Custom Multi-Cache Architectures for Heap Manipulating Programs

Felix Winterstein* Kermin Fleming† Hsin-Jung Yang‡

Imperial College London
Member, IEEE

Intel Corporation
Member, IEEE

Massachusetts Institute of Technology

George Constantinides§

Imperial College London
Senior Member, IEEE

September 5, 2016

Abstract

Memory-intensive implementations often require access to an external, off-chip memory which can substantially slow down an FPGA accelerator due to memory bandwidth limitations. Buffering frequently reused data on chip is a common approach to address this problem and the optimization of the cache architecture introduces yet another complex design space. This paper presents a high-level synthesis (HLS) design aid that automatically generates parallel multi-cache systems which are tailored to the specific requirements of the application. Our program analysis identifies non-overlapping memory regions, supported by private caches, and regions which are shared by parallel units after parallelization, which are supported by coherent caches and synchronization primitives. It also decides whether the parallelization is legal with respect to data dependencies. The novelty of this work is the focus on programs using dynamically allocated, pointer-based data structures which, while common in software engineering, remain difficult to analyze and are beyond the scope of the overwhelming majority of HLS techniques to date. Secondly, we devise a high-level cache performance estimation to find a heterogeneous configuration of cache sizes that maximizes the performance of the multi-cache system subject to an on-chip memory resource constraint. We demonstrate our technique with three case studies of applications using dynamic data structures and use Xilinx Vivado HLS as an exemplary HLS tool. We show up to $15\times$ *speed-up after parallelization of the HLS implementations and the insertion of the application-specific distributed hybrid multi-cache architecture.*

I. Introduction

Design flows for application-specific integrated circuits (ASICs) and field programmable gate arrays (FPGAs) traditionally rely on a register transfer level (RTL) description of the application given in a hardware description language (HDL). High-level synthesis (HLS) raises the abstraction level from RTL to high-level languages, such as C/C++, and promises significant shortening of the design cycle compared with an RTL-based design entry. Although modern HLS tools can achieve a quality of results (QoR, measured in terms of latency and resource utilization) comparable to hand-written HDL code [1, 2], substantial source code refactoring is often required to obtain good HLS results [2]. This work automates some of this code refactoring. In particular, we focus on the extraction of parallelism, which is crucial for achieving good QoR. To ensure that the memory system is not a sequential bottleneck to performance, our technique partitions and distributes the application data over multiple on-chip memory blocks so as to benefit from the enormous memory bandwidth of FPGAs. This work leverages the customizability of FPGA implementations to automatically construct a memory system which is tailored to the requirements of a specific application.

Our departure point from related tool flows is the focus on heap-manipulating C/C++ programs, making the work a complement to existing work based on run-time profiling [3], manual code annotations [4] or automated analyses targeting static arrays referenced in static loop nests [5–7]. Pointer-based memory references and dynamic memory allocation have widespread use in software engineering, their analysis and automated code optimizations, however, are beyond the scope of the most HLS techniques to date. The memory model in

*Felix Winterstein is with the Department of Electrical and Electronic Engineering, Imperial College London, SW7 2BT, United Kingdom (f.winterstein12@imperial.ac.uk).

†Kermin Fleming is with the Software Services Group, Intel Corporation, 77 Reed Road, Hudson, MA 01749 USA (kermin.fleming@intel.com).

‡Hsin-Jung Yang is with the department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, 32 Vassar Street, Cambridge, MA 02139, USA (hjyang@mit.edu).

§George Constantinides is with the Department of Electrical and Electronic Engineering, Imperial College London, SW7 2BT, United Kingdom (g.constantinides@imperial.ac.uk).

C/C++ assumes the presence of a *heap*, a large monolithic memory space and the identification of independent and shared portions in heap remains complicated because of the difficulty of disambiguating aliases and predicting the referenced memory locations.

The heap analysis in this work is based on *separation logic* [8], a theory for formal reasoning about programs. Leveraging separation logic in an HLS context is a recent approach explored within this line of work. Our heap analysis builds on a previously developed baseline analysis [9], which determines whether (parts of) the accessed heap memory can be completely partitioned into disjoint, non-overlapping portions, referred to as *heaplets*. Finding a solution to this question is a pre-requisite for parallelization and partitioning the on-chip memory space. The analysis guides automated code transformations which ensure the synthesizability of heap-manipulating C++ programs by off-the-shelf HLS tools and implement the partitioning and parallelization.

