Extended Memory Reuse

An Optimisation for Reducing Memory Allocations

Hans-Nikolai Vießmann
Heriot-Watt University
Edinburgh, UK
hv15@hw.ac.uk

Artjoms Šinkarovs
Heriot-Watt University
Edinburgh, UK
a.sinkarovs@hw.ac.uk

Sven-Bodo Scholz
Heriot-Watt University
Edinburgh, UK
S.Scholz@hw.ac.uk

ABSTRACT

In this paper we present an optimisation for reference counting based garbage collection. The optimisation aims at reducing the total number of calls to the heap manager while preserving the key benefits of reference counting, *i.e.* the opportunities for in-place updates as well as memory deallocation without global garbage collection. The key idea is to carefully extend the lifetime of variables so that memory deallocations followed by memory allocations of the same size can be replaced by a direct memory reuse. Such memory reuse turns out particularly useful in the context of innermost loops of compute-intensive applications. It leads to a runtime behaviour that performs pointer swaps between buffers in the same way it would be implemented manually in languages that require explicit memory management, *e.g.* C.

We have implemented the proposed optimisation in the context of the Single-Assignment C compiler tool chain. The paper provides an algorithmic description of our optimisation and an evaluation of its effectiveness over a collection of benchmarks including a subset of the Rodinia benchmarks and the NAS Parallel Benchmarks. We show that for several benchmarks with allocations within loops our optimisation reduces the amount of allocations by a few orders of magnitude. We also observe no negative impact on the overall memory footprint nor on the overall runtime. Instead, for some sequential executions we find mild improvement, and on GPU devices we observe speedups of up to a factor of 4×.

CCS CONCEPTS

• **Software and its engineering → Memory management**; *Compilers*; Functional languages;

KEYWORDS

memory management, reference counting, compiler optimisation

ACM Reference Format:
Hans-Nikolai Vießmann, Artjoms Šinkarovs, and Sven-Bodo Scholz. 2018. Extended Memory Reuse: An Optimisation for Reducing Memory Allocations. In *Proceedings of the 30th Symposium on Implementation and Application of Functional Languages (IFL 2018), September 5–7, 2018, Lowell, MA, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3310232.3310242

IFL 2018, September 5–7, 2018, Lowell, MA, USA

ACM ISBN 978-1-4503-7143-8/18/09.
https://doi.org/10.1145/3310232.3310242

1 INTRODUCTION

Many modern programming languages use implicit memory management. Languages such as SISAL [4, 5], Python [10], Swift [18, 27], or Single-Assignment C (SaC) [12] use reference counting to implement non-delayed garbage collection. The key idea of reference counting is to associate each data structure with a counter that keeps track of the number of shared references that exist in the system. Once the last reference to a data structure is no longer needed the corresponding memory can be freed. For the price of maintaining such a reference counter for each data structure, this approach not only enables earliest possible asynchronous garbage collection, it also provides an elegant solution to the aggregate update problem [17]: whenever a reference counter is one, the corresponding data structure can be updated in place.

Several optimisations to reference counting have been proposed in the literature. They predominantly focus on avoiding reference counter maintenance overheads [7, 21, 25, 30] and on maximising the potential for in-place updates of aggregate data structures [8, 13]. Some other works aim at improving the actual memory handling process, be it in the context of sequential or parallel executions [19, 31].

This paper proposes a novel optimisation for reference counting based garbage collection named *extended memory reuse* (*EMR*). It extends the lifetime of memory allocations to avoid sequences of deallocations and reallocations in favour of a direct memory reuse. While the overheads of such dual calls to the memory manager on shared-memory systems usually has little impact on the overall runtime performance, in the context of accelerator systems, like GPUs, they can have devastating effects on the overall performance.

Accelerator systems typically have separate memory contexts, one for the host and one on the accelerator, and all memory management, including memory allocations on the accelerator, has to be performed by the host [24]. Any memory deallocation or reallocation that has to happen between two executions on the GPU requires a control transfer from GPU to the host and back. Moreover, the change in allocated memory on the GPU may induce superfluous data-transfers between host and device which are known to be the primary source for poor GPU performance [14].

The main contributions of this paper are:

(1) Design of a code-transformation that implements the inference of extended reuse opportunities. We present it as a series of rewrites for a first-order functional language with array comprehensions which resembles the core of SaC.
(2) A full-fledged implementation of the proposed optimisation for the SaC compiler.
(3) Evaluation of the effects of the proposed transformation at the example of benchmarks from the Livermore Loops [23],

Rodinia [6], and NAS [1] Benchmark Suites. We present the effects of *EMR* on these benchmarks with respect to the number of memory operations performed, memory footprint, and overall runtime for two different architectures: a shared memory system and a GPU accelerated system.

Our results suggest that the object lifetime extension pattern that we make use of can be picked up in a wider contexts like static analysers for languages with manual memory allocations like C. Specifically, this becomes desirable in the context of GPU or FPGA accelerator programming using languages like CUDA or OpenCL.

The rest of the paper is organised as follows: in section 2 we briefly describe the SaC language and explain how reference counting works within the language by giving examples. In section 3 we extend on these examples to demonstrate the problem we tackle and how we intend to solve it. In section 4 we provide an overview of the code-transformations and what they are meant to achieve. This is followed by explanations on how the code-transformations work, with some examples given to demonstrate this. In section 5 we present and evaluate the results from applying the *EMR* optimisation on various benchmarks. In section 6 we give a discussion on certain aspects of the optimisation, such as its effect on heap usage during runtime. In section 7 we present and compare some related work, and finally in section 8 we give our conclusion and ideas for future work.

2 BACKGROUND

We start with a brief overview of the SaC language and its reference counting model. The main goal here is to introduce a large enough subset of the language so that we can conveniently illustrate the underlying problem and our solution. For a full introduction consider [11, 29].

2.1 The Core of SaC

SaC is a first-order functional language with built-in arrays. In SaC, arrays are predominantly constructed by an array comprehension construct called the *with*-loop. In this paper we assume the sole construct for array construction to be a restricted form of the *with*-loop which has the following form[1]:

$$\{\ \mathtt{x}\ \texttt{->}\ e_1\ \mid\ \mathtt{x}\ \texttt{<}\ e_2\ \}$$

where x is a variable and e_1 and e_2 are expressions. `x -> `e_1 can be seen as a mapping from indices x to element values e_1 which define the array elements for all its legal indices. The range of legal indices is delimited by e_2 which evaluates to a vector that defines the shape of the resulting array.

A few instances of such *with*-loops are contained in our running example shown in listing 1. Consider line 4, where we use a *with*-loop to implement element-wise addition of the 5-element 1-d arrays a and b. The index iv ranges over the indices {[0],[1],[2],[3],[4]}, and we compute a 5-element 1-d array c where at every index iv the value is computed by the expression a[iv] + b[iv]. The square brackets here denote element selections from a and b, respectively.

The *with*-loops in lines 8 and 12 compute rotations of the array a using the built-in modulo operator denoted by the %-symbol.

