Work Efficient Higher-Order Vectorisation

Ben Lippmeier[†] Manuel M. T. Chakravarty[†] Gabriele Keller[†] Roman Leshchinskiy[†]
Simon Peyton Jones[‡]

[†]Computer Science and Engineering
University of New South Wales, Australia
{benl,chak,keller,rl}@cse.unsw.edu.au

[‡]Microsoft Research Ltd
Cambridge, England
{simonpj}@microsoft.com

Abstract

Existing approaches to *higher-order vectorisation*, also known as *flattening nested data parallelism,* do not preserve the asymptotic work complexity of the source program. Straightforward examples, such as sparse matrix-vector multiplication, can suffer a severe blow-up in both time and space, which limits the practicality of this method. We discuss why this problem arises, identify the mis-handling of index space transforms as the root cause, and present a solution using a refined representation of nested arrays. We have implemented this solution in Data Parallel Haskell (DPH) and present benchmarks showing that realistic programs, which used to suffer the blow-up, now have the correct asymptotic work complexity. In some cases, the asymptotic complexity of the vectorised program is even better than the original.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features—Concurrent programming structures; Polymorphism; Abstract data types

General Terms Languages, Performance

Keywords Arrays, Data parallelism, Haskell

1. Introduction

Data Parallel Haskell (DPH) is an extension to the Glasgow Haskell Compiler (GHC) that offers *nested data parallelism*. With nested parallelism, each parallel computation may spawn further parallel computations of arbitrary complexity, whereas with flat parallelism, they cannot; so nested data parallelism is vastly more expressive for the programmer. On the other hand, flat data parallelism is far easier to implement, because flat data parallelism admits a simple load balancing strategy and can be used on SIMD hardware (including GPUs). The *higher-order vectorisation* (or *flattening*) transform [17] bridges the gap, by transforming source programs using *nested* data parallelism into ones using just *flat* data parallelism [1, 17]. That is, it transforms the program we want to write into the one we want to run.

Unfortunately, practical implementations, including ours, have had a serious flaw: the standard transformation only guarantees to preserve the parallel *depth complexity* of the source program, and not its asymptotic *work complexity* as well. If our benchmark machines had an infinite number of processors, this would be of no concern, but alas they do not. Nor is this phenomenon rare: while working on DPH we have encountered simple programs that suffer a severe, and sometimes even exponential, blow-up in time and space when vectorised.

This is a well-known problem that arises due to the flat representation of nested arrays in vectorised code [3, Appendix C]. Several attempts have been made to solve it, but so far they have been either incomplete [16], do not work with higher order languages [10], or give up on flattening the parallelism [4, 8] or arrays [18] altogether. In this paper, we will show how to overcome the problem for full-scale higher-order vectorisation. Overall, we make the following contributions:

1. We present the first approach to higher-order vectorisation that, we believe, ensures that vectorised programs maintain the asymptotic work complexity of the originals, while allowing nested arrays to retain their flattened form (§4). We only require that vectorised programs are *contained* [2, 18], a property related to the standard handling of branches in SIMD-style parallel programming (§5.6).
2. We identify the key problem of mishandled index space transforms, which worsen the asymptotic complexity of vectorised code using prior flat array representations (§3).
3. We introduce a novel delayed implementation of the central index space transforms (§4) and discuss the pragmatics of achieving good constant factors, in addition to the required asymptotic performance (§6).
4. Finally, we present performance figures for several realistic programs, including the Barnes-Hut *n*-body algorithm. This supports our claim that our delayed implementation of the index space transforms leads to vectorised programs that operate within the required asymptotic bounds (§7).

The claim that our new approach to higher-order vectorisation is work efficient is supported by experiments with a concrete implementation in GHC — but not yet by formal proof, which we leave to future work. Nevertheless, our work presents a significant advance of the state of the art on a long-standing problem. Achieving good *space* complexity is an orthogonal problem that we discuss in §5.5. A reference implementation of our new array representation is available in the companion technical report [14].

2. The Asymptotic Complexity Problem

We start with an example illustrating vectorisation. The function `retrieve` simultaneously indexes several arrays, the `xss`, each of which is shared across one subarray of indices contained in `iss`. It returns a nested array of the results and uses nested parallelism — an inner parallel computation (`mapP indexP xss`) is performed for each of the outer ones.

```
retrieve :: [:[:Char:]:] -> [:[:Int:]:] -> [:[:Char:]:]
retrieve xss iss
  = zipWithP mapP (mapP indexP xss) iss
```

Here is `retrieve` applied to two example arrays.[1]

```
retrieve [[A B]   [C D E] [F G] [H]]       (xss)
         [[1 0 1] [2]     [1 0] [0]]       (iss)
     ==> [[B A B] [E]     [G F] [H]]
```

In the type signature, `[:Char:]` refers to *bulk-strict, parallel, one-dimensional arrays*. Elements of these arrays are stored unboxed, so that demanding any element causes them all to be computed. `zipWithP` and `mapP` are parallel versions of the corresponding list functions, while `indexP` is array indexing — Figure 1 shows these and other typical array operations. The work-complexity of `retrieve` is linear in the number of leaf elements of the array `iss` (seven here), since each is used once for indexing. (Technically it is also linear in the number of sub-arrays in `iss`, since empty arrays in `iss` would still cost.)

The vectorised form of `retrieve` is the following — the accompanying technical report [14] includes the full derivation.

```
retrieve_v :: PA (PA Char) -> PA (PA Int) -> PA (PA Char)
retrieve_v xss iss
 = let ns = takeLengths iss
   in  unconcat iss
        $ index_l (sum ns) (replicates ns xss)
        $ concat iss
```

The type `PA` is an generic representation type that determines the layout of the user-visible type `[::]` in a type-dependent manner [6]. When applied to our example array, the function first concatenates `iss` to yield a flat array of indices, and uses `takeLengths` to get the lengths of the inner arrays of `iss`:

```
ns   = takeLengths iss = [3      1  2    1]
iss1 = concat iss      = [1 0 1  2  1 0  0]
```

The `replicates` function distributes the subarrays of `xss` across the flat indices array. It takes an array of replication counts and an array of elements, and replicates each element by its corresponding count:

```
xss1 = replicates ns xss
     = replicates [3 1 2 1] [[A B] [C D E] [F G] [H]
     = [[A B] [A B] [A B] [C D E] [F G] [F G] [H]]
```

Now we have one sub-array for each of the elements of `iss1`. Continuing on, we use the *lifted indexing* operator `index_l`, which has the following type:

```
index_l :: Int -> PA (PA e) -> PA Int -> PA e
```

Given an array of arrays, and an array of indices of the same length, for each subarray-index pair, `index_l` retrieves the corresponding element of the array. In other words, `index_l` is effectively `zipWithP indexP`, except that it gets the length of the two arrays as an additional first argument.

[1] The concrete syntax for array literals is `[:x1, ..., xn:]`. To save space, we elide the colon and comma.

```
lengthP      :: [:e:] -> Int
indexP, (!:) :: [:e:] -> Int -> e
concatP      :: [:[:e:]:]     -> [:e:]
mapP         :: (d -> e)      -> [:d:] -> [:e:]
zipWithP     :: (c -> d -> e) -> [:c:] -> [:d:] -> [:e:]
foldP        :: (e -> e -> e) -> [:e:] -> e
```

Figure 1. User Visible Array Operators

```
data PA e = PA {length :: Int, pdata :: PData e}
data family   PData e
data instance PData Int    = PInt  (Vector Int)
data instance PData Char   = PChar (Vector Char)
data instance PData (PA e) = PNested Segd (PData e)
data Segd = Segd {lengths, indices :: Vector Int}
index       :: PA e      -> Int      -> e
index_l     :: Int       -> PA (PA e) -> PA Int -> PA e
replicate   :: Int       -> e        -> PA e
replicates  :: PA Int    -> PA e     -> PA e
concat      :: PA (PA e) -> PA e
unconcat    :: PA (PA e) -> PA e     -> PA (PA e)
```

Figure 2. Baseline Array Representation and Parallel Primitives

Applying `index_l` to our example yields the following:

```
xss2
 = index_l (sum ns) (replicates ns xss) (concat iss)
 = index_l 7 [[A B] [A B] [A B] [C D E] [F G] [F G] [H]]
             [1     0     1     2       1     0     0]
 = [B A B E G F H]
```

Finally, we use `unconcat` to reapply the original nesting structure to this flat result:

```
xss3 = unconcat [[1 0 1] [2] [1 0] [0]] [B A B E G F H]
     = [[B A B] [E] [G F] [H]]
```

In the vectorised function `retrieve_v`, all parallelism comes from the implementation of the primitive flat parallel array operators such as `index_l` and `replicates`. However, simply converting nested parallelism to flat parallelism is not sufficient. We previously implemented `replicates` by physically copying each of the subarrays. With that implementation, suppose we evaluate the following expression:

```
retrieve [[A B C D E F G H]] [[0 1 2 3 4 5 6 7]]
```

In terms of the source program, this expression takes eight steps, one for each index in the second array. However, in the vectorised program, `replicates` will also copy `[A B C D E F G H]` eight times. As we have the same number of characters in the first array as indices in the second array, vectorisation turned a function that performs $O(n)$ work into an $O(n^2)$ function: Disaster!

