, USA, 2008. ACM.

[22] K. McKelvey and F. Menczer. Design and prototyping of a social media observatory. In *Proceedings of the 22nd international conference on World Wide Web companion*, WWW '13 Companion, pages 1351–1358, 2013.

[23] K. McKelvey and F. Menczer. Truthy: Enabling the Study of Online Social Networks. In *Proc. 16th ACM Conference on Computer Supported Cooperative Work and Social Computing Companion (CSCW)*, 2013.

[24] Microsoft Corporation. Word (. doc) Binary File Format. Technical report, 2014.

[25] Microsoft Corporation. Bitmap Storage, 2015.

[26] C. Nester, M. Philippsen, and B. Haumacher. A more efficient RMI for Java. In *Proceedings of the ACM 1999 conference on Java Grande*, pages 152–159. ACM, 1999.

[27] E. Shekita and M. Zwilling. Cricket: A Mapped, Persistent Object Store. 1996.

[28] TIS Committee. Tool Interface Standard (TIS) Executable and Linking Format (ELF) Specification (Version 1.2). Technical Report May, 1995.

[29] M. Tofte, L. Birkedal, M. Elsman, and N. Hallenberg. A retrospective on region-based memory management. *Higher-Order and Symbolic Computation*, 17(3):245–265, 2004.

[30] K. Varda. Cap'n Proto, 2015.

[31] P. Wilson and S. Kakkad. Pointer swizzling at page fault time: efficiently and compatibly supporting huge address spaces on standard hardware. *[1992] Proceedings of the Second International Workshop on Object Orientation in Operating Systems*, 1992.