Other components of the language are first-order functions and conditionals. As shown in our running example, we use SaC syntax for function definitions and function applications. Partial applications of functions are not supported. Conditional expressions are restricted to the ternary ?: operator, whose syntax is identical to the corresponding construct in C. In contrast to full-fledged SaC, we do not support special syntax for loops. We resort to recursive functions instead in order to keep our language core concise. The function stencil in lines 19–25 together with its call in line 15 of our running example constitutes such a loop. It represents a do-loop that performs 10 iterations of a two-point stencil with cyclic boundary conditions.

```
1  int[5] fun (int[5] a, int[5] b)
2  {
3    // element-wise add
4    c = { iv -> a[iv] + b[iv] | iv < [5] };
5    print (c);
6
7    // rotate right by one element
8    d = { iv -> a[(iv-1) % 5] | iv < [5] };
9    print (d);
10
11   // rotate left by one element
12   e = { iv -> a[(iv+1) % 5] | iv < [5] };
13
14   // 10 iterations 2 point stencil
15   f = stencil (e, 10);
16   return f;
17 }
18
19 int[5] stencil (int[5] e, int n)
20 {
21   f = { iv -> e[(iv-1) % 5] + e[(iv+1) % 5]
22         | iv < [5] };
23
24   r = n == 0 ? f : stencil (f, n-1);
25   return r;
26 }
```

Listing 1: SaC code example

2.2 Basic Reference Counting in SaC

Arrays defined by *with*-loops initially have a reference count of 1. New references to arrays are generated whenever they are passed as arguments to abstractions. At that point one reference is consumed and as many references are generated as there are references to the formal parameter within the body of the abstraction. Our core language here supports three forms of abstractions: functions, assignments to variables which constitute nested let-constructs, and *with*-loops themselves as they replicate their element-expression for each array element. Built-in operations like +, selection, *etc.*,

[1] In SaC, a similar syntactical form exists as a notational short-cut for *with*-loops named set-expression [26]. The form we use here can be expanded into a proper *with*-loop by applying the following expansion rule:

$[\![\{\ \mathtt{x}\ \texttt{->}\ e_1\ \mid x < e_2\ \}]\!] = \mathtt{with}\ \{(0 * e_2 <= x < e_2)\texttt{:}\ e_1;\ \}\texttt{:}\ \mathtt{genarray}\ (e_2, 0)$

consume their arguments. At runtime, when the reference count of an array is decreased to 0, the object is deallocated.

Consider our running example from listing 1 again, we extend it in listing 2 to show function fun annotated with explicit reference count operations (*e.g.* incrc).

```
1  int WLBc (int[1] iv, int[5] a, int[5] b)
2  {
3    incrc (iv, 1); incrc (a, 0); incrc (b, 0);
4    return a[iv] + b[iv];
5  }
6  int WLBd (int[1] iv, int[5] a) { /* ... */ }
7  int WLBe (int[1] iv, int[5] a) { /* ... */ }
8  int[5] fun (int[5] a, int[5] b)
9  {
10   incrc (a, 2);          // 3 references
11   incrc (b, 0);          // 1 reference
12   // ——— WL c ———
13      incrc (a, 4);       //  a s in c with-loop body
14      incrc (b, 4);       //  b s in c with-loop body
15      c = {
16        iv -> WLBc (iv, a, b)
17        | iv < [5]
18      };                  // rc (c) == 1
19   incrc (c, 0);          // referenced in print-call
20   print (c);
21
22   // ——— WL d ———
23   // ...                 // rc (d) == 1
24   incrc (d, 0); print (d);
25
26   // ——— WL e ———
27   // ...                 // rc (d) == 1
28   incrc (e, 0);          // referenced in stencil-call
29   f = stencil (e, 10);
30   incrc (f, 0);          // referenced in return
31   return f;
32 }
```

Listing 2: Example from listing 1 with reference counting

At the beginning of the function we adjust the reference counts of the formal parameters of fun. Array a is referenced 3 times and array b is referenced once. As fun on application consumes its arguments we increment the reference counts of a and b by 2 and 0, respectively. This is because all function arguments have at least a reference count of 1. After each assignment, the reference counts of the let-bound variables are adjusted. In our example, all variables are used exactly once, resulting in increments by 0, *i.e.* no adjustments at all. Every *with*-loop allocates new memory for its result with a reference count of one.

In lines 10–16 we show the reference counting of the first *with*-loop. Each element computation acts like a function call. The conceptual arguments of those functions are the index variable (here iv) and the relatively free variables (here a and b). We make this idea explicit by abstracting the body of all *with*-loops into corresponding functions WLBc, WLBd and WLBe.

Before starting the computation of the array c, the reference counts of the relatively free variables are adjusted by the number of array elements minus one. Within the element computation (the body of WLBc function), as iv occurs twice in a[iv] + b[iv], we increment the reference count of iv by 1. Similarly, the reference counts of a and b are left unmodified as they occur exactly once in the element expression.

Consider an execution where fun is called with reference counts of 1 for both a and b. When reaching the *with*-loop computation in line 14, the reference counts of a and b have been changed to 7 and 5, respectively. Within each element computation, the reference counts of a and b are decremented by one at the end of each corresponding selection. Since we have 5 element computations, the reference count of a is going to be reduced to 2 and b is going to be freed directly after the last element selection.

The *with*-loops in lines 20 and 25 are treated in the same way; so we have omitted the details in listing 2.

Note here, that in this example the arrays c and d will be freed immediately after they are printed.

Reference counting for conditionals requires reference count adjustments of all relatively free variables whenever the individual alternatives are being computed. A more detailed account of reference counting in SaC can be found in [12].

2.3 Improved Reference Counting in SaC

The basic reference counting in SaC has been significantly extended by several optimisations [2, 28]. Our implementation is an extension of the proposed mechanisms that builds on the memory reuse optimisations in [12, 13].

As can be seen from the naïve reference counting in listing 2, we always allocate fresh memory for the result of a *with*-loop. Looking at the example of the *with*-loop in lines 10–16, we can observe that the array b would actually be an excellent choice for memory reuse in all those cases where fun is called with an argument b with reference count 1. Specifically, we could avoid allocating arrays c, d, and e and reuse the memory of array b instead. Not only would it avoid an extra allocations and deallocations, it would also improve the cache locality of the overall code.

To enable such an optimisation we identify arrays that are referenced in the *with*-loop body that

(1) have the same shape as the *with*-loop,
(2) have reference count one in the beginning of the *with*-loop, and
(3) have a well-behaved access pattern.

Techniques to identify the latter can be found in [12, 13]. We refer to such arrays as *reuse candidates* (*RC*s). Once identified, the reuse candidates are kept until runtime. Once the *with*-loop is executed, the reference counts of the reuse candidates are inspected. As soon as a reference count of 1 is found the corresponding memory is reused. Only if no reuse candidate with reference count 1 is found, is new memory allocated.

3 MOTIVATION FOR *EXTENDED* MEMORY REUSE

In the previous section, we have seen how memory can be reused for computing new arrays from existing ones. While this works

fine for the first *with*-loop of our running example, it does not apply for the three other *with*-loops in listing 1. There the array `c` is deallocated during the call to `print` in line 5 although an array of the same size is needed for the computation of the array `d` in line 8. Similarly, `d` is discarded in line 9 just before an array of the same size is needed in line 12.

The loop function `stencil` in lines 19–25 exposes a similar pattern. While the argument array `e` cannot be reused for `f`, which is passed into the next iteration, it could serve as memory for the next instance of `f` to be computed in the next iteration. If we look at an unrolled version of the stencil operation we can see exactly the same situation as in the function body of the function `fun`:

```
f0 = { iv -> e[(iv-1) % 5] + e[(iv+1) % 5]
       | iv < [5] };
f1 = { iv -> f0[(iv-1) % 5] + f0[(iv+1) % 5]
       | iv < [5] };
f2 = { iv -> f1[(iv-1) % 5] + f1[(iv+1) % 5]
       | iv < [5] };
// ...
```

The memory of `e` can be reused for f_1, the memory of f_0 can be reused for f_2, *etc.* Effectively, this would lead to two memory locations being used for the *with*-loop computation in an alternating fashion.

Our goal here is to achieve the same effect without the need for explicitly unrolling of the loop. This can be achieved by identifying loop functions that contain such *with*-loops and extending their signatures. For our running example, we aim at an extended stencil function of the form:

```
int[5] stencil1 (int[5] e, int n, int[5] f0)
{
  f = { iv -> e[(iv-1) % 5] + e[(iv+1) % 5]
        | iv < [5] };

  r = n == 0 ? f : stencil1 (f, n-1, e);
  return r;
}
```

The extra parameter f_0 effectively provides a pointer to memory that can be reused for the computation of `f`. In the subsequent iteration, this memory is provided from the array `e` whose lifetime is now extended into the next iteration.

Note here that this completely elides the need to interact with the heap manager within this loop. In particular in the context of executions on accelerators such as GPUs we expect to see a noticeable impact for this optimisation.

In the next section, we provide an algorithmic description which systematically transforms programs like our running example into a form where memory is being reused within the scope of individual functions and across loops.