It turns out that the trouble with `replicates` is just one of a class of problems related to the mishandling of index space transforms during vectorisation. These transforms change the mapping between elements in the source and result arrays, but do not compute new element values. In addition to identifying index space transforms as the culprit, in the next two sections we contribute a novel delayed implementation, which enables vectorised programs to remain within the required asymptotic complexity bounds. What are those bounds? Consider an absolutely *direct* implementation of DPH, in which a value of type `[:a:]` is represented by an ordinary array of pointers to values of type `a`.

> **Complexity Goal:** for the output of vectorisation to have the same *asymptotic work complexity* as the direct implementation, but with much better *constant factors* and *amenability to parallelism.*

3. Baseline Representation of Nested Arrays

A key idea of Blelloch's vectorisation transformation is to flatten the representation of nested arrays, as well as the parallelism itself. More precisely: an array A of sub-arrays $A_0, A_1, ..., A_{n-1}$ (each with its own length) is represented by (a) a single long array of data, $D = [A_0, A_1, ..., A_{n-1}]$ all laid out in one contiguous block, and (b) a *segment descriptor* that gives the length of each A_i in the data block D. We call A_i the *segments* of A. The idea is to divide the data block D evenly over the processors, and process each chunk independently in parallel. This provides both excellent granularity and excellent data locality, which is intended to satisfy the second part of our Complexity Goal. There is some book-keeping to do on the segment descriptor; generating that book-keeping code is the job of the vectorisation transformation.

Figure 2 gives the representation of nested arrays in Haskell, using GHC's *data families* [5]. An array of type `(PA e)` is represented by a pair `PA n d`, where `n` is the length of the array, and `d :: PData e` contains its data. The representation of `PData` is type-dependent — hence, its declaration as a `data family`. When the argument type is a scalar, matters are simple: `PData Int` is represented merely by a `Vector Int`, which we take as primitive here[2]. Arrays of `Char` are represented similarly. On the other hand, the data component of a *nested array*, with type `PData (PA e)` is represented by a pair of a segment descriptor of type `Segd`, and the *data block* of type `PData e`. The segment descriptor `Segd` has two fields, `lengths` and `indices`. The latter is just the scan (running sum) of the former, but we maintain both in the implementation to avoid recomputing `indices` from `lengths` repeatedly. Each is a flat `Vector` of `Int` values.

Using the example from the previous section, the array `xss1` has type `(PA (PA Char))` and is represented like this:

```
   replicates [3 1 2 1] [[A B] [C D E] [F G] [H]]
   = [[A B] [A B] [A B] [C D E] [F G] [F G] [H]]
 ------------------------------------------- (ARR0)
  PA 7 (PNested
   (Segd lengths: [2 2 2 3 2 2  1]
         indices: [0 2 4 6 9 11 13])
   (PChar [A B A B A B C D E F G F G H]))
```

We show the *logical value* of the array above the line, and its *physical representation* below. The representation is determined by the data type declarations in Figure 2. The result array is built with an outer `PA` constructor, pairing its length, 7, with the payload of type `PData (PA Char)`. From the `data instance` for `PData (PA e)`, again in Figure 2, we see that the data field consists of a `PNested` constructor pairing a segment descriptor with a value of type `PData Char`. Finally, the latter consists of a `PChar` constructor wrapping a flat `Vector` of `Char` values.

The process continues recursively in the case of deeper nesting: the reader may care to write down the representation of a value of type `PA (PA (PA Int))`. We will see an example in §4.4.

Now the problem with `replicates` becomes glaringly obvious. The baseline representation of arrays, which was carefully chosen to give good locality and granularity, is *physically incapable of representing the sharing between subarrays in the result* — and losing that sharing leads directly to worsening the asymptotic complexity. It is not possible to simply eliminate the call to `replicates` itself, because this function plays a critical role in vectorisation. In the example from §2, `replicates` distributes shared values from the context of the outer computation (`zipWithP mapP`) into the inner computation (`mapP indexP`). Since we cannot eliminate `replicates`, the only way forward is to change the representation of nested arrays.

[2] It is provided by the `vector` library.

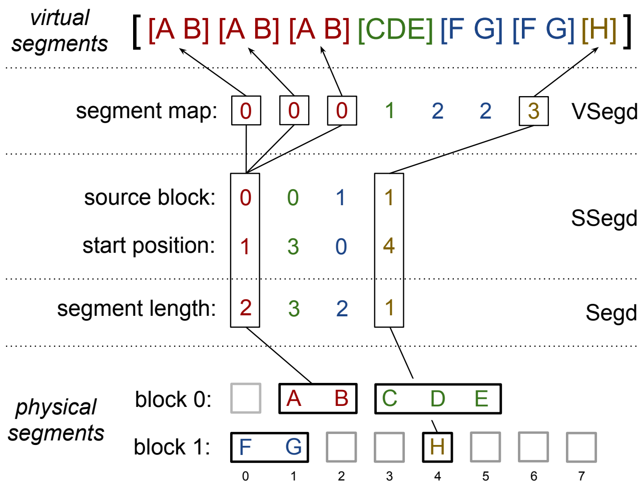

Figure 3. New Array Representation

4. New Representation of Nested Arrays

The new array representation must support all index space transforms that vectorisation introduces, in a way that allows the vectorised program to have the same asymptotic complexity as the original unvectorised program. This is up to *containment* which is discussed in §5.6. In most cases this comes down to having the same complexity as the direct representation, where nested arrays are stored as flat arrays of pointers to more arrays. However, we cannot use this representation as described, because it would lose the granularity and data locality benefits of the baseline segmented representation. We need the best of both worlds.

4.1 Physical, Virtual and Scattered Segments

An example array with the same value as `ARR0` is shown in Figure 3. Our new representation has the following key features:

1. We distinguish between *physical* and *virtual* segments. Physical segments consist of real element data in memory, while virtual segments are defined by mapping onto physical segments. This distinction enables us to define nested arrays with repeated segments without copying element data.
2. The physical segments of a nested array may now be scattered through several data blocks, instead of being contiguous. Although we prefer segments to be contiguous for locality reasons, we must also allow them to be scattered, so that we can filter a nested array without copying element data.

In the example, there are seven virtual segments defined from four physical segments. The physical segments lie scattered in two data blocks. We will see why we need to allow physical segments to lie in separate data blocks in §4.5. The overall segment descriptor is now stratified into three layers: `VSegd` (virtual segments); `SSegd` (scattered segments) and plain `Segd` (contiguous segments). In our terminology, we refer to all of `VSegd`, `SSegd` and `Segd` "segment descriptors". At the bottom layer, `Segd` gives the length of each segment, and would be sufficient to describe the array if all segments were contiguous in a single block. The `SSegd` gives the index of the source data block, and starting position for each physical segment in its block. The `VSegd` provides the mapping between virtual and physical segments. We have elided the `indices` field from the diagram for clarity, but also include this in our new array representation as part of the `Segd`.

```
data PA e = PA { length :: Int, pdata :: PData e }
data family   PData  e
data family   PDatas e
data instance PData  Int  = PInt   (Vector Int)
data instance PDatas Int  = PInts  (Vector (Vector Int))
data instance PData  Char = PChar  (Vector Char)
data instance PDatas Char = PChars (Vector (Vector Char))

data instance PData (PA e)
 = PNested { vsegd :: VSegd, pdatas :: PDatas e }

data instance PDatas (PA e)
 = PNesteds (Vector (PData (PA e)))

data VSegd  -- Virtual-segment descriptor.
 = VSegd { segmap  :: Vector PsId, ssegd :: SSegd }

data SSegd  -- Scattered-segment descriptor.
 = SSegd { sources :: Vector DbId, starts :: Vector Int }
         , segd    :: Segd }

data Segd   -- Contiguous-segment descriptor.
 = Segd  { lengths :: Vector Int, indices :: Vector Int }

type PsId = Int  -- Physical segment Id, indexes 'sources'
type DbId = Int  -- Data block Id, indexes 'pdatas'
```

Figure 4. Definition of the New Array Representation

4.2 The Concrete Definition

The concrete definition of our new array type is given in Figure 4. The data type PA is unchanged from Figure 2: a pair of a length and payload. The instances for PData Int and PData Char are also unchanged. The difference is in the representation of nested arrays:

```
  data instance PData (PA a)
   = PNested { vsegd  :: VSegd, pdatas :: PDatas a }
```

The payload is now a PDatas (plural) rather than PData. Where PData represents a single data block, PDatas represents a vector of data blocks. We use the type DbId (short for data-block identifier) to index this vector of PData values.