The applicability of the baseline technique in [9] is limited to cases where the on-chip memory capacity is sufficient and the accessed memory space can be split into independent, private partitions. However, parallel on-chip memory capacity remains a scarce resource and many FPGA applications with a large memory footprint require access to a larger off-chip memory. The bandwidth limitations of external memory can significantly slow down an FPGA accelerator and potentially eliminate the gain of parallelization. The work presented in this paper seeks to bridge the gap between a large accessible memory space and fast on-chip memory by inserting parallel on-chip caches to buffer frequently reused data on the chip and to reduce the number of expensive accesses to an external memory. This paper builds on work previously published in [10], which applies the baseline analysis in [9] to the synthesis of such an application-specific high-performance memory hierarchy and extends it to shared resources. Application specificity is introduced by distinguishing between independent *private caches* and *coherent caches* which are backed by a mechanism to maintain inter-cache coherency. We leverage a static program analysis to determine whether or not and for which caches such a coherency mechanism is required in the generated memory system. The contributions in [10] are:

- In addition to the identification of disjoint heap regions, we extend the baseline program analysis in [9] by an identification of heaplets that would be shared by the parallel loop kernels after parallelization by the source-to-source translator. Our analysis inserts additional synchronization primitives for program parts that access shared resources (Section IV).

- Even if coherency is ensured, updates to the shared resource may happen in a different order after parallelization compared to the sequential program. We implement a *commutativity analysis* for the shared heap update to prove that the parallelization is semantics-preserving (Section iii).

- We target FPGA accelerators with access to an off-chip memory. The disjointness and sharing information provided by our analyses are used to break the heap (residing in off-chip memory by default) into heaplets, to generate a custom parallel multi-cache architecture and (if needed) coherency mechanisms: we synthesize parallel private caches for disjoint heap regions and (inherently more expensive) coherent parallel caches for shared regions (Section V).

- We demonstrate the effectiveness of our technique using three applications as test cases which dynamically allocate memory and traverse and update heap-allocated data structures. We use Xilinx Vivado HLS as an exemplary back-end HLS tool and implement for a Virtex 7 FPGA (Section VII).

This paper is an extended version of the work published in [10]. In contrast to [10], the program analysis and code transformations operate on the LLVM intermediate representation [11], which allows us to combine the framework with an LLVM-based technique [12] that optimizes the capacity of the multi-cache system produced by our automated flow. In applications with large memory footprints, the bulk of the data necessarily resides off-chip. The HLS core then often keeps only small data structures on-chip. Consequently, the amount of on-chip block random access memory (BRAM) used by the core is often smaller than the amount of the BRAM available. We leverage our previous work in [12] which provides a technique for repurposing left-over on-chip memory to scale up the size of individual caches in the multi-cache system. This paper integrates this technique in the above cache construction flow to automatically generate and size the multi-cache system for heap-manipulation code as shown in Fig. 1.

- We extend the cache compilation framework by a dynamic (input data dependent) program analysis to implement an automated size scaling of private caches using spare on-chip memory resources. The original LLVM code is instrumented with profiling instructions for gathering the memory reference trace of the program under test. We include a cache hit rate estimator based on this trace (Section ii).

- The cache size assignment is cast into a Knapsack Problem to find the best distribution for the memory access pattern of a particular application (Section iii).

- We evaluate the combined tool flow with physical implementations on a Virtex 7 FPGA connected to a DDR3 memory, showing up to 3× improvement of the execution time compared to the cache system in [10].

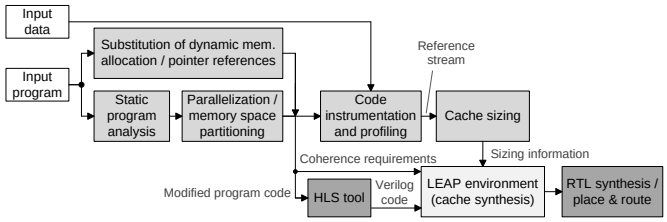

Figure 1: *Summary of the extended tool flow in this paper.*

We also characterize the impact of the cache insertion by the combined flow in terms of power and energy consumption (Section VII)

The remainder of this paper starts with an introduction of LEAP, the open-source framework this work leverages, and a discussion of related work.