4 EXTENDED MEMORY REUSE

The basic idea for implementing *EMR* is to extend the idea of reuse candidates as outlined in section 2.3. Instead of restricting the compiler to only use arrays as reuse candidates whose last reference is within the with-loop body in a suitable way, we would like to also include arrays that are neither referenced in the loop body nor within the remainder of the function body. We call such arrays *Extended Reuse Candidates* or *ERC*s for short. Once these are identified, the idea is to inspect the reference counts of such ERC at runtime and, similar to the reuse candidates from section 2.3, reuse their memory in case the reference count is 1.

To do so, we first collect arrays that are of suitable size and that have been defined prior to the with-loop in question. This happens in a phase we call *ERC inference*. Once we have collected such potential *ERC*s, we filter out non-suitable candidates in a second phase named *ERC filtering*. Finally, we adjust the function signatures and function calls to the tail-end recursive functions that represent our loops. We call this phase *ERC loop optimisation*.

4.1 Overview

Before providing a more formal description of these three phases, we sketch their intended behaviour by using our running example.

4.1.1 ERC inference. First of all, we annotate every *with*-loop with the set of potential *ERC*s. These are variables that have been defined before the *with*-loop, whose shape is identical to that of the *with*-loop result, and which are not referenced within the *with*-loop itself. For our running example from listing 1, this leads to the following annotations of the *with*-loops with *RC*s and *ERC*s:

```
1  int[5] fun (int[5] a, int[5] b)
2  {
3    c = { /*.*/ a[iv] + b[iv] /*.*/ }; // RC={b} ERC={}
4    print (c);
5
6    d = { /*.*/ a[(iv-1) % 5] /*.*/ }; // RC={} ERC={c,b}
7    print (d);
8
9    e = { /*.*/ a[(iv+1) % 5] /*.*/ }; // RC={} ERC={d,c,b}
10
11   f = stencil (e, 10);
12   return f;
13 }
```

As can be seen, for the *with*-loop in line 3, `b` is a *RC*, as it has a well-behaved access and is not referenced in the function body any further. The array `a` cannot serve as *RC* in line 3 since it is referenced later. In the remaining *with*-loops the array `a` cannot serve as a *RC* due to the access patterns that stem from the rotations.

The *ERC* inference now traverses the functions top to bottom, collects the variables that denote arrays of suitable shape and attaches those as *ERC*s to *with*-loops that are not contained in the corresponding *with*-loop body.

4.1.2 ERC filtering. The *ERC*s inferred by the previous step require further filtering as the inference tends to over-approximate and can sometimes pick candidates which are referenced at a later stage. When an *ERC* is selected for a given *with*-loop, from that point onward it *is* a reference to the resulting array. If at a later stage we select the same *ERC*, we cause a conflict that can lead to an additional allocation. Consequently, an array that serves as an *ERC* for the *with*-loop in line 9 cannot be used as an *ERC* for the *with*-loops in lines 3 or 6; an array that serves as an *ERC* for the *with*-loop in line 6 cannot be used as an *ERC* for the *with*-loop in

line 3. We achieve this by performing a bottom-up traversal that filters out those *ERC*s that are referenced within the remainder of the function body. Once this is done we pick the first remaining *ERC*, provided the *with*-loop does not have an *RC*. All other *ERC*s are filtered out as well. Then we add the chosen *ERC* to the set of variables that are referenced and continue traversing bottom-up. By picking the first *ERC* we ensure that the most recently defined array is always chosen as the *ERC*, which implicitly minimises the extension of array lifetimes. For our running example this results in the following *ERC* annotations:

```
1  int[5] fun (int[5] a, int[5] b)
2  {
3    c = { /*.*/ a[iv] + b[iv] /*.*/ }; // RC={b} ERC={}
4    print (c);
5
6    d = { /*.*/ a[(iv-1) % 5] /*.*/ }; // RC={} ERC={c}
7    print (d);
8
9    e = { /*.*/ a[(iv+1) % 5] /*.*/ }; // RC={} ERC={d}
10
11   f = stencil (e, 10);
12   return f;
13 }
```

4.1.3 ERC loop optimisation. When applying the traversals explained so far to the loop function `stencil`, the *with*-loop within that function remains without an *RC* or *ERC* since there simply is no array that possibly can be reused. The task of this phase is to identify *with*-loops without *RC*s and *ERC*s within loop functions, invent new identifier names, and add these identifiers as *ERC*s to the corresponding *with*-loops. This traversal also adjusts the function signatures of the loop functions as well as all applications of these functions accordingly. For our running example we obtain for the loop function:

```
1  int[5] stencil (int[5] e, int n,
2                  int[5] f0 /* Fresh variable */)
3  {
4    /* RC={}, ERC={f0} */
5    f = { /*.*/ e[(iv-1) % 5] + e[(iv+1) % 5] /*.*/ };
6
7    r =  n == 0
8      ? f
9      : stencil (f, n-1, e /* shape(e) == shape(f0) */);
10   return r;
11 }
```

Note that the choice of variable to expand the recursive function call only needs to match the type and shape of what is needed and it must not be used as existing argument within the recursive call. In what order this variable is created in the function body does not matter.

After all loop functions are extended, we need to extend their call sites as well. Given the type and shape information of the newly added function arguments, we can either find variables in the current context or create new arrays before the function call. For our example, we obtain:

```
9   // ...
10  f0 = { iv -> 0 | iv < [5] };
11  f = stencil (e, 10, f0);
12  // ...
```

4.2 Algorithmic Formulation of the Transformations

We base our transformations on the language core delineated in section 2. More specifically, we make the following assumptions:

(1) a program is a list of functions where each function consists of a return type, arguments and body;
(2) the body of each function is a list of assignment statements in static single assignment form and a single return statement at the end of that list (note that functions like `print` assign the unit type value to some temporary variable);
(3) the right-hand side of each assignment is either a *with*-loop, a function application, or a conditional (all conditionals are in the form x ? e_1 : e_2 where x is a variable of type bool and e_1 and e_2 are expressions);
(4) loop functions are tail-recursive functions where the recursive call has been identified;

Note here, that although this core language is designed to reflect SaC, the only SaC-specific construct is the *with*-loop. Conceptually, this language construct can be replaced by any other language construct that creates a reference-counted data structure. With such a modification our transformation is immediately applicable to any other programming language that uses reference counting as the garbage collection mechanism.

4.2.1 ERC Inference. The *ERC* inference is shown in algorithm 1. It expects a function definition (F_d) and computes *ERC* annotations for all *with*-loops within that function. We represent these annotations as **EC**, a mapping from expressions to variable lists.

During the top-down traversal of the function body we create a local variable list C which we use to store all candidates found so far. We define a local recursive function T that operates on expressions: for *with*-loops it computes the corresponding *ERC*s; for conditionals it recursively inspects both branches. In the former case we filter the current C by removing all the variables that are free in the body of the *with*-loop or whose shapes do not match the shape of the return value of the *with*-loop. As conditionals cannot have local variable bindings, we simply annotate the *with*-loops in the individual branches with the *ERC*s.

The actual top-down traversal is defined in lines 11–12. We iterate through all assignments in F_d, and apply T to *expr* looking for assignments made by either *with*-loops or function applications. Once done with the expression of the assignment, we add the defined variable *var* to the set of candidates C. Once the entire function body has been traversed, all found *ERC*s are stored in **EC** which we will refer to in the subsequent transformations.