The vsegd field holds the *virtual segment descriptor*. It consists of a vector of physical segment identifiers (segmap), and a scattered segment descriptor (ssegd). The segmap maps virtual segments onto physical segments and corresponds 1-1 with the outer level of the array being represented. In Figure 3, we have seven entries in this map and seven subarrays in the overall nested array.

Each entry in the segmap is a *physical segment identifier*, of type PsId. A PsId is the index of one of the physical segments described by the SSegd and Segd types. Crucially, the segmap can contain repeated use of the same physical segment. In Figure 3 we have used [0 0 0 1 2 2 3] to indicate three copies of the first physical segment, one copy of the second, and so on. This is how we represent the sharing defined by replicates. Note that we can not just store the replication counts [3 1 2 1] directly because we must be able to map virtual segments to physical segments in constant time.

The SSegd and Segd together describe the physical segments. Together they contain four vectors, *all of the same length*, two of them nested inside the segd field. The sources vector gives the data block identifier, DbId, which is the index of one of the data blocks in the pdatas field. The next two, starts and lengths, give the starting position and length of the physical segment in that data block. Finally indices is, as before, a cached copy of the scan (accumuated sum) of lengths. Keeping SSegd and Segd separate is helpful when optimising vectorised code for absolute performance, which we discuss in §6.

Finally, the array representation must obey the following invariants:

1. The lengths of the sources, starts, lengths, and indices fields must all be the same.
2. Every PsId in the segmap must be less than the length of the sources field. This ensures that each physical segment identifier points to a real physical segment.
3. Each DbId in sources must be less than the length of the pdatas vector. This ensures that each data block identifier points to a real data block.
4. Each element of starts[i] must be less than the length of pdatas[sources[i]].
5. The indices field is equal to init (scan (+) 0 lengths).
6. All physical segments defined by the SSegd and Segd types must be reachable from the segmap. More precisely, the set of physical segment identifiers in the segmap must cover [0..np-1], where np is the length of the starts, sources, lengths, and indices fields.
7. All pdata blocks must be reachable from the sources field. More precisely, the set of sources must cover [0..nb-1], where nb is the length of the pdatas vector.

Invariants 1 to 4 are standard well-formedness conditions. Invariant 5 says that indices is precomputed from lengths. The reason for this is discussed in §6. Invariants 6 and 7 ensure that the size of the internal structure of the array is bounded by the number of virtual segments, which is necessary for the complexity bound on append (§4.5). Invariant 7 is also needed to ensure that the parallel implementation of reductions such as sum do not duplicate work (§5.4). However, an implementation may be able to relax these last two invariants in certain cases (§6).

4.3 Replicates again

Now let us implement replicates using our new array representation. The start is easy, because the result PA array must be built with a PA constructor:

```
replicates :: Vector Int -> PA e -> PA e
replicates ns arr = PA (sum ns) (replicatesPR ns arr)
```

The real work is in replicatesPR. But now we encounter a slight problem: since the representation of PData is indexed by the element type e, we require a type-indexed function to operate over PData values. That is, we need a type class, with an instance for Int and an instance for (PA e):

```
class PR e where
  replicatesPR :: Vector Int -> PData e -> PData e
  ...more methods...

instance PR Int where
  replicatesPR = replicatesI
  ...
instance PR e => PR (PA e) where
  replicatesPR = replicatesPA
  ...
```

The PR (Parallel Representation) class is given in Figure 5, and conveniently collects all the necessary primitive operations over arrays. We will see more of them in this section, but replicatesPR is one. So, in fact, we lied: the types of replicates and replicatesPR are overloaded thus:

```
replicates   :: PR e => Int -> e -> PA e
replicatesPR :: PR e => Int -> e -> PData e
```

```
class PR e where
emptyPR      :: PData  e
lengthPR     :: PData  e -> Int

replicatePR  :: Int -> e -> PData e
replicatesPR :: Vector Int -> PData e -> PData e

appendPR     :: PData  e -> PData e -> PData e

indexPR      :: PData  e -> Int -> e
indexvsPR    :: PDatas e -> VSegd
             -> Vector (Int, Int) -> PData e

extractPR    :: PData  e -> Int   -> Int -> PData e
extractvsPR  :: PDatas e -> VSegd -> PData e

packPR   :: PData  e -> Vector Bool -> PData e
combinePR:: Vector Bool -> PData e -> PData e -> PData e

lengthdPR    :: PDatas e -> Int
emptydPR     :: PDatas e
singletondPR :: PData  e -> PDatas e
appenddPR    :: PDatas e -> PDatas e -> PDatas e
indexdPR     :: PDatas e -> Int -> PData e

------- Utility functions --------
sumV        :: Vector Int -> Int
singletonV  :: e -> Vector e
replicateV  :: Int -> e -> Vector e
replicatesV :: Vector Int -> Vector e -> Vector e
```

Figure 5. Primitive Array Operators

(In what follows we will often omit the "`PR =>`" context from types to save space.) Now we are ready to implement the two cases. The case for `Int` is straightforward:

```
replicatesI :: Int -> Int -> PData Int
replicatesI c i = PInt (replicatesV c i)
```

where `replicatesV` is the `Vector`-level replication operation shown in Figure 5. The interesting case is the one for nested arrays:

```
instance PR e => PR (PA e) where
  replicatesPR = replicatesPA

 replicatesPA :: Vector Int -> PData (PA e) -> PData (PA e)
 replicatesPA lens (PNested segmap pdatas)
  = PNested (VSegd segmap' ssegd) pdatas
  where segmap' = replicatesV lens segmap
```

With our new array representation, we can apply segmented replicate to an array by using `replicatesV` on the `segmap` field. The element data, `pdatas`, does not need to be copied, and is untouched in the result. Continuing the example from §3, replicating array from Figure 3 again yields the following result:

```
   replicates [0 0 1 1 0 0 1] {Figure 3}
 = [[A B] [C D E] [H]]
 ------------------------------------------------ (ARR1)
  PA 3 (PNested
   (VSegd  segmap: [0 1 3]
   (SSegd sources: [0 0 1 1]  starts: [1 3 0 4])
   (Segd  lengths: [2 3 2 1] indices: [0 2 5 7]))
   (PChars 0: [X A B C D E]
           1: [F G X X H X X X])
```

In fact, the above definition of `replicatesPA` function is not yet complete. Physical segments 0, 1 and 3 are used, but segment 2 is not, which violates invariant 6. We will discuss why this matters in §4.6.

4.4 Plain replicate

Vectorisation also uses a simpler form of replication, which we call `replicate` (singular). The call `(replicate n x)` returns an array of `n` elements, each a (virtual) copy of `x`. This function is introduced when an inner parallel computation uses a shared constant or a free variable that is defined in an outer context. This is essentially the same reason that the more general `replicates` function is introduced, though with plain `replicate` the shared value is used uniformly by all inner computations. We will see an example in §6.1. Note that unlike `replicates`, the result of plain `replicate` has a greater nesting depth than the source element. The interesting case is for nested arrays:

```
replicatePA :: Int -> PA e -> PData (PA e)
replicatePA c (PA n pdata)
 = replicatesPR (singletonV c)
 $ PNested (singletonVSegd n) (singletondPR pdata)

singletonVSegd :: Int -> VSegd
singletonVSegd len
 = VSegd (singletonV 0)
  (SSegd (singletonV 0)   (singletonV 0)
  (Segd  (singletonV len) (singletonV 0)))
```

To perform a `replicate` we simply add a new segment descriptor on top of the old array. This furnishes us with an example array of greater nesting depth:

```
  replicate 2 {Figure 3}
 ------------------------------------------------ (ARR2)
  PA 2 (PNested
   (VSegd  segmap: [0 0]
   (SSegd sources: [0]  starts: [0]
   (Segd  lengths: [7] indices: [0])))
   (PNesteds
    0: PNested
       (VSegd  segmap: [0 0 0 1 2 2 3]
       (SSegd sources: [0 0 1 1]  starts: [1 3 0 4])
       (Segd  lengths: [2 3 2 1] indices: [0 2 5 7]))
       (PChars 0: [X A B C D E]
               1: [F G X X H X X X])
```

Notice that the cost of `(replicate n x)` is $O(\mathtt{n})$, regardless of how much data `x` contains. With our new representation the complexity of `replicate` is linear in the length of the created `segmap`, which is also the length of the overall array.

4.5 Append

Let us consider another important operation: appending two arrays.

```
  appendPA :: PA e -> PA e -> PA e
```

As mentioned in §4.2 we need invariants 6 and 7 to achieve the Complexity Goal here. Append should be linear in the length of the two argument arrays, regardless of how deeply nested they are. This is impossible with the baseline representation from §3 because we would need to copy all elements into a single data block.