II. Related Work

Automatic cache design in an HLS context requires the extraction of application-specific properties from program descriptions and remains outside the scope of most HLS flows. This work provides a static memory access analysis and automated code generation. The latter embeds the C/C++-based HLS kernels in the LEAP (Latency-insensitive Environment for Application Programming) framework [13], an environment that provides access to a physical FPGA device and memory. Like an operating system, LEAP provides a unified layer of abstraction on top of device-specific drivers that interface to the underlying FPGA device, on-board memory and the host system. We build a memory hierarchy including the multi-cache system using *LEAP Memories* [14, 15]. LEAP Memories are abstractions, which build an interface to off-chip memory underneath a uniform interface. Each single memory contains an optional direct-mapped on-chip cache [14] which holds a copy of data residing in external memory. LEAP memories without on-chip caches forward all requests to off-chip memory which results in longer response times. The same applies for cache misses. Evicted items are automatically flushed to the next memory level. A multi-cache system is built by instantiating multiple memories with on-chip caches. LEAP memories also provide an optional inter-cache coherency mechanism [15]. We use both types, private (for disjoint memory regions) and coherent memories (for shared memory regions). The latter appear as independent interfaces to the application, while they are internally connected via an additional ring network. Coherent memories are more expensive (in terms of FPGA resources) and slower (in terms of response time) than their private counterparts. Our automated technique thus seeks to avoid using costly coherent memories whenever possible, based on the specific requirements of the application.

Recent work has also explored the design space of the cache micro-architecture [15–19]. Matthews *et al.* [17] explore the efficiency in terms of speed-up versus area increase of parallel coherent L1 caches with respect to size, associativity and replacement rule in an FPGA-based soft multi-core processor. Similarly, Choi *et al.* [18] compare different configurations of cache size, line size and associativity of shared on-chip caches, in addition to two approaches for increasing the number of access ports of the shared cache. FCache [16] and LEAP Coherent Memories [15] target the micro-architecture of coherency mechanisms for shared memory systems in FPGAs. The goal in this work is different. We determine coherence and synchronization requirements for a particular application, infer cost/performance estimates prior to implementation and devise an automated cache system construction for a given application instead of exploring the cache micro-architecture.

There are several approaches to cache construction in an HLS context. Cheng *et al.* [3] use run-time profiling to group memory accesses the same locations into partitions and to instantiate separate on-chip caches assigned to disjoint memory regions. This approach does not make any assumptions about the type of program to analyze, but a disadvantage is the need for user-provided input data to settle the disjointness question and that it requires a mechanism to take corner cases into account that have been missed during simulation. We determine the need for coherency, a parameter that affects functional correctness, in a compile time analysis. Similar to our work, the CHiMPS framework is a C-to-FPGA flow that generates a distributed multi-cache architecture [4]. A main difference to our work is that independent memory regions must be manually indicated with source code annotations as opposed to an automated analysis. Secondly, shared memory regions are not supported by caches, while we automatically insert a coherency network when it is required. CHiMPS' many-cache system is notable in that it also constructs parallel caches based on left-over BRAM, clock rate degradation and predicted miss rate. The third key difference of our work is the non-uniform sizing, which is realized by solving an optimization problem to find the best assignment of cache sizes subject to a resource constraint.