4.2.2 ERC Filtering. The *ERC* filtering is shown in algorithm 2. It relies on mappings **EC** and **RC** which hold the *ERC*s inferred by

Algorithm 1: Extended Reuse Candidate Inference

```
   Input: A function definition F_d
1  begin
2      Let EC be an empty mapping
         (expression ↦ variable list)
3      Let C be the found candidates of F_d
4      function T(expr) is
5          switch expr do
6              case expr is a with-loop do
7                  ℰℛC = Filter(
                     x ⇒ x ∉ FreeVariables(expr)
                     and Shape(x) == Shape(expr), C)
8                  Add expr ↦ ℰℛC to EC
9              case expr is a conditional x ? e_1 : e_2 do
10                 T(e_1); T(e_2)
11     foreach (var = expr) in F_d do          /* top-down */
12         T(expr); C = var ++ C
   Output: The updated mapping EC
```

Algorithm 2: Extended Reuse Candidate Filtering

```
   Input: A function definition F_d
          A mapping EC (expression ↦ variable list)
          A mapping RC (expression ↦ variable list)
1  begin
2      Let 𝒱ℛ be the return value of F_d
3      function T(expr) is
4          switch expr do
5              case expr is a with-loop do
6                  ℰℛC = Filter(x ⇒ x ∉ 𝒱ℛ, EC[expr])
7                  𝒱 = FreeVariables(expr body)
8                  if RC[expr] = ∅ ∧ ℰℛC ≠ ∅ then
9                      EC[expr] = ℰℛC_0
10                     𝒱 = ℰℛC_0 ∪ 𝒱
11                 else
12                     EC[expr] = ∅
13                 return 𝒱
14             case expr is conditional x ? e_1 : e_2 do
15                 return T(e_1)++T(e_2)
16             otherwise do
17                 return FreeVariables(expr)
18     foreach (var = expr) in F_d do          /* bottom-up */
19         𝒱ℛ = T(expr) ++ 𝒱ℛ
   Output: The updated mapping EC
```

algorithm 1 and *RC*s for all expressions, respectively. As a result of this bottom-up traversal, the mapping **EC** is being updated.

During the traversal, a list of referenced variables $\mathcal{VR}$ is maintained which is initialised with the function's return value. This list is used to do the actual filtering of *ERC*s for *with*-loops. Similar to the previous algorithm, we define a local function T which we map over all assignments in a bottom-up fashion (lines 18–19). When encountering a *with*-loop, we first filter out all *ERC*s that are referenced in the code below the current assigment (line 6). We then collect the free variables contained in the *with*-loop body in $\mathcal{V}$ which we will append to our global list $\mathcal{VR}$ once we are done with the *with*-loop. If our filtering of **EC**[*expr*] left an *ERC* and there is no *RC*, we pick the first *ERC*, update **EC**[*expr*] accordingly, and we add it to the list of referenced variables in $\mathcal{V}$. Otherwise, we set **EC**[*expr*] to empty. Upon return of T the variables referenced in the *with*-loop including the remaining *ERC* contained in $\mathcal{V}$ are added to $\mathcal{VR}$ before continuing with the bottom-up traversal.

4.2.3 ERC Loop Optimisation. *ERC* loop optimisation is shown in two algorithms, algorithm 3 and algorithm 4. This split is purely for presentational purposes as we describe our algorithms on a per-function basis and *ERC* loop optimisation requires changes of two functions, the actual loop function and the function that calls the loop-function. The former is described in algorithm 3 and it has to happen before the corresponding loop function call is modified as described in algorithm 4.

Transformation of Loop Functions. This top-down traversal utilises the mappings **EC** and **RC** for *ERC*s and *RC*s for each *with*-loop. Similar to the previous algorithms, we map a local function T successively on all assignments of the function F_d, as shown in algorithm 3. We use a list $\mathcal{TR}$ of variable-expression pairs to store fresh variables that will be used as *ERC*s for *with*-loops without *RC*s or *ERC*s, if possible. When we come across a *with*-loop, we check if it has any *RC*s or *ERC*s. If it has *none*, we create a variable for a new array with the same type and shape as the return value of the *with*-loop and put this jointly with a reference to the *with*-loop into $\mathcal{TR}$. After the body of F_d is traversed, we use $\mathcal{TR}$ to extend the F_d arguments with the variables in $\mathcal{TR}$.

When traversing the recursive call, we iterate over $\mathcal{TR}$ and search the body of F_d for the variables of the corresponding type and shape that can be used as arguments of the extended function call. This search happens in the `FindMatching` helper function. If the search is not successful, we elide the variable-expression pair from $\mathcal{TR}$. Otherwise, we append the found variable $\mathcal{N}_a$ to the arguments of the function application and we add the variable from $\mathcal{TR}$ to *ERC*s of the *with*-loop referred in the variable-expression pair of $\mathcal{TR}$. This way, we make sure the number of added arguments to the recursive call matches the number of entries in $\mathcal{TR}$.

Adjustment of Loop Function Calls. The adjustment of the external loop function calls is shown in algorithm 4. It assumes that all loop functions have already been adjusted through algorithm 3. Again, we iterate through the assignments within the body of F_d again searching for a function call to a loop function. Once found, we check whether the function signature has changed during the application of algorithm 3. If it has, we use a helper function `CreateArgs` to create the missing number of arrays of the right type and shape and append these to the loop function call.

Algorithm 3: *ERC* Loop Optimisation for Loop Functions

Input: A loop function definition F_d
A mapping **EC** ($expression \mapsto variable\ list$)
A mapping **RC** ($expression \mapsto variable\ list$)

```
1  begin
2      Let TR be a empty list tuples (variable × expression)
3      function T(expr) is
4          switch expr do
5              case expr is a with-loop do
6                  if EC[expr] = ∅ and RC[expr] = ∅ then
7                      V_rc = fresh variable to store expr
8                      TR ++ [(V_rc, expr)]
9              case expr is a function application do
10                 if expr is the recursive call then
11                     foreach (V_rc, expr') in TR do
12                         N_a = FindMatching(V_rc, F_d)
13                         if N_a = ∅ then
14                             Delete (V_rc, expr') from TR
15                         else
16                             Append N_a to arguments of expr
17                             EC[expr'] = V_rc
18             case expr is conditional x ? e_1 : e_2 do
19                 T(e_1); T(e_2)
20     foreach (var = expr) in F_d do          /* top-down */
21         T(expr)
22     foreach (V_rc, expr') in TR do
23         Append V_rc to arguments of F_d
```

Output: The updated mapping **EC**
The updated function F_d

Algorithm 4: *ERC* Loop Optimisation - Adjust Loop Calls

Input: A function definition F_d

```
1  begin
2      function T(expr) is
3          switch expr do
4              case expr is a loop function f application and
                expr no. arguments < f no. arguments do
5                  N_a = CreateArgs(Δ no. arguments)
6                  Append N_a to arguments of expr
7              case expr is conditional x ? e_1 : e_2 do
8                  T(e_1); T(e_2)
9      foreach (var = expr) in F_d do          /* top-down */
10         T(expr)
```

Output: The updated function F_d

5 EXPERIMENTAL EVALUATION

We implemented the *EMR* optimisation in a feature branch of the sac2c compiler based on sac2c 1.3.2. We evaluate the effects of *EMR* on a variety of benchmarks from different benchmark suites which represent various types of computations.

5.1 Experimental Setup

The benchmarks are compiled to run both sequentially on an AMD Opteron 6376 and on an NVIDIA K20 GPU (driver version 384.81) using GCC 4.8.5 and CUDA 9.0, respectively. Memory measurements are generated using compiler-instrumented SAC code through sac2c flag '`-profile m`' as well as NVIDIA's profiling tool `nvprof`. Runtime measurements are wall-clock times observed by the GNU `time` command. We present the average of five runs, and show the extreme values we found as well, in order to obtain an idea of the measurement noise.

The benchmarks include basic ones such as matrix multiplication and matrix relaxation. We also use a kernel from a real-world application by the British Geological Survey (BGS) called the Global Geomagnetic Model, which was translated to SAC and evaluated in [32]. Additionally we include benchmarks[2] from the Livermore Loops suite [23], the Rodinia Benchmark suite [6], and the NAS Parallel Benchmark suite [1]. A list of benchmarks and their respective problem sizes is given in table 1.

Some of the benchmarks we use have several different implementations, and we indicate these by adding suffixes to their names. For implementations that closely reflect their C counterpart, which use no *with*-loops, we use *C*. The suffix *APL* indicates an APL-like implementation that uses SAC implementations of APL operators. The suffix *N* indicates a naïve SAC implementation whereas the suffix *SC* is a more optimal SAC implementation. Additionally, for the Rodinia Pathfinder benchmark we use two implementations as one of them performs better on a GPU than on a shared-memory architecture [3]. These are suffixed with *nCD* and *CD*, respectively.