With our new representation we do not need to copy array elements. To append two nested arrays we append the two `PDatas` and combine the segment descriptor fields. Although we can simply append the `lengths` and `starts` fields, we need to recompute the `indices`. We also need to increment the entries in the second `segmap` and `sources` field to account for the physical segments and data blocks defined by the first array. For this process to have complexity linear in the length of the two argument arrays, the lengths of their `starts`, `sources`, `lengths` and `indices` fields can be no greater than the length of their `segmap`. To put this another way: the number of physical segments can be no greater than the number of virtual segments. Likewise, the length of the two `PDatas` can be no greater than the `sources` fields. These constraints are implied by invariants 6 and 7.

```
appendPR :: PData (PA e) -> PData (PA e) -> PData (PA e)
appendPR (PNested vsegd1 pds1) (PNested vsegd2 pds2)
  = PNested (appendVSegd (length pds1) vsegd1 vsegd2)
            (pds1 ++ pds2)

appendVSegd ps1 (VSegd sm1 ssegd1) (VSegd sm2 ssegd2)
  = VSegd (sm1 ++ map (+ lengthSSegd ssegd1) sm2)
  $ appendSSegd ps1 ssegd1 ssegd2

appendSSegd ps1 (SSegd ss1 us1 segd1) (SSegd ss2 us2 segd2)
  = SSegd (ss1 ++ ss2) (us1 ++ map (+ ps1) us2)
  $ appendSegd segd1 segd2

appendSegd (Segd ls1 is1) (Segd ls2 is2)
  = let n1 = sum ls1
    in  Segd  (ls1 ++ ls2) (is1 ++ map (+ n1) is2)
```

Here is an example array that we will use in a moment:

```
                [[K] [] [L M N O]]
------------------------------------------------ (ARR3)
PA 3 (PNested
(VSegd  segmap: [0 1 2]
(SSegd sources: [0 0 0] starts:  [0 1 1])
(Segd  lengths: [1 0 4] indices: [0 1 1]))
(PChars 0: [K L M N O]))
```

Appending the array from Figure 3 with ARR3 above yields:

```
   [[A B] [A B] [A B] [C D E] ...  [K] [] [L M N O]
------------------------------------------------ (ARR4)
PA 10 (PNested
(VSegd  segmap: [0 0 0 1 2 2 3 4 5 6]
(SSegd sources: [0 0 1 1 2 2 2]  starts: [1 3 0 4 0 1 1]
(Segd  lengths: [2 3 2 1 1 0 4] indices: [0 2 5 7 8 9 9])))
(PChars 0: [X A B C D E]
        1: [F G X X H X X X]
        2: [K L M N O]))
```

The data block of ARR3 joins the set of data blocks in the result without any copying.

4.6 Culling Physical Segments

As mentioned in §4.5, we need invariants 6 and 7 to ensure that appendPA has the correct asymptotic complexity. Suppose we wish to append ARR1 from §4.3 to ARR3 above. Invariant 7 is already satisfied, so this part is fine. However, as we produced ARR1 by using a replicates operation with zero valued replication counts, physical segment 2 is no longer reachable from the segmap, which violates invariant 6. To recover this we use the following operations:

```
cullOnSegmap::Vector PsId-> SSegd-> (Vector PsId, SSegd)
cullOnSSegd ::   SSegd   -> PDatas e-> (SSegd, PDatas e)
```

The cullOnSegmap function takes the segmap and SSegd for an array. It filters out the physical segments from the SSegd that are unreachable from the segmap, returning an updated segmap and SSegd. In the result, the number of physical segments is necessarily bounded by the length of segmap. Likewise, cullOnSSegd filters out data blocks in the PDatas not reachable from the sources field of the SSegd. We need this second operation because performing just the first could leave some data blocks unreachable from the sources field, thus violating invariant 7. Culling ARR1 yields:

```
              [[A B] [C D E] [H]]
 ----------------------------------------------- (ARR5)
  PA 3 (PNested
   (VSegd  segmap: [0 1 2]
   (SSegd sources: [0 0 1]  starts: [1 3 4])
   (Segd  lengths: [2 3 1] indices: [0 2 5]))
   (PChars 0: [X A B C D E]
           1: [F G X X H X X X])
```

All array operators that filter out entries from the segmap need to apply cullOnSegmap and cullOnSSegd to preserve the invariants. For example, the invariant preserving version of replicatesPA is as follows:

```
replicatesPA:: Vector Int -> PData (PA e) -> PData (PA e)
replicatesPA lens (PNested (VSegd segmap ssegd) pdatas)
 = PNested (VSegd segmap' ssegd'') pdatas'
 where (segmap', ssegd')
        = cullOnSegmap (replicatesV lens segmap) ssegd
       (ssegd'',  pdatas')
        = cullOnSSegd   ssegd' pdatas
```

We will now sketch how cullOnSegmap is implemented, leaving the full details to the companion technical report [14]. The operation of cullOnSSegd is similar. We start by producing a vector of flags that record which of the physical segments are reachable from the segmap. For ARR1 this is [T T F T]. The flags are calculated by first filling the target vector with the default value F and then using *concurrent writes* to set elements referenced by the segmap to T. Then, we use the flags vector to compute the physical segment identifiers that appear in the result: [0 1 3]. We expand this vector to one that maps between the physical segment identifiers in the result to the identifiers in the source: [0 1 X 2]. The X indicates an unused element, which the implementation can fill with any value. Finally, we use this mapping to permute the segmap, sources, starts and lengths fields of the source array, and then recompute the indices.

The work and space complexity of cullOnSegmap is linear in the length of the segmap being processed and the number of physical segments referenced. Likewise, the complexity of cullOnSSegd is linear in the length of the SSegd and the number of data blocks. This ensures that we do not break the complexity budget of operations such as replicates that make use of these functions.

5. Projection, Concatenation and Reduction

The replicates and append operators described in the previous sections highlight the fundamental features of our array representation. The work efficient implementation of replicates requires that we represent shared segments without copying element data. As replicates may also drop segments, we must handle scattered segments as well. In addition, the work efficient implementation of append requires multiple data blocks with scattered segments as well as the use of the culling operations from §4.6. Culling ensures that the size of the physical representation of an array is bounded by the size of its logical value. We now move on to describe the other operators that we need to support with the new representation when vectorising programs. Happily, we can support them with the correct work complexity without any further extensions.

5.1 Index and Extract

Indexing into a nested array is straightforward. We use the segmap to determine the target segment and then extract (slice) it from its data block. We present this operation for expository purposes only: indexing operators in the source program will be vectorised to lifted indexing, which we discuss in a moment.

```
 indexPA (PNested (VSegd segmap
                  (SSegd sources starts
                  (Segd  lengths _))) pdatas) ix
  = PA len (extractPR pdata start len)
  where psegid  = segmap  ! ix
        source  = sources ! psegid
        start   = starts  ! psegid
        len     = lengths ! psegid
        pdata   = indexdPR pdatas source
```

For indexing to be constant time, `extractPR` must be as well. When the returned value is a `Vector Int`, or some other vector of scalars, the `vector` package provides constant time extract by storing a starting index as well as the slice length in the returned `Vector`. To extract a range of subarrays from a nested array we extract their physical segment identifiers from the `segmap` and then cull the other fields to enforce invariants 6 and 7 from §4. For example, extracting the middle two segments from `ARR4` yields:

```
                     [[F G] [F G]]
------------------------------------------------- (ARR6)
  PA 2 (PNested
   (VSegd  segmap: [0 0]
   (SSegd sources: [0]    starts: [0]
   (Segd  lengths: [2]   indices: [0])))
   (PInts 0: [F G X X H X X X]))
```

Unfortunately, when `indexPA` returns a nested array, the called `extractPR` instance must cull unused physical segments; hence, the overall indexing operation is not constant time. For this reason, our array representation cannot perform all operations that the direct pointer based one could within the same complexity. However, this does *not* worsen the complexity of vectorised programs, because index and extract operations in the source program are converted to *lifted* versions by the vectoriser. We can perform these lifted operations within the required complexity bounds.

Lifted indexing itself is a simple wrapper for the `indexvsPR` function, whose signature is shown in Figure 5.

```
indexlPR :: Int -> PData (PA e) -> PData Int -> PData e
indexlPR c (PNested vsegd pdatas) (PInt is)
  = indexvsPR pdatas vsegd $ zipV (enumFromN 0 c) is
```

The `indexvsPR` function takes a set of data blocks, a virtual segment descriptor, and an array of pairs of virtual segment identifiers and element indices within those segments. As we wish to lookup one element from each segment, we enumerate all the available segment identifiers with `enumFromN`. The `indexvsPR` function itself implements *virtual shared indexing*, it retrieves several elements from some shared data blocks (`pdatas`). It uses the index space transform expressed by the `vsegd` to map the logical view of the array referred to by the segment identifiers and element indices, to the physical view of the array in terms of the `pdatas`. The definition of `indexvsPR` is similar to `indexPR`, though we leave the full details for the technical report [14].