Significant advancements in automated static analyses have been made for loop analyses using the *polyhedral model*, an algebraic representation of the iteration space of static loop nests. The model can precisely analyze the accesses to static arrays referenced in such loop nests. For example, Liu *et al.* [5] and related work in [6] use the polyhedral model to determine the addresses of reused data items and buffer them in on-chip memories, whereas Pouchet *et al.* [7] present an on-chip buffer insertion in combination with automatic loop parallelization. The

```
1  //main kernel function
2  void filter(TR *root, CS cinit, CI *z) {
3    CS* c0 = new CS;
4    *c0 = cinit;
5    ST *s = push(root, c0, true, NULL);
6    while (s != NULL) {
7      TR *u; CS *c; bool d;
8      s = pop(&u, &c, &d, s);
9      CS cs = *c;
10     if (d) {
11       delete c;
12     }
13     CS *cnew = new CS;
14     *cnew = subfunction1(cs);
15     if  (u->left!=NULL) && (u->right!=NULL) &&
           (subfunction2(cs))) {
16       s = push(u->left, cnew, true, s);
17       s = push(u->right, cnew, false, s);
18     } else {
19       delete cnew;
20       // update centroid information
21       CI w = u->wgtCent;
22       CI wprev = z->wgtCent;
23       z->wgtCent = wprev+ w;
24     }
25     delete u;
26   }
27 }
28 //auxiliary function push (create new entry)
29 inline ST* push(TR *u, CS *c, bool d, ST *s){
30   ST *t = new ST;
31   t->u=u; t->c=c; t->d=d; t->n=s;
32   return t;
33 }
34 //auxiliary function pop (delete list head)
35 inline ST* pop(TR **u, CS **c, bool *d, ST *s){
36   *u=s->u; *c=s->c; *d=s->d; ST *t=s->n;
37   delete s; return t;
38 }
```

Listing 1: *Pseudo code from a K-means clustering kernel [2].*

polyhedral model provides a powerful abstraction for the analysis of static loop kernels and array references, but it cannot analyze arbitrary memory accesses such as indirect array references or pointer accesses or capture dynamic memory allocation. Our focus on heap-allocated data-structures significantly increases the body of code for which automated parallelization and automatic memory-system optimizations can be applied.

III. Motivating Example

This section reviews a motivating example and explains how the extensions of the baseline analysis in [9] are applied to generate a multi-cache architecture for both private and shared heap regions. Our example is taken from a high-performance implementation of the *filtering algorithm* [20] for *K*-means clustering. Listing 1 shows the tree-based *K*-means clustering kernel code [2]. The `while`-loop in `filter` accesses four heap-allocated data structures: the binary tree (type `TR`), the sets of candidate centers (type `CS`), the stack (type `ST`) and the centroid information (type `CI`). The tree has been built up from the data set to be clustered. The center sets are intermediate

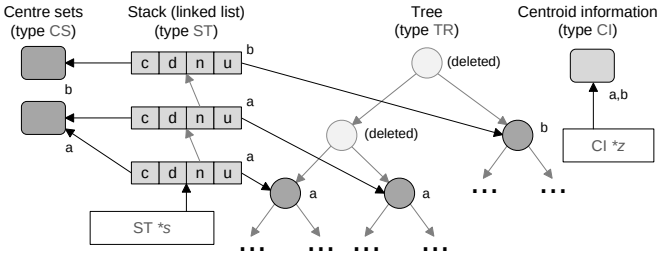

Figure 2: *Snapshot of the pointer-linked dynamic data structures accessed by the loop in Listing 1.*

solutions propagated through the call graph. The stack data structure is a pointer-linked list which manages the tree traversal. It stores the pointers to left and right subtrees and the center sets. The auxiliary functions `push` (Lines 5, 16 and 17) and `pop` (Line 8) retrieve these pointers from it and perfom `new`/`delete` operations on its head. If the data-dependent conditional (Line 15) evaluates to false (dead end of the tree traversal) the centroid data structure is updated (Lines 22 and 23) which contains the information from which the final clustering result is calculated. As we shall see below, this code fragment results in a shared resource after parallelization of the application.