Our first experiment analyses the overall impact of *EMR* on the number of memory allocations that are being performed during the lifetime of the chosen benchmarks. Figure 1 shows for each benchmark the total number of allocations made in four different columns: the first two columns show the number of allocations during sequential executions, first without *EMR* then with *EMR* being applied. The second set of columns presents the corresponding numbers for executions on the GPU.

Note here that the y-axis is a logarithmic scale. For the first benchmark, the BGS-Kernel, we can see that for both, sequential execution and execution on the GPU, the number of memory allocation shrinks by more than two orders of magnitude, from roughly 2.5×10^6 to 10^4.

We observe that 10 out of the 37 benchmarks benefit from *EMR*. The other 27 benchmarks essentially remain the same or have small improvements only. 9 out of the 10 benchmarks that benefit from *EMR* have improvements of more than 2 orders of magnitude. In 8 out of the 10 cases the number of allocations decreases to levels that are very similar to those of manually written code that have explicit memory management.

Only Gauss-Jordan seems to benefit less significantly. Closer inspection reveals that, nevertheless, in the sequential case the

[2] More information on the SAC implementations of the Livermore Loops, Rodinia Benchmarks, and the NAS Parallel Benchmarks (as well as other benchmarks) are available at https://github.com/SacBase.

Suite	Benchmark	Problem Size
Livermore Loops	Loop01	100001 vector
	Loop02	100001 vector
	Loop03	100001 vector
	Loop04	10001 vector
	Loop05	100001 vector
	Loop07	100001 × 64 array
	Loop08-split	1001 × 64 array
Rodinia Benchmarks	Back-propagation	65536 input units
	Hotspot	1024 × 1024 grid
	Kmeans	34 features, 5 clusters
	Leukocytes	640 × 480 image, 88 cells
	LUD	2048 × 2048 matrix
	Needleman-Wunsch	24 × 24 grid
	Particle Filter	10 frames at 128 × 128, 1000 particles
	Pathfinder	100 × 512000 grid
	SRAD	2048 × 2048 matrix
NAS Parallel	Conjugate Gradient	14000, 75000, and 150000
	Embarrassingly Parallel	2^{24}
	Multigrid	128^3
Other	Matrix Multiplication	1000 × 1000 matrices
	Matrix Relaxation	10000 × 10000 matrix
	Gauss-Jordon	500 × 500 matrix
	BGS Kernel	1000 × 1000 matrix

Table 1: Benchmarks and their Problem Sizes

number of allocations is reduced to 50% of the unoptimised version and for the GPU case, the number of allocations is reduced by 30%.

The fact that we see different effects for the two architectures has two reasons: Firstly, we only measure the memory allocations on the machine that executes the data-parallel kernels. Consequently, the GPU figures are potentially smaller. Secondly, we perform different code optimisations depending on the target platform which may impact the number of intermediate arrays materialised in memory.

In this context, two benchmarks stick out: Relax-Expo only gains on the sequential platform and the Rodinia implementation of Needleman-Wunsch benefits on the GPU only. In the case of Relax-Expo, the reason for this behaviour lies in the use of a reduction operation in the innermost loop which cannot be computed on the GPU platform but must be done on the host. This prevents the application of *EMR*, thereby leading to significant communication between the host and the GPU device. In the case of Needleman-Wunsch, a variance in the applied optimisations is responsible for the difference in effectiveness of *EMR*.

5.2 Impact of *EMR* on Memory Pressure

While the total number of allocations gives us an idea of how many calls to the heap manager could be eliminated, it is not clear how relevant these are in terms of the sizes of memory allocations that could be avoided. In order to quantify this aspect, we measured the sum of all memory allocation requests. Figure 2 shows these numbers, again using a logarithmic scale. For these results, we only show the numbers for the sequential execution in order to make sure that we capture all allocations rather than only those on the architecture where the data-parallel computation is being executed, as is the case with using a GPU.

Here, we can observe that all benchmarks where benefits were found in terms of the total number of allocations, also benefit by a reduction to the total amount of memory allocation requests. The reductions are typically bigger than 2 orders of magnitude. An excellent example is the Livermore Loop 8, where the total sum of allocations shrinks by almost 6 orders of magnitude from 380GB down to 500KB. This suggests that the optimisation of innermost loops is effective in these examples.

Another interesting aspect is the observation that the NAS Conjugate Gradient actually benefits from *EMR* as well, which was not evident from the total number of allocations. From this experiment we learn that the total amount of memory that is requested by NAS CG is cut by roughly a factor of 4.

5.3 Impact of *EMR* on the Runtime

After verifying that *EMR* actually reduces the number of memory allocations and that these are of relevant size, we investigate the impact of *EMR* on the runtime for both architectures.Our measurements are presented in fig. 3 which shows the speedup of *EMR* for all benchmarks on both architectures. The baseline for both architectures is the corresponding unoptimised runtime. The horizontal bars indicate the maximum and minimum speedups obtained from the maxima and minima of 5 runs each.

The first observation we can make is that the runtime gains are mainly realised on the GPU platform. This does not really come as a surprise as runtime experiments with our GPU backend triggered this line of research in the first place. The need to perform allocations and deallocations of GPU memory from the host has a very noticeable effect and the change of memory often also generates the need for superfluous data-transfers between host and device. 7 out of the 9 applications that benefited in terms of memory allocations on the GPU show speedups between a factor of 2 and a factor of 4 against the unoptimised versions.

The two applications that do not benefit on GPUs despite savings in terms of memory allocations are Gauss-Jordan and Rodinia's Pathfinder. For these two examples the optimisations apply to parts of the application that are not the main hotspots. Additionally, we observe a slowdown in Relax-Expo although the overall allocations in the GPU didn't decrease. As previously mentioned this is due to the additional communication caused by a reduction operation.

Finally, we observe some speedups for the sequential CPU executions as well. 4 of the 10 applications that benefit from fewer memory allocations expose performance gains between 10% and 50%.

5.4 Impact of *EMR* on the Memory Footprint

Our final experiment concerns the memory footprint of the benchmarks. Since *EMR* extends the lifetime of variables it can potentially lead to an increase in the maximum of memory that is allocated at some time during execution. Our measurements indicate that none of the examples suffered from any increment of the memory footprint.

6 DISCUSSION

The evaluation in the previous section shows that *EMR* decreases the number of memory allocations by several orders of magnitude for more than a quarter of the compute intensive kernels we investigated. Most of these come down to levels of allocations that are similar to what codes with hand-written explicit memory management would look like. About half of the examples we investigated

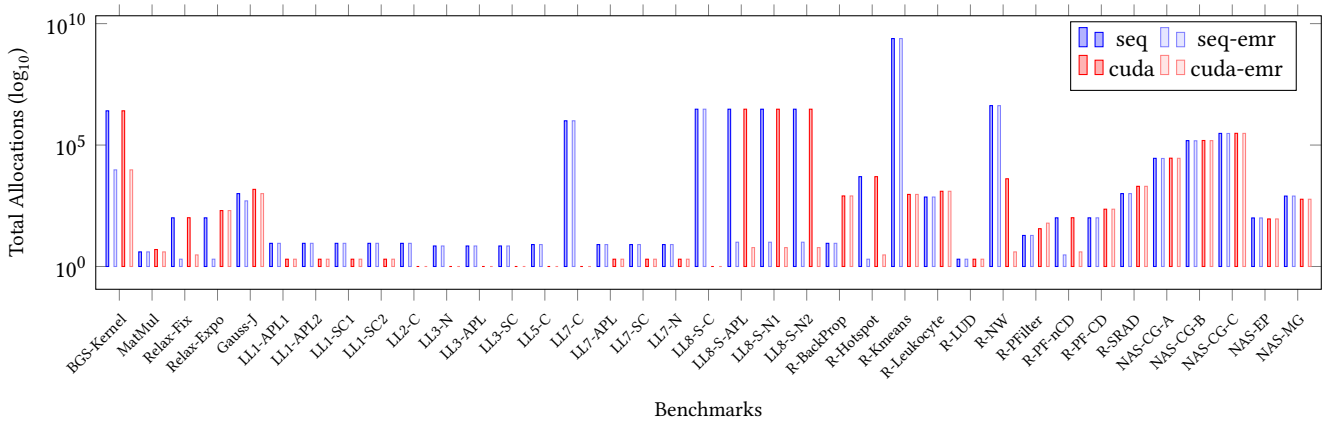

Figure 1: Allocation Count (normal vs. optimised)[+]