5.2 Concatenation

A central feature of Blelloch's approach to flattening nested parallelism is the use of the `concat` and `unconcat` operators to avoid the need for multiply lifted versions of vectorised functions. These operators are used when vectorising higher order functions such as `mapP` and `zipWithP`. An example of their use is in the vectorised function `retrieve_v` from §2. See the derivation of this function in [14] for how they are introduced.

With the baseline array representation from Figure 2, both `concat` and `unconcat` are constant time operations. To concatenate an array we simply remove the segment descriptor, and to unconcatenate we reattach it. This is possible with the baseline representation, because the form of the segment descriptor implies that the physical segments lie contiguously in a single, flat data block. The description of the segments consists fundamentally of the `lengths` field, with the `indices` being computed directly from it. There is no scattering information such as the `starts` and `sources` fields of our `SSegd`.

As we have seen, the limitation of the baseline representation is that it cannot represent index space transformations on nested arrays except by copying element data. In our new representation, we encode such index space transforms in the segment descriptor, which avoids this copying. The price we pay is that the physical segments in a nested array are no longer guaranteed to be contiguous, so we cannot simply discard the segment descriptor to concatenate them. Instead, the `concat` function must now copy the segment data through the index space transform defined by the segment descriptor, to produce a fresh contiguous array. This is essentially a *gather* operation. The main job is done by `extractvsPR` from Figure 5, with `concat` itself being a wrapper for it:

```
concat :: PA (PA e) -> PA e
concat (PA _ (PNested vsegd pdatas))
 = let  pdata   = extractvsPR pdatas vsegd
   in   PA (lengthPR pdata) pdata
```

The `extractvsPR` function takes some data blocks, a segment descriptor that describes the logical array formed from those blocks, and copies out the segment data into a fresh contiguous array. Importantly, although both `extractvsPR` and `concat` are now *linear* in the length of the result, this does not worsen the complexity of the vectorised program compared with the baseline representation. The reason is that `concat/unconcat` trick is only needed when vectorising higher order functions such as `mapP`. In terms of the unvectorised source program, `mapP` is at least linear in the length of its argument array, because it produces a result of the same length. The vectorised version of `mapP` is implemented by concatenating the argument array, applying the (lifted) worker function, and then unconcatenating the result. The `concat` and `unconcat` functions can then be linear in the length of this result, because the unvectorised version of `mapP` has this complexity anyway.

Note that the linear complexity of `concat` is independent of the depth of nesting of the source array. To concatenate an array of type `(PA (PA (PA (PA Int))))` we only need to merge the two outer-most segment descriptors. The third level segment descriptors, and underlying `Int` data blocks are not touched. There is an example of this in the accompanying technical report [14].

5.3 Demotion, Promotion and Unconcatenation

The `unconcat` function is defined in terms of generally useful demotion and promotion operators that convert between the different segment descriptor types. We will discuss these operators first before continuing onto `unconcat`. The operators are as follows:

```
demoteVSegd  :: VSegd -> SSegd
demoteSSegd  :: SSegd -> Segd
promoteSegd  :: Segd  -> SSegd
promoteSSegd :: SSegd -> VSegd
```

Abstractly, demoting a `VSegd` to a `SSegd` or a `SSegd` to a `Segd` discards information about the extended structure of the array, such as how segments are shared or scattered through the store. Going the other way, promoting a `Segd` to a `SSegd` or a `SSegd` to a `VSegd` adds redundant information. In our concrete implementation, many array functions (including `unconcat`) are defined in terms of these operators. The fact that these functions are defined this way is also used when optimising for absolute performance, which we will discuss in §6.

5.3.1 Demotion

Demoting a segment descriptor eliminates fields from its representation. Consider the following example:

```
 virtual segs: [ [B C D] [G] [] [B C D] [E F] [A] ]
physical segs: [ [A] [G] [B C D] [E F] [] ]
------------------------------------------------- (ARR7)
PA 6 (PNested
 (VSegd  segmap: [2 1 4 2 3 0]
 (SSegd sources: [1 0 1 0 0] starts:  [0 2 1 0 0]
 (Segd  lengths: [1 1 3 2 0] indices: [0 1 2 4 6])))
 (PChars 0: [E F G] 1: [A B C D]))
```

Here we have shown the virtual segments as described by the VSegd, as well as the physical segments described by the SSegd. Note that the virtual segments need not appear in the same order as the physical segments are defined, which allows us to implement permutation operations on nested arrays by permuting the segmap. Demoting the VSegd to a SSegd pushes the information about sharing encoded by the segmap into the other fields of the segment descriptor. It also forces the entries in the SSegd to appear in the same order as the logical array they define:

```
         [ [B C D] [G] [] [B C D] [E F] [A] ]
-------------------------------------------------------
 (SSegd sources: [1 0 0 1 0 1]  starts: [1 2 0 1 0 0]
 (Segd  lengths: [3 1 0 3 2 1] indices: [0 3 4 4 7 9]))
 (PChars 0: [E F G] 1: [A B C D])
```

To demote the array we have computed new starts, sources and lengths fields by permuting the originals using the segmap. In practice, when we demote a VSegd, we must be mindful of the potential for *index space overflow*. By this we mean that if a nested array consists of many virtual copies of a large sub-array, then the total number of elements in the virtual array may be larger than the address space of the machine, even though all the *physical* data fits within it. In this case the elements of the indices field may no longer fit in a machine word. We will return to this point in §5.4.1. Avoiding index space overflow is the main reason we use an explicit segmap, instead of representing all arrays with the above demoted form (without a segmap).

Continuing on, we demote a SSegd to a Segd by simply discarding the outer SSegd wrapper, along with the sources and starts fields. To represent the same logical array, we must then gather the segment data into a fresh data block, similarly to the extractvsPR function described in §5.1. For our example this produces:

```
         [ [B C D] [G] [] [B C D] [E F] [A] ]
--------------------------------------------------------
 (Segd lengths: [3 1 0 3 2 1] indices: [0 3 4 4 7 9])
 (PChars 0: [B C D G B C D E F A])
```

As with the previous demotion, our nested array still has the same logical value as the original. However, by giving up the sources and starts fields we have lost information about how the segments were originally scattered through the store. This forces us to copy them into a fresh data block to represent the original logical array, leaving us with the old array representation from Figure 2.

5.3.2 Promotion

Promoting an array fills in missing segment descriptor fields with redundant information. To promote a Segd to a SSegd, we reuse the existing indices field for starts and fill the sources with all zeros. This indicates that all physical segments lie contiguously in a single flat array. To promote the SSegd to a VSegd we then enumerate the physical segments in the segmap. Performing both promotions to the demoted array from the previous section yields the following:

```
          [ [B C D] [G] [] [B C D] [E F] [A] ]
------------------------------------------------ (ARR8)
PA 5 (PNested
 (VSegd  segmap: [0 1 2 3 4 5]
 (SSegd sources: [0 0 0 0 0 0] starts:  [0 3 4 4 7 9]
 (Segd  lengths: [3 1 0 3 2 1] indices: [0 3 4 4 7 9])))
 (PChars 0: [B C D G B C D E F A]))
```

Note that promoting a segment descriptor does not change the logical structure of the array, it just fills in redundant fields in the representation. In our concrete implementation the initialisation of the segmap and sources fields with these "boring" values can often be avoided (§6).

5.3.3 Unconcatenation

To unconcatenate an array, we demote the source segment descriptor down to a plain Segd and then re-promote it back to a VSegd, before attaching it to the second array:

```
unconcatPR :: PA (PA a) -> PA b -> PA (PA b)
unconcatPR (PA n (PNested vsegd _)) (PA _ pdata)
 = let  segd    = demoteSSegd  $ demoteVSegd vsegd
        vsegd'  = promoteSSegd $ promoteSegd segd
   in   PA n (PNested vsegd' (singletondPR pdata))
```

We need the demotion-promotion process because the sharing and scattering information in the VSegd is only relevant to the first array, not the second array (of type (PA b)) that we attach it to.

Finally, we can normalise the physical structure of an array by concatenating it down to atomic elements and then unconcatenating to re-apply the nesting structure. This eliminates all unused array elements from the data blocks, which improves locality of reference for subsequent operations, and is useful when writing arrays to the file system. Here is the version for triply nested arrays:

```
normalise3 :: PA (PA (PA e)) -> PA (PA (PA e))
normalise3 arr2
 = let  arr1    = concat arr2
        arr0    = concat arr1
   in   unconcat arr2 (unconcat arr1 arr0)
```

Creating versions of normalise for other degrees of nesting is straightforward. Normalising the doubly nested ARR7 from §5.3.1 yields exactly ARR8 from §5.3.2. Note that if we were to elide the VSegd and SSegd layers, a normalised arrays have the same form as the baseline representation from §3.