All data structures accessed by this program are created at run-time using dynamic memory allocation. Dynamic allocation (and disposal) results in efficient memory usage if the average-case amount of required memory is much smaller than the worst-case amount. An efficient memory architecture for this program provides fast access to this small amount of memory space and, at the same time, supports worst-case allocation by providing a large memory as a backup. Hence, our approach is to place, by default, all heap-allocated data in a large off-chip memory connected to the FPGA accelerator and to insert on-chip caches which mirror parts of the off-chip data and provide fast data access. We describe the extensions of our baseline analysis below.

i. Memory Partitioning and Parallelization

Fig. 2 shows an example of the data structures allocated in the heap after executing two `while`-loop iterations of Listing 1. The data structures are grouped according to their types. The loop can be split into parallel sub-loops as shown in Listing 2 (two in this example). If we ignore the centroid data structure (type `CI`) in Fig. 2 for a moment, the baseline method in [9] can prove that the pointers dereferenced in any iteration of a sub-loop never refer to the data structures used by the other loop. Hence, we call these loop kernels 'communication free' with respect to each other, *i.e.* these program parts can run in parallel because they access different memory locations. The analysis partitions the remaining tree data structure (dark gray nodes, type `TR`) into two groups of sub-trees labeled with *a* and *b*. It splits the linked list (type `ST`)

into the uppermost node and the nodes beneath, and the pool of center sets (type CS) is partitioned accordingly. Each parallel sub-loop obtains its own interfaces to off-chip memory and the fact that the memory regions can be proven to be disjoint allows our tool to instantiate fast and cheap private memories for each partition without the need to ensure costly coherency between them.

ii. Parallel Access to Shared Resources

Our baseline analysis in [9] aborts if it cannot fully split the heap-allocated data into the desired number of partitions. For example, the shared centroid information in Fig. 2 cannot be partitioned since it is accesed by both sub-loops. Our extended analysis marks it as a shared resource, indicated by the label $\{a, b\}$, as both sub-loops would update it after parallelization. After the detection of a shared heap region, our framework instantiates a coherent memory interface to this region in each sub-loop consisting of two parts: caches with a coherence mechanism and locks which enable atomic updates of the shared resource in the presence of multiple accessors. The detection of a shared resource triggers a second analysis as sharing invalidates the independence assumption that parallel units access different data. Assuming that coherency is ensured between parallel units, it remains to prove that the modified order in which the shared resource is updated after parallelization does not alter the program semantics. The centroid information is updated in Line 23 of Listing 1

$$z \rightarrow \texttt{wgtCent} = w_{prev} + w;$$

where w is the contribution of the tree nodes. In the original, sequential program, z receives the contributions of all nodes in the right sub-tree (labeled with a) before it receives the first contribution from the left sub-tree (labeled with b in Fig. 2). However, in the parallelized version z may be updated with data from left and right sub-tree in an arbitrarily interleaved fashion. Even if atomicity of the update is ensured, we must also ensure that this new update order is legal. In this example, the parallelization is legal because of the commutativity and associativity of the addition[1]. We address this question with a *commutativity and associativity analysis* of the update function. Listing 2 shows the output of a source code transformation based on the result of all analyses above. The transformed code, when run through a back-end HLS tool and RTL implementation, results in a custom configuration of multiple private/coherent memories with a custom degree of parallelism. The on-chip FPGA memory blocks are aggregated accordingly to construct the custom parallel caching scheme.

[1]We focus on integer or fixed-point systems and ignore non-associativity caused by floating-point representations.

```
1  void filter(TR *root, CS cinit, CI *z) {
2    ...preamble (pointers access partitions a and
         b)
3    while (s_a != NULL) { //parallel loop kernel a
4      ..access private memory for CS, partition a
5      ..access private memory for ST, partition a
6      ..access private memory for TR, partition a
7      acquireLock();
8      ..access coherent memory for CI, partition a
9      releaseLock();
10   }
11   while (s_b != NULL) { //parallel loop kernel b
12     ..access private memory for CS, partition b
13     ..access private memory for ST, partition b
14     ..access private memory for TR, partition b
15     acquireLock();
16     ..access coherent memory for CI, partition b
17     releaseLock();
18   }
19 }
```

Listing 2: *Transformed program from Listing 1.*

iii. Custom Cache Sizing

The above identification of disjoint and shared heap regions and the legality of parallelization provides information about the cache types, but no information about their size. Hence, all caches inserted by the tool flow above have the same size by default. Since we synthesize a cache for each data structure partition, the access patterns to these memory regions may be very different: For example, the stack data structure in Listing 1 is usually small compared to the tree structure and has high access locality at the head of the stack. In this case, using the available on-chip memory to build a small cache for the stack and a large cache for the tree is more beneficial than both caches having the same size. In addition to the static analysis above, we extend our tool flow by a profiling-based analysis in Section VI which enables custom sizing to maximize the aggregate hit rate of the multi-cache system. We refer to the profiling-based analysis as a dynamic analysis.