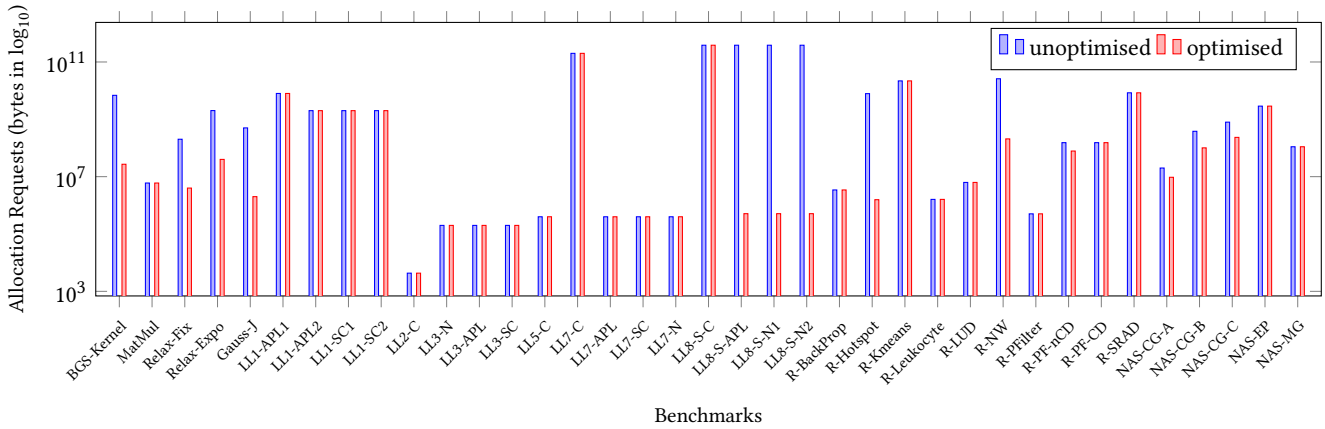

Figure 2: Total Memory Allocation Requests for *sequential* Target (normal vs. optimised)[+]

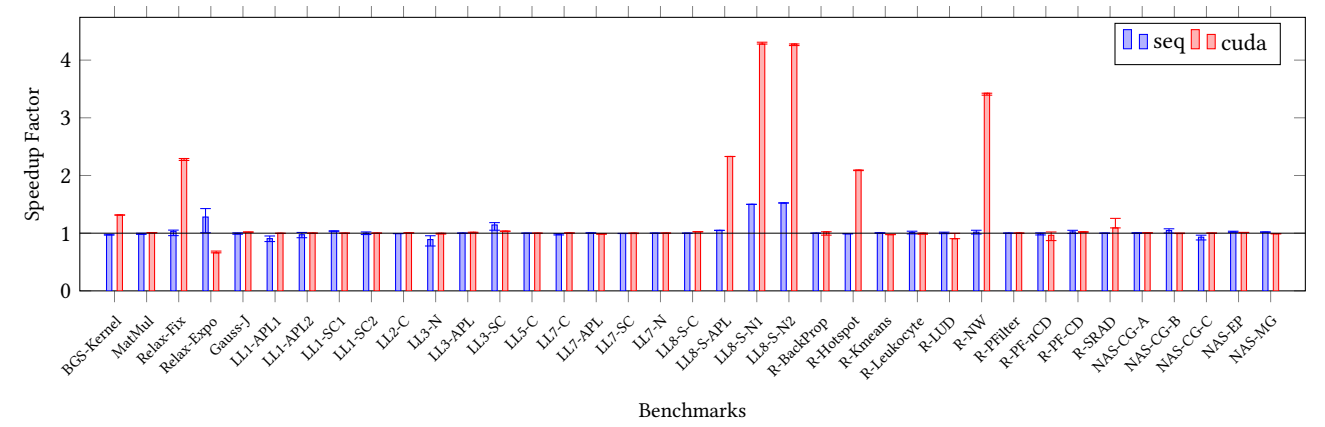

Figure 3: Runtime Speedup (normal vs. optimised)[+]

[+] Some of the benchmarks have more than one implementation-style. These alternatives are marked with suffixes: *-SC* for SaC, *-N* for naïve SaC, *-APL* for APL, *-C* for C/C++.

do not benefit *EMR* simply because the pre-existing reuse analysis explained in section 2.3 already brings down the allocations to the hand-written level.

An analysis of the remaining quarter of applications identifies two main reasons why deallocations and subsequent allocations remain within innermost loops.

The first reason is that some of the benchmarks are written in a C-style using *for*-loops rather than *with*-loops. For those cases to benefit from *EMR*, we need an extension of *EMR* to deal with these loops in the same way as we deal with *with*-loops. While this is conceptually easy, given that the underlying operation is a scalar modification of the array, the implementation effort that would be required is non-trivial. Since most SaC programs are defined in terms of *with*-loops whenever possible, we decided to leave this extension as future work.

The second reason lies in the fact that several of our benchmarks operate on differently sized arrays within the innermost loops. Examples for these are Gauss-Jordan and Conjugate Gradient. Trying to capture these cases poses a bigger challenge. It would require the memory system to allow memory chunks to be larger than what is strictly needed at allocation time. This immediately increases the potential danger of memory footprint increases. Furthermore, in case of growing memory demands, it would require further analyses to predict the maximum memory need over the lifetime of loops. Given these major challenges and it not being clear whether these applications would benefit significantly, support for varying-size memory reuse remains outside of the scope of this paper.

Though the evaluation of *EMR* shows improvements and effectiveness throughout, two aspects remain as possible concerns. The first relates to the way *ERC* filtering is done and the second relates to the change of memory footprint of applications.

6.1 Choice of Extended Reuse Candidates

The filtering of *ERC*s from section 4.2 currently picks the most recent possible candidate. While our experimentation shows that this works well in practice, in theory, this can lead to sub-optimal choices. If we have more choices than we need for memory reuse the choice of the first candidate may undermine the potential success of memory reuse. Consider the following example:

```
int[10] strange (int[10] a, int[10] b)
{
   c = { iv -> 0 | iv < [10] };
   return c;
}
```

Here, after inference and initial filtering, we have *ERC*s `a` and `b` to choose from. Let us assume *EMR* has chosen `b`, if it turns out at runtime that the reference count of `a` is 1 and that of `b` is 2, we forgo the opportunity for reuse. While it would be possible to maintain both choices until runtime in this case, a slight variation of the example shows that this cannot be done in general. Consider this variant:

```
int strange(int[10] a, int[10] b)
{
   c = { iv -> 0 | iv < [10] };
   d = { iv -> 1 | iv < [10] };
   r = f (c, d);
   return r;
}
```

Now let us assume we would keep both, `a` and `b`, as *ERC*s for both *with*-loops. Furthermore, let us assume `a` and `b` at runtime both come in with a reference count of 1. If the *with*-loop for `c` chooses to reuse `a`, then after the computation of `c`, array `c` will have a reference count of 1. Since `c` reuses `a`, an inspection of the reference count of `a` which conceptually is no longer exists will still yield 1. From the perspective of the *with*-loop that computes `d` this suggests that it can use `a` for memory reuse as well which would simply be wrong.

Consequently, we either need to establish a mechanism that communicates the choice made by the first *with*-loop to the second *with*-loop, or we need to ensure that the sets of *ERC*s after filtering are disjoint. Our proposed approach opts for the latter. We consider loosing out on options in situations like our first example here less of a concern since, in the worst case, it only results in the same reuse behaviour as without using *EMR* in the first place.