5.4 Reduction and Dynamic Hoisting

Consider the following function retsum, which indexes several shared arrays, and adds the retrieved value to the sum of the array it came from. This has a similar structure to retrieve from §2.

```
retsum :: [:[:Int:]:] -> [:[:Int:]:] -> [:[:Int:]:]
retsum xss iss
 = zipWithP mapP
          (mapP (\xs i. indexP xs i + sumP xs) xss) iss
```

Here is retsum applied to some example arrays:

```
retsum  [[1 2]   [4 5 6] [8]]       (xss)
        [[1 0 1] [1 2]   [0]]       (iss)
   ==>  [[5 4 5] [20 21] [16]]
```

The subexpression sum xs duplicates work for every application of the inner function abstraction, because it sums the entire xs array once for each of the integer elements in the result. The result of vectorisation, inlining and simplifying retsum is shown below — the accompanying technical report [14] includes the full derivation.

```
retsum_v :: PA (PA Int) -> PA (PA Int) -> PA (PA Int)
retsum_v xss iss
 = let ns      = lengths iss
       n       = sum ns
       yss'    = replicates ns xss
   in  unconcat iss
        $ add_l n (index_l n yss' (concat iss))
                  (sum_l n yss')
```

In retsum_v, the fact that the original sum expression duplicates work is revealed in the fact that the lifted version (sum_l) is being applied to a replicated array. At runtime, the segments of the first array are replicated according to the lengths of the segments in the second. The intermediate result (replicates ns xss) is on the next page.

```
      [[1 2] [1 2] [1 2] [4 5 6] [4 5 6] [8]]
 --------------------------------------------- (ARR9)
   PA 6 (PNested
    (VSegd  segmap: [0 0 0 1 1 2]
    (SSegd sources: [0 0 0]    starts: [0 2 5]
    (Segd  lengths: [2 3 1]   indices: [0 2 5])))
    (PInts 0: [1 2 4 5 6 8])
```

Our segmap directly encodes which of the physical segments are being shared. Instead of repeatedly summing segments we know to be identical, we can instead sum the physical segments defined by the SSegd, and replicate the results according to the segmap. By doing this we actually *improve* the asymptotic complexity of the original program, by avoiding repeated computation that it would otherwise perform. Note that this process depends on Invariant 6 from §4, as we do not wish to sum unreachable physical segments that are not part of the logical array.

Avoiding repeated computation in this way achieves the same result as the *hoisting* or *full laziness* program transformation, but in a dynamic way. In contrast, performing this transform statically at compile-time would yield the following:

```
retsum xss iss
  = zipWithP mapP
           (mapP (\xs. let x = sumP xs
                       in \i. indexP xs i + x) xss) iss
```

However, the GHC simplifier will *not* in-fact perform the above transform, as it does not generally improve performance [11]. Finally, although dynamic hoisting may seem like an opportunistic improvement, perhaps not worth the trouble, failing to perform it has other ramifications, which we discuss in the next section.

5.4.1 Fused Hylomorphisms and Index Space Overflow

A subtle point about the retsum example is that if an implementation does not perform dynamic hoisting, it could risk overflowing machine words. This is a general problem with *fused hylomorphisms*, with a hylomorphism being a computation that first builds a structure (like with replicates) before reducing it (like with sum_l). Although it may be possible to fuse these two operations together so the intermediate structure is never actually created, it is problematic when the index space of that structure is larger than the address space of the machine.

For example, suppose the xss array from the previous section contains 10 elements and iss contains 500 million. Although this amount of data is easily stored on current hardware, the total number of virtual elements produced by replicates would be 5 x 10^9. This number is not representable in a 32-bit word. This problem is acute because the function that defines the intermediate structure (replicates) is introduced by vectorisation and does not appear in the source program. Simply telling the user "you can't do that" would be unreasonable.

Managing this problem is the main reason that we include an explicit segmap in our array representation. Without the segmap, we would instead record each virtual segment separately, like with the first demoted array of §5.3.1. However, in cases of index-space overflow, elements of the indices field would become too large to be stored. The indices field itself is needed when partitioning the work in the implementation of sum_l. We also need the total size of the array, which would again be too large.

Requiring 64-bit array indices and eliminating the segmap is an alternate solution, but it is not clear whether this would be better overall. On 32-bit machines, memory traffic to the indices field would double because of the larger word size. On all machines, operations such as replicates would need to process both the sources and starts field instead of touching the singular segmap. On the other hand, indexing operations would not need to dereference the segmap, or maintain invariant 6, but as as we will see in §6 this can often be avoided anyway. The code to maintain this invariant is localised, and very similar to that needed for invariant 7, so it would not be a significant reduction in implementation complexity. For now we choose to keep the segmap and leave the quantitative comparison to future work.

5.5 Flattening and Space Usage

In contrast to the problem with replication outlined in §2, flattening nested parallelism can increase the asymptotic space complexity in a way that this paper does not address [15, 19]. For example, suppose we vectorise the following function that takes an array of n points and computes the maximum distance between any pair. The full derivation is in the companion technical report [14].

```
furthest :: PA (Float, Float) -> Float
furthest ps = maxP (mapP (\p. maxP (mapP (dist p) ps)) ps)
```

The flattened version is a hylomorphism that first computes $O(n^2)$ distances before reducing them to determine the maximum. Whereas the sequential version would run in $O(n)$ space, the flattened version needs $O(n^2)$ space to hold the intermediate vector of distances. Note that vectorisation does not increase the asymptotic *work* complexity, because these distances must be computed anyway.

5.6 Pack and Combine

The pack and combine functions from Figure 5 are used in the parallel implementation of if-then-else. The pack function takes an array of elements, an array of flags of the same length, and returns on only those elements that have their flag set. This function is used to split the parallel context of if-then-else into the elements associated with each branch. It can be implemented in terms of replicates, using a replication count of 1 for True flags and 0 for False flags. We mention it separately because pack is the common name for this operation in the literature. The combine function takes an array of flags, two arrays of elements, and intersperses the elements according to the flags. For example combine [T F F T] [1 2] [3 4] = [1 3 4 2]. This function is used to merge the results of each branch once they have been computed. On a high level, the implementation of combine is similar to append because the result contains elements from both source arrays, though we leave the implementation to [14].

To achieve our Complexity Goal, both pack and combine must be linear in the length of the flags array. This is because entering a branch in the source program is a constant time operation. In vectorised code, many branches are entered in one parallel step, so the functions that implement this operation must be linear in the number of elements being processed. Achieving this goal with the baseline array representation is not possible, because packing and combining nested arrays requires that we copy element data. In contrast, with our new representation we can simply pack and combine the segment descriptors, leaving the underlying element data untouched. In [1], Blelloch suggested that it would be more efficient to work on *sparse segments*, which we are now able to do.

As mentioned in §1 vectorisation can only preserve the complexity of the source program up to *containment*. This problem stems from the fact that flattening if-then-else causes the computations that take each branch to be executed one after another, instead of concurrently. In [18] Riely and Prins give an example recursive function that calls itself in both branches of an if-then-else, and where vectorisation worsens its asymptotic complexity independent of considerations of the array representation. Luckily, the containment problem is rarely met in practice. Riely and Prins prove that provided one branch in each if-then-else executes with a constant number of parallel steps, the containment problem is avoided. This constraint is met by the base case of most recursive functions. However, their language is first order, so their proof does not automatically apply to ours.

6. Pragmatics

Although our new array representation allows the index space transforms introduced by the vectoriser to have the correct asymptotic complexity, there are several cases where a direct implementation would perform poorly in absolute terms. For example, implementing `promoteSSegd` from §5.3.2 by physically filling the `segmap` with `[0 1 2 ...]` leaves something to be desired. However, in many cases the construction of these fields can be sidestepped using rewrite rules. For example, in our concrete implementation we have the following functions:

```
sum_vs :: VSegd -> PDatas Int -> PData Int
sum_s  :: Segd  -> PData  Int -> PData Int
```

Both of these can be used to implement lifted sum (`sum_l`) by using a simple wrapper. The difference is that while `sum_vs` accepts a full `VSegd` to describe the segmentation of the array, `sum_s` only accepts a plain `Segd`. The implementation of the latter is simpler, as it does not need to worry about the `segmap`, `starts` and `sources` fields that define the sharing and scattering of segments. During code transformation, we apply the following rewrite rule:

```
RULE "sum_vs/promote"  forall segd arr.
      sum_vs (promoteSSegd (promoteSegd  segd))
             (singletondPR arr)       = sum_s segd arr
```

The rule says that to sum the segments of a nested array defined by a promoted `Segd`, we can just use the `Segd` directly. Rules like this one are frequently applicable because the definition of key operations such as `unconcat` and `extractvsPR` explicitly use `promoteSegd` and so on to construct their results. Note that the ability to apply rules such as this depends critically on the split between the `VSegd`, `SSegd` and `Segd` types, and the fact that `indices` field is in `Segd` rather than `SSegd`. Abstractly, the fact that an array is representable with just a `Segd` tells us that all the segments lie contiguously in the store. The fact that it is representable with just a `SSegd` tells us that the number of physical segments matches the number of logical segments, though several entries in the `SSegd` may still point to the same element data.