IV. Extended Static Program Analysis

This section describes the program analyses enabling the code transformations that turn a sequential heap-manipulating program into a parallelized HLS implementation with an application-specific off-chip memory interface. The following background section briefly reviews theoretical background of separation logic [8] and the heap analysis developed previously in [9], which we refer to the 'baseline analysis'. New extensions of the baseline are described after Section i.

i. Background

Our static analysis is based on the *symbolic execution* of a program under test. The values assigned to program variables and memory cells accessed are referred to as *program state*. During execution, the state is modified

by program statements (commands). Additionally, a program contains control parts such as conditionals and loops. The symbolic execution engine in our analysis propagates the program state through all paths of the control flow graph (CFG). Branching creates multiple control flow paths and loops create a cycle.

The analysis uses a formal description of the program state. It describes the values assigned to program variables (*e.g.* $x = 3 \wedge y = 5$ means that variable x and y currently hold the value 3 and 5, respectively, where '$\wedge$' is the classical 'and'-conjunction). At each CFG node, the symbolic execution engine updates the current description of the program state, *e.g.* the assignment statement $x := 1$ results in the new state $x = 1 \wedge y = 5$. In addition to this, the state model consists of the *heap* which describes the values assigned to addressable memory locations (*e.g.* $v \mapsto 4$ means that the memory cell referenced by the pointer variable v contains the value 4). Reasoning about the program semantics in this way is substantially more complicated if a program uses heap-directed pointers. Assuming that the current program state is $u \mapsto 4 \wedge v \mapsto 6$ and we execute the heap update command $[u] := 1$, we may wish to conclude that the assignment does not affect the heap cell referenced by v, *i.e.* $u \mapsto 1 \wedge v \mapsto 6$. This, of course, can only be ensured if we explicitly rule out the potential aliasing of u and v by adding an additional constraint: $u \neq v \wedge u \mapsto 1 \wedge v \mapsto 6$. These constraints are required for each pair of pointers in the program, quickly limiting the applicability of an automated heap analysis to real-life programs arising in practice.

Separation logic solves this problem by extending the classical first order logic by a 'separating conjunction' ($*$): The formula $u \mapsto 4 * v \mapsto 6$ means that the heap is split into two disjoint portions h_0 and h_1, where $u \mapsto 4$ holds for h_0 and $v \mapsto 6$ holds for h_1. We call disjoint heap portions *heaplets*. The $*$-operator rules out aliasing of pointers u and v by definition, *i.e.* it implies $u \neq v$. Hence, each heap cell can be updated without any side effects for the other one. In addition to single values, $u \mapsto [\mathtt{f_1} : x'_1, .., \mathtt{f_n} : x'_n]$ describes a heap-allocated record (*structs* in C/C++): u points to a record containing the fields $\mathtt{f_1}, ..., \mathtt{f_n}$ with content $x'_1, ..., x'_n$. In addition to program variables u, v, x and y, the primed variables $x'_1, ..., x'_n$ are symbolic replacements of values and are only used in formulae (not as program variables). The abbreviation $u \mapsto \_$ means that u points to 'some' record. In general, formulae in separation logic are of the form $\Pi \wedge \Sigma$, where Π are assertions in classical logic (connected by '$\wedge$') and Σ are spatial assertions such as $u \mapsto 4 * v \mapsto 6$. Σ can also include the value *emp* which denotes an empty heap where nothing is allocated. Pointer variables may hold a special value `nil` corresponding to the `NULL` expression in C/C++. The effect of heap-manipulating program commands (`new`, `delete` and dereferencing for read/write) can be specified with concise separation logic formulae and used by the symbolic execution engine. Our analyses use a 'heap footprint' analysis [21] to find disjoint and shared heap regions in the program.