6.2 Memory Footprint

By reusing memory we increase the lifetime of allocated memory. This potentially translates into a larger utilisation of memory over the runtime of the program. Consider the following example:

```
{
   // ...
   c = { iv -> 0 | iv < [1000] };
   print (c);
   d = { iv -> 1 | iv < [1001] }
   print (d);
   e = { iv -> 2 | iv < [1000] };
   print (e);
   // ...
}
```

Assuming that we have no further references to the arrays `c` and `d`, compilation without *EMR* would result in a memory footprint of this code snippet of `1001×sizeof (int)`. Using *EMR*, this figure increases to `2001×sizeof (int)`. If we construct this within a non-tail-end-recursive function, we can construct an example where the footprint increases linear with the number of recursive calls:

```
int elephant (int n)
{
   c = { iv -> 0 | iv < [1000] };
   print (c);
   res = n == 0 ? 0 : elephant (n-1);
   e = { iv -> 2 | iv < [1000] };
   print (e);
   return res;
}
```

Here the memory footprint of the function `elephant` without *EMR* is constant while the application of *EMR* renders the memory footprint linear to the value of the parameter `n` as all instances of the body will keep the memory allocated for `c` for reuse in `e` while the recursive calls happen.

Although we have not seen this effect in any of the examples we ran, it would be better if we could rule out such a memory footprint increase by construction. One way to tackle this problem would be to rule out all *ERC*s whose scope would need to be extended across any code that potentially allocates memory. Unfortunately, this would rule out almost all examples, even the most simple ones. The reason is that we cannot guarantee that a function call such as `print (a)` does not allocate some memory *after* deallocating the array `a`. This would require sophisticated analyses and might still turn out to be too conservative in many practical cases.

7 RELATED WORK

Most of the literature that deals with improving the performance of reference counting based garbage collection focuses on decreasing overheads or improving the inference of in-place update opportunities.

In the work by Park and Goldberg [25], the authors introduce a compile-time based analysis to track the lifetime of references. With this information, the authors can determine points within the code where a reference counter need not be updated. For instances, when a reference is made and then quickly discarded before the total number of references for the memory object reaches zero. By avoiding these reference counter updates, the author's analysis reduces the runtime overheads of reference counting.

Shahriyar et al. [30] propose a solution to make reference counting more amenable to high performance environments. In the context of Java, they looking at an existing reference count implementation that uses cyclical garbage collection. They perform a series of analyses looking at intrinsic properties of the reference counting mechanism such as counter storage and reference count operations at runtime. From this they introduce and implement several optimisations which, *a.* change how memory objects are tracked for cyclical garbage collections, and *b.* lazily allocate memory objects at the point where their counter is initially incremented. Their experiments demonstrate a positive speedup for their optimisations.

For GPU based applications, reference counting can offer a possible way to handle communication between the host and GPU device, as is done by Jablin et al. [19]. Here the authors introduce a runtime-library and serious of code-transformations that they call the CPU-GPU Communication Manager. The runtime-library provides wrappers for `malloc` and `free` which do reference counting, and the code-transformations move communication primitives around to minimise communication. For a set of benchmark, the authors solution results in speedups.

Outside of reference counting based garbage collection, other techniques exist which try to reuse heap allocated memory. For instance Hamilton and Jones [16] introduce a compile-time based solution for a functional program that searches the code for in-place update opportunities. They do this by applying *necessity* analysis to determine which parts of a list are needed and where. Whenever they find a list part that has no further necessity, that list part can be safely reused.

Similarly, Kågedal and Debray [20] extend on this idea in a single-assignment context. They provide a code-transformation that introduces runtime-time primitives to the code that either perform a new allocation or do an in-place update. Their code-transformation is able to deal with iterative constructs, and can move copy-operations out of inner loops. Their experiments demonstrate large speedups with this technique.

Alternatively, Hage and Holdermans [15] do not do any compile-time analysis but instead introduce a new construct to a lazy functional programming language. The construct is used to mark opportunities for memory reuse, and what structure to use for this. They provide a set of semantics and type rules to ensure that reuse only occurs when it is safe to do so. Lee et al. [22] do something similar for a ML-like functional language. Here though they perform a static memory analysis, after which the inferred information from this is used to place free primitives which destructively update nodes in a list.

8 CONCLUSION AND FUTURE WORK

In this paper we propose a novel compiler optimisation to increase memory reuse. One of the main insights that made this work possible is the observation that small increases in the lifetime of an object offers large opportunities for memory reuse.

We implement the proposed optimisation in the context of the SaC compiler. The underlying memory management of SaC is based on reference counting. This makes the entire analysis easier, but in no way restricts us to either SaC or reference counting to apply the proposed technique in other contexts.

We evaluate the effect of the proposed transformation by running a set of 37 benchmarks on CPUs and GPUs. Most of benchmarks come from Livermore Loops, Rodinia and NAS benchmark suites which are typically used when evaluating high-performance compilers.

In our experiments we look for changes in the number of memory allocations, the amount of totally allocated memory and runtime performance. On all the three fronts the majority of benchmarks show either no changes or significant improvements. In 8 benchmarks, the number of memory allocations comes down by several orders of magnitude, ending up close to what we might find in hand-written codes. The runtime performance on GPUs for six benchmarks improves between 2× to 4× while only one benchmark shows a slowdown when running on a GPU due to the interplay of *EMR* with the code generation scheme for reductions on GPUs.

We also see some improvements for executions on CPUs albeit on a more moderate level leading to speedups between 10% and 50%.

The observed results suggest that the first step towards better memory reuse is successful. At the same time this opens up a lot of opportunities for future work.

There are two main directions of the future work we would like to pursue. First, we would like to further improve the applicability of *EMR*. As explained in section 6, our optimisation is limited to the case where we are dealing with identical array sizes. It just naturally happens that in high-performance codes data objects of the same size are used over and over again. If this is not the case, it would be desirable to develop some form of cost model that allows an extended version of *EMR* to decide whether to reuse memory that is larger than needed or to allocate more memory than needed, just for later reuse.

Secondly, we believe that the underlying memory reuse pattern that we build on can be used in other contexts. For example we could envision a static analyser for languages with explicit memory allocations, *e.g.* C/C++ family. In our experiments we have seen that memory reallocation is relatively cheap on CPUs, but on GPUs it can have a significant overhead. Furthermore, memory allocations on GPUs are always done from the host, and they cannot be done asynchronously. So, given a CUDA/OpenCL program where deallocation/reallocation of the same size happens between the kernels, we can apply our mechanism and replace the `free (x); y = malloc (sizeof (x))` with `y = x`.

At the same time, languages like C++ have smart pointers, some of which use reference counting which makes our technique immediately applicable.

Finally, explicit lifetime annotations in Rust may be used to manually control the lifetime of the memory object. We can envision a static analyser that identifies the cases that are described in the paper, and uses lifetime annotations to implement the proposed reuse technique.

ACKNOWLEDGMENTS

This work is supported by the Engineering and Physical Sciences Research Council through grants EP/L00058X/1 and EP/L016834/1. We also made use of Edinburgh Centre for Robotics' Robotarium Cluster [9] located at Heriot-Watt University, Edinburgh, UK for our experiments.

REFERENCES

[1] David H Bailey, Eric Barszcz, John T Barton, David S Browning, Robert L Carter, Leonardo Dagum, Rod A Fatoohi, Paul O Frederickson, Thomas A Lasinski, Rob S Schreiber, et al. 1991. The NAS Parallel Benchmarks. *The International Journal of Supercomputing Applications* 5, 3 (1991), 63–73.

[2] Robert Bernecky, Stephan Herhut, Sven-Bodo Scholz, Kai Trojahner, Clemens Grelck, and Alex Shafarenko. 2007. Index Vector Elimination: Making Index Vectors Affordable. In *Implementation and Application of Functional Languages, 18th International Symposium (IFL'06), Budapest, Hungary, Revised Selected Papers (Lecture Notes in Computer Science)*, Zoltan Horváth, Viktória Zsók, and Andrew Butterfield (Eds.), Vol. 4449. Springer, 19–36. https://doi.org/10.1007/978-3-540-74130-5_2

[3] Robert Bernecky and Sven-Bodo Scholz. 2015. Abstract Expressionism for Parallel Performance. In *Proceedings of the 2nd ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming*. ACM, 54–59. https://doi.org/10.1145/2774959.2774962

[4] David C. Cann. 1989. *Compilation Techniques for High-performance Applicative Computation*. Ph.D. Dissertation. Fort Collins, CO, USA. AAI9007070.