Another technique we use is to store a lazy, pre-concatenated version of the array in the array structure itself. In our concrete implementation, the `PNested` structure contains two extra fields holding a plain `Segd` and the `PData` corresponding to the concatenated version of the overall array. Every function that constructs an array is responsible for initialising these fields, either with pre-existing concatenated data, such as produced by `extractvsPR`, or a suspended computation that will concatenate when demanded. When a consumer, such as `mapP`, requires a concatenated version of the array, it can use these fields instead of explicitly concatenating the data itself. With this method array consumers avoid repeatedly concatenating (and thus copying) arrays that the producers know are already concatenated.

We use a similar method to avoid some applications of the `cullOnSegmap` function discussed in §4.6. The key point here is that while reduction operations like `sum_l` need invariant 6 to avoid duplicating work, indexing operations are oblivious to unreachable physical segments. In our implementation, we suspend calls to `cullOnSegmap` with lazy evaluation, and also keep an *unculled* version of the `SSegd` in the array representation. If, say, a nested array is packed and then immediately indexed, then the indexing operation uses the unculled `SSegd`, avoiding the need to call `cullOnSegmap` at all.

6.1 Rewrite Rules and Replication

A reader may be wondering why we cannot also use a rewrite rule to eliminate calls to replication operators in the vectorised program, instead of introducing a new array representation. Suppose we vectorise the following function that gathers multiple character values from a shared array called `table`.

```
gather :: [:Char:] -> [:Int:] -> [:Char:]
gather table indices
 = mapP (\ix -> table !: ix) indices
```

The vectorised version is as follows:

```
gather_v :: PA Char -> PA Int -> PA Char
gather_v table indices
 = index_l len (replicate len table) indices
 where len = length indices
```

As per §2, `index_l` is lifted indexing. Note that with the old array representation, this function would have the wrong asymptotic complexity due to the use of `replicate`. However, suppose we had a second version of indexing (`index_s`) that could retrieve elements from a single shared array. This operation is also known as *backwards permutation*.

```
index_l :: Int -> PA (PA a) -> PA Int -> PA a
index_s :: Int ->     PA a  -> PA Int -> PA a
```

Given `index_s`, a seemingly obvious way to optimise `gather_v` is to apply the following rewrite rule:

```
RULE "index_l/index_s" forall c xs ys.
      index_l c (replicate c xs) ys = index_s c xs ys
```

The problem is that this rule improves the asymptotic complexity of the program, which turns out to be a bad thing engineering wise. The left of the rule uses work and space $O(\textit{length}\ \texttt{xs}\ .\ \textit{length}\ \texttt{ys})$ while the right uses $O(\textit{length}\ \texttt{ys})$. These are different because indexing is a *projection*, which does not inspect all of its input data.

The trouble is that for the rule to fire, the producer (`replicate`) and consumer (`index_l`) of the replicated array must come together during code transformation. If the program is written in a way that prevents this from happening, then `replicate` will not be eliminated. For example, suppose we parameterised `gather` over a function to apply to each index value:

```
gatherFun ::([:Char:] -> Int -> Char)
          -> [:Char:] -> [:Int:] -> [:Char:]
gatherFun fun table indices = mapP (fun table) indices
```

Vectorising this function yields the following:

```
gatherFun_v :: PA (PA Char :-> Int :-> Char)
            -> PA Char -> PA Int -> PA Char
gatherFun_v (AClo fv fl envs) table indices
 = fl c envs (replicate c table) $:^ indices
 where c = length indices
```

When we vectorize higher order functions, the parameter function is represented as an *array closure*. The array closure constructor `AClo` bundles up the closure-converted version of the function (`fv`), the lifted version (`fl`) and the environment that was captured in its closure (`envs`). The lifted application operator (`$:^`) then applies the closure to its final argument `indices`. See [12, 13] for a more detailed explanation. The main point here is that the parameter function `fl` is unknown to the vectoriser. With just the code above, it is impossible to eliminate the `replicate` operation, because we do not know what `fl` will turn out to be.

A tempting hack-around is to force every function in the program to be inlined, and hope that follow on code transformation discovers the true identity of `fl`. We used this approach in our previous implementation of DPH, and it turns out to be a slippery slope to suffering. Small changes in the structure of the source program, or behaviour of the various transforms, can result in a previously well performing program becoming unrunable due to the changed asymptotic complexity. This approach also does not "fix" recursive programs where the producer and consumer are the same function, as the the same function cannot be inlined into itself indefinitely. An example of such a program is given in §7.2. Our solution is to instead provide a new array representation, that guarantees that even with all follow-on optimisations disabled, the program will still run with the correct asymptotic complexity.

7. Benchmarks

In this section we present several programs were the baseline array representation worsened the asymptotic complexity of the vectorised program. All benchmarks were taken on an Intel i7 Quad-Core / 8GB desktop machine. We have opted to present data for the program running in single threaded mode only. We have not finished adapting our parallel stream fusion framework to the new array representation, so parallel speedup is currently dominated by the creation of intermediate arrays such as the "boring" fields described in §5.3.2 and §6. However, all of the underling primitives we use operate on bulk arrays and are amenable to parallelisation.

7.1 Sparse Matrix-Vector Multiplication

This program multiples a sparse matrix by a dense vector and is discussed in [12]. The matrix is represented as an array of rows, where each row is an pair of a column index and the `Double` value in that column.

```
smvm :: [:[:(Int, Double):]:] -> [:Double:] -> [:Double:]
smvm matrix vector
 = let term (ix, coeff) = coeff * (vector ! ix)
       sumRow row       = sumP (mapP term row)
   in  mapP sumRow matrix
```

As `vector` is free in the definition of `term`, with the old array representation it would be copied once for every non-zero element of the matrix. With the new array representation the vector is not copied and it runs with the same asymptotic complexity as an unvectorised reference implementation written with the `Data.Vector` package. The `Data.Vector` version is currently faster than with our new representation because stream fusion [7] does a better job at eliminating intermediate values.

7.2 Tree Lookup

The following microbenchmark exposes the replicate problem in sharp relief. It performs a divide-and-conquer of an `indices` array, while referring to a top level `table`. In the base case, a single index is used to access the top-level table, and the table is rebuilt during the return calls.

```
 treeLookup :: [:Int:] -> [:Int:] -> [:Int:]
 treeLookup table indices
   | lengthP indices == 1 = [:table !: (indices !: 0):]
   | otherwise
   = let half = lengthP indices `div` 2
         s1   = sliceP 0    half indices
         s2   = sliceP half half indices
     in concatP (mapP (treeLookup table) [: s1, s2 :])
```

As `table` is partially applied to `treeLookup`, with the baseline representation the entire `table` is copied once for every element of `indices`. In the vectorised version the call to `replicate` cannot be eliminated with the rewrite rules discussed in §6.1, because the producer and consumer of the replicated array are in different recursive calls. With our new array representation, the sharing is managed by our `segmap` and the elements of `table` are not copied.

7.3 Barnes-Hut

The Barnes-Hut algorithm performs a two dimensional gravitation simulation of many massive bodies. At each time step, the algorithm builds a quad-tree to partition the space the bodies lie in, and computes the centroid of all bodies in each branch. The tree is then used to compute the force between each body and all the others, approximating the force between distant bodies by using the centroids. Whereas a naive algorithm would use work $O(n^2)$ in the number of bodies, the Barnes-Hut approximation is $O(n.log\ n)$.

With the baseline array representation, the copying replication problem appears at the very top level. Once we have built the quad-tree, we use it to compute the force on each body.

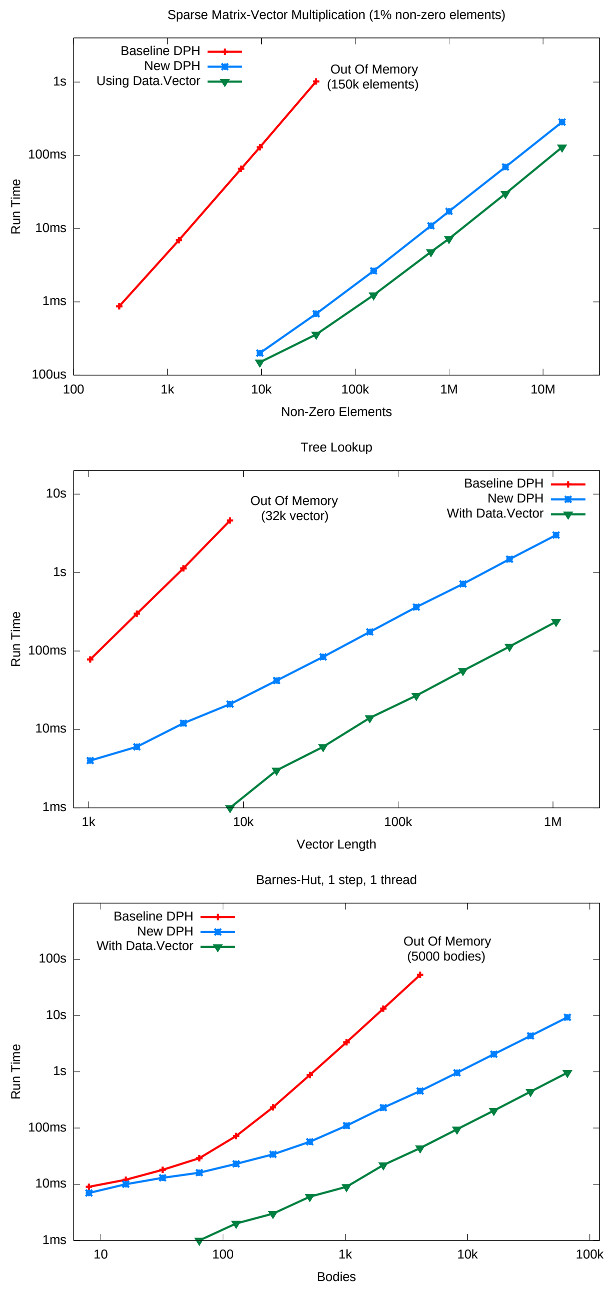

Figure 6. Benchmark Runtime Performance

This is done by the following function:

```
calcAccels :: Double -> Box -> [:MassPoint:] -> [:Accel:]
calcAccels epsilon boundingBox points
 = mapP (\m -> calcAccel epsilon m tree) points
 where tree = buildTree boundingBox points
```

As `tree` is free in the closure passed to `mapP`, it is copied once for every body. As before, with the new array representation the tree is not copied and the program runs with the same asymptotic complexity as an unvectorised reference implementation written with the `Data.Vector` package.

8. Related Work

Approaches to the implementation of irregular parallelism roughly fall into two categories: thread-based implementations, like Manticore [8] or Ct [9], to name a few, and those based on flattening. Both have their advantages and drawbacks. With the former approach, scheduling, synchronisation, and granularity control are a concern, as well as a more restricted set of target architectures, though they do not suffer from the complexity problem described in this paper. However, the problem we describe is a fundamental issue for all flattening based approaches, and has been identified and discussed in several publications.

The original first-order flattening transform and array representation shown in Figure 2 was introduced by Blelloch and Sabot in in [1]. In this work the `replicate` function is called `distribute` when applied to scalars and `distribute-segment` when applied to arrays. As a possible extension to the handling of conditionals they suggest operating on sparse segments instead of first eliminating gaps between them with the `pack` operation. This idea is not elaborated further. The single example program they present (Quicksort) only uses `distribute` and `pack` on arrays of scalars, and thus does not suffer problems with asymptotic complexity.

In [2] Blelloch proves that a subset of programs written with the scan-vector instruction set can be vectorised while preserving their asymptotic work and step complexity. Such programs must be both *contained*, and not use indirect memory access, which is equivalent to disallowing functions to have free variables. Appendix C of the NESL manual [3] gives the work complexity of vectorised programs, and states that the contents of free variables is copied across each iteration of the `apply-to-each` (`map`) construct. Finally, in [4] he presents a provably time and space efficient version of NESL, but the operational semantics is based around fine grained threads instead of SIMD style vectors.

In [16], Palmer et al. address the issue by disallowing partial applications and removing some of the problematic cases using rewrite rules. For Haskell, ruling out partial applications to appear anywhere in a parallel context would be neither desirable nor statically enforceable. The rewrite rules used do not fire if the offending index space transform is applied indirectly as part of another function. This is a general drawback of using rewrite rules, which can be acceptable if the rewriting only leads to a constant improvement, but not in our case, where the failure to identify problematic expressions results in asymptotically worse performance.

In [18], Riely and Prins solve this problem by using vectors of references, but at the time the article was written, there was no implementation, so they could not provide any experimental data as to the absolute performance. To the best of our knowledge, they have not published any further results on this approach. The suggested representation is similar to one of the states our representation can take on, for example, as a result of creating a nested vector by collecting a number of different flat arrays in a nested one. However, the use of purely pointer based representations can lead to poor locality and complicates distribution and load balancing. It also increases garbage collection overhead, as every subarray must be traversed individually. In contrast, our approach aims at keeping the data representation as flat as possible, and only resorts to the partially flattened representation whenever the completely flat representation would lead to worse work complexity.

Acknowledgements This work was supported in part by the Australian Research Council under grant number LP0989507.

References

[1] G. Blelloch and G. W. Sabot. Compiling collection-oriented languages onto massively parallel computers. *Journal of Parallel and Distributed Computing*, 8:119–134, 1990.

[2] G. E. Blelloch. *Vector models for data-parallel computing*. MIT Press, 1990.

[3] G. E. Blelloch. NESL: A nested data-parallel language (version 3.1). Technical report, Carnegie Mellon University, 1995.

[4] G. E. Blelloch and J. Greiner. A provable time and space efficient implementation of NESL. In *ACM SIGPLAN International Conference on Functional Programming*, pages 213–225, 1996.

[5] M. M. T. Chakravarty, G. Keller, S. Peyton Jones, and S. Marlow. Associated types with class. In *POPL '05: Proceedings of the 32nd ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages*, pages 1–13. ACM Press, 2005.

[6] M. M. T. Chakravarty, R. Leshchinskiy, S. Peyton Jones, G. Keller, and S. Marlow. Data Parallel Haskell: a status report. In *DAMP 2007: Workshop on Decl. Aspects of Multicore Progr.* ACM Press, 2007.

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

[8] M. Fluet, M. Rainey, and J. Reppy. A scheduling framework for general-purpose parallel languages. In *Proceedings of the 13th ACM SIGPLAN international conference on Functional programming*, ICFP '08, pages 241–252. ACM, 2008.

[9] A. Ghuloum, T. Smith, G. Wu, X. Zhou, J. Fang, P. Guo, B. So, M. Rajagopalan, Y. Chen, and B. Chen. Future-proof data parallel algorithms and software on Intel multi-core architecture. *Intel Technology Journal*, November 2007.

[10] J. Hill, K. M. Clarke, and R. Bornat. Vectorising a non-strict data-parallel functional language, 1994.

[11] S. P. Jones, W. Partain, and A. Santos. Let-floating: Moving bindings to give faster programs. In *Proc. of the 1996 ACM SIGPLAN International Conference on Functional Programming*, pages 1–12, 1996.

[12] R. Leshchincskiy. *Higher-Order Nested Data Parallelism*. PhD thesis, Technische Universität Berlin, 2006.

[13] R. Leshchinskiy, M. M. T. Chakravarty, and G. Keller. Higher order flattening. In *Computational Science - ICCS 2006, 6th Int. Conf., Proceedings, Part II*, volume 3992, pages 920–928. Springer, 2006.

[14] B. Lippmeier, M. M. T. Chakravarty, G. Keller, R. Leshchinskiy, and S. P. Jones. Work efficient higher-order vectorisation (unabridged). Technical Report UNSW-CSE-TR-201208, University of New South Wales, 2012. URL `TBA`.

[15] D. W. Palmer, J. F. Prins, S. Chatterjee, and R. E. Faith. Piecewise execution of nested data-parallel programs. In *Languages and Compilers for Parallel Computing, volume 1033 of Lecture Notes in Computer Science*, pages 346–361. Springer-Verlag, 1995.

[16] D. W. Palmer, J. F. Prins, and S. Westfold. Work-efficient nested data-parallelism. In *Proc. of the 5th Symposium on the Frontiers of Massively Parallel Processing*, pages 186–193. IEEE, 1995.

[17] S. Peyton Jones, R. Leshchinskiy, G. Keller, and M. M. T. Chakravarty. Harnessing the multicores: Nested data parallelism in Haskell. In *IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS 2008)*, LIPIcs, pages 383–414. Schloss Dagstuhl, 2008.

[18] J. Riely and J. Prins. Flattening is an improvement. In *Proc. of the 7th International Symposium on Static Analysis*, pages 360–376, 2000.

[19] D. Spoonhower, G. E. Blelloch, R. Harper, and P. B. Gibbons. Space profiling for parallel functional programs. In *Proc. of the ACM SIGPLAN Int. Conf. on Functional Programming (ICFP 2008)*, 2008.