[5] David C. Cann and Paraskevas Evripidou. 1995. Advanced array optimizations for high performance functional languages. *IEEE Transactions on Parallel and Distributed Systems* 6, 3 (March 1995), 229–239. https://doi.org/10.1109/71.372771

[6] Shuai Che, M. Boyer, Jiayuan Meng, D. Tarjan, J. W. Sheaffer, Sang-Ha Lee, and K. Skadron. 2009. Rodinia: A benchmark suite for heterogeneous computing. In *IEEE International Symposium on Workload Characterization (IISWC '09)*. 44–54. https://doi.org/10.1109/IISWC.2009.5306797

[7] David L. Detlefs, Paul A. Martin, Mark Moir, and Guy L. Steele Jr. 2002. Lock-free reference counting. *Distributed Computing* 15, 4 (01 Dec 2002), 255–271. https://doi.org/10.1007/s00446-002-0079-z

[8] Steven M. Fitzgerald and Rodney R. Oldehoeft. 1996. Update-in-place Analysis for True Multidimensional Arrays. *Sci. Program.* 5, 2 (July 1996), 147–160. https://doi.org/10.1155/1996/493673

[9] Edinburgh Centre for Robotics. 2014. Robotarium Cluster. https://doi.org/10.5281/zenodo.1455754 The cluster is part of the EPSRC Centre for Doctoral Training in Robotics and Autonomous Systems (RAS) in Edinburgh grant (EP/L016834/1) funded by The Engineering and Physical Sciences Research Council (EPSRC) (UK).

[10] Python Software Foundation. 2018. Python 3.7 Language Documentation. https://docs.python.org/3/c-api/index.html Online, accessed 14 August 2018.

[11] Clemens Grelck. 2012. Single Assignment C (SaC): High Productivity Meets High Performance. In *4th Central European Functional Programming Summer School (CEFP'11), Budapest, Hungary (Lecture Notes in Computer Science)*, V. Zsók, Z. Horváth, and R. Plasmeijer (Eds.), Vol. 7241. Springer, 207–278. https://doi.org/10.1007/978-3-642-32096-5_5

[12] Clemens Grelck and Kai Trojahner. 2004. Implicit Memory Management for SaC. In *Implementation and Application of Functional Languages, 16th International Workshop, IFL'04*, Clemens Grelck and Frank Huch (Eds.). University of Kiel, Institute of Computer Science and Applied Mathematics, 335–348. Technical Report 0408.

[13] Jing Guo, Robert Bernecky, Jeyarajan Thiyagalingam, and Sven-Bodo Scholz. 2014. Polyhedral Methods for Improving Parallel Update-in-Place. In *Proceedings of the 4th International Workshop on Polyhedral Compilation Techniques*, Sanjay Rajopadhye and Sven Verdoolaege (Eds.). Vienna, Austria.

[14] Jing Guo, Jeyarajan Thiyagalingam, and Sven-Bodo Scholz. 2011. Breaking the Gpu Programming Barrier with the Auto-parallelising Sac Compiler. In *6th Workshop on Declarative Aspects of Multicore Programming (DAMP'11), Austin, USA*. ACM Press, 15–24. https://doi.org/10.1145/1926354.1926359

[15] Jurriaan Hage and Stefan Holdermans. 2008. Heap recycling for lazy languages. In *Proceedings of the 2008 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation, PEPM 2008, San Francisco, California, USA, January 7-8, 2008*. 189–197. https://doi.org/10.1145/1328408.1328436

[16] G. W. Hamilton and S. B. Jones. 1991. Compile-Time Garbage Collection by Necessity Analysis. In *Functional Programming, Glasgow 1990*, Simon L. Peyton Jones, Graham Hutton, and Carsten Kehler Holst (Eds.). Springer London, London, 66–70.

[17] Paul Hudak and Adrienne Bloss. 1985. The Aggregate Update Problem in Functional Programming Systems. In *Proceedings of the 12th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL '85)*. ACM, New York, NY, USA, 300–314. https://doi.org/10.1145/318593.318660

[18] Apple Inc. 2018. Swift Language Documentation. https://docs.swift.org/swift-book/ Online, accessed 14 August 2018.

[19] Thomas B. Jablin, Prakash Prabhu, James A. Jablin, Nick P. Johnson, Stephen R. Beard, and David I. August. 2011. Automatic CPU-GPU Communication Management and Optimization. *SIGPLAN Not.* 46, 6 (June 2011), 142–151. https://doi.org/10.1145/1993316.1993516

[20] Andreas Kågedal and Saumya Debray. 1997. A Practical Approach to Structure Reuse of Arrays in Single Assignment Languages. In *Proceedings of the 14th International Conference on Logic Programming*. MIT Press, 18–32.

[21] Akash Lal and G. Ramalingam. 2010. Reference Count Analysis with Shallow Aliasing. *Inform. Process. Lett.* 111, 2 (Dec. 2010), 57–63. https://doi.org/10.1016/j.ipl.2010.08.003

[22] Oukseh Lee, Hongseok Yang, and Kwangkeun Yi. 2003. Inserting Safe Memory Reuse Commands into ML-Like Programs. In *Static Analysis, 10th International Symposium, SAS 2003, San Diego, CA, USA, June 11-13, 2003, Proceedings*. 171–188.

[23] Frank H. McMahon. 1986. *The Livermore Fortran Kernels: A computer test of the numerical performance range*. Technical Report UCRL-53745. Lawrence Livermore National Lab., CA, USA.

[24] NVIDIA Corporation 2018. *CUDA C Programming Guide* (9.0.176 ed.). NVIDIA Corporation. https://docs.nvidia.com/cuda/archive/9.0/ Online, accessed 12 Aug. 2018.

[25] Young Park and Benjamin Goldberg. 1995. Static analysis for optimizing reference counting. *Inform. Process. Lett.* 55, 4 (1995), 229 – 234. https://doi.org/10.1016/0020-0190(95)00096-U

[26] SaC Development Team 2016. *SaC EBNF Grammar*. SaC Development Team. Available online at http://www.sac-home.org/doku.php?id=docs:syntax.

[27] Kazuki Sakamoto and Tomohiko Furumoto. 2012. *Life Before Automatic Reference Counting*. Apress, Berkeley, CA, 1–29. https://doi.org/10.1007/978-1-4302-4117-1_1

[28] Sven-Bodo Scholz. 1997. An Overview of Sc Sac – a Functional Language for Numerical Applications. In *Programming Languages and Fundamentals of Programming, Technical Report 9717*, R. Berghammer and F. Simon (Eds.). Institut für Informatik und Praktische Mathematik, Universität Kiel.

[29] Sven-Bodo Scholz. 2003. Single Assignment C: efficient support for high-level array operations in a functional setting. *Journal of Functional Programming* 13, 6 (2003), 1005–1059. https://doi.org/10.1017/S0956796802004458

[30] Rifat Shahriyar, Stephen M. Blackburn, and Daniel Frampton. 2012. Down for the Count? Getting Reference Counting Back in the Ring. *SIGPLAN Not.* 47, 11 (June 2012), 73–84. https://doi.org/10.1145/2426642.2259008

[31] H. Sundell. 2005. Wait-free reference counting and memory management. In *19th IEEE International Parallel and Distributed Processing Symposium*. 10 pp.–. https://doi.org/10.1109/IPDPS.2005.451

[32] Hans-Nikolai Vießmann, Sven-Bodo Scholz, Artjoms Šinkarovs, Brian Bainbridge, Brian Hamilton, and Simon Flower. 2015. Making Fortran Legacy Code More Functional. *27th Symposium on Implementation and Application of Functional Languages (IFL '15)* (2015). https://doi.org/10.1145/2897336.2897348