Our baseline analysis for identifying private heap regions and memory partitioning [9] is the starting point for all subsequent analyses related to parallelization, shared resources and commutativity of shared resource updates. Loop parallelization and its follow-up analyses are only triggered if (at least parts of) the heap-allocated data structures accessed by the loop can be split into P partitions, where P is the desired parallelism degree. The inner `do-while`-loop in Algorithm 1 summarizes the baseline analysis in [9]. The analysis starts with the separation logic formula describing the pre-state of the loop ($\Pi \wedge \Sigma$). The `while`-loop in Line 24 symbolically executes (function SymbExeLoopBody) the loop body. We refer to this process as 'peeling off' (unrolling) loop iterations. For our motivating example above, two loop iterations of the `while`-loop in Listing 1 are peeled off and a new formula describing the new state are obtained. In each peeling step, the analysis adds additional pointer variables to the state formula. We name these variables *cut-points* [9] and the insertion function CutpInsert. Cut-points split the state formula in partitions. After two peeling steps for this example, the analysis has found a valid set of cut-points C, that is 1) C consists of $P = 2$ cut-points, 2) these cut-points reference heaplets of the same shape, *i.e.* describe the same type of data structure. These two cut-points are the pointer s in Fig. 2 and a second pointer s_b referencing the uppermost stack record in Fig. 2. The built-in *fix-point calculation* (function FixpCalc) [9] now aims to prove that the initial partitioning of the heap-allocated data structures is maintained for all subsequent loop iterations of the `while`-loop in Listing 1, *i.e.* any loop iteration accesses either the partition referenced by s or that referenced by s_b, but never both. If we ignore the centroid information in Fig. 2 for the moment this proof is successful; the loop iterations can hence be split into $P = 2$ groups with accesses to disjoint regions of memory, a prerequisite for parallelization.

ii. Detecting Private and Shared Resources

The baseline analysis in [9] is limited to cases where the accessed memory space can be split into independent, private partitions. In that work, we aborted the analysis reporting a failed proof after a fixed parameter of L unrollings if the program state cannot be completely partitioned (Line 32 in Algorithm 1). Here, we relax the constraint that the inherent parallelism of the application needs to be communication-free. The outermost `do-while`-loop in Algorithm 1 shows the extended analysis to identify disjoint and shared resources. If we now include

Algorithm 1 Detecting private and shared resources.

```
 1: Input:
 2: Loop body specification (code)
 3: Initial state formula (Π ∧ Σ)_initial
 4: Output:
 5: Assignment of pointer statements to heap partitions
 6: Number of initial unrollings
 7: Shared/private predicate for pointer statements
 8: Variables:
 9: it: Iteration counter (number of iterations to be unrolled)
10: C: Set of cut-points
11: C_shared: Set of cut-points referencing shared heaplets
12: S_cutpoints: Set of cut-point states
13: Π ∧ Σ: Loop pre-state formula
14: StmtS: Set of statement sets accessing shared heaplets
15: function HeapAnalysis
16:     C_shared ← ∅
17:     do
18:         it ← 0
19:         C ← ∅
20:         Π ∧ Σ ← (Π ∧ Σ)_initial
21:         StmtsS ← ∅
22:         success ← false
23:         do
24:             while C not valid do
25:                 Π ∧ Σ ← SymbExeLoopBody(Π ∧ Σ, it)
26:                 Π ∧ Σ, C ← CutpInsert(Π ∧ Σ, C_shared)
27:                 it ← it + 1
28:             end while
29:             S_cutpoints ← AssignCPStates(C)
30:             success, Stmts_shared ← FixpCalc(
                    Π ∧ Σ, C, S_cutpoints, C_shared)
31:             StmtsS ← StmtsS ∪ {Stmts_shared}
32:         while (not success) and (it < L)
33:         if it = L then
34:             C_shared ← ExtrctCutp( argmin_{Stmts∈StmtsS} |Stmts|)
35:         end if
36:     while not success
37:     ... generate analysis output
38: end function
```

the centroid information of our motivating example and run the disjointness analysis, the fix-point calculation cannot maintain the partitioning as it always finds a non-singleton label set attached to it and never reports a valid proof. Our goal is to mark this heaplet as a shared resource. The shared resource analysis requires two extensions of the baseline analysis: 1) identifying shared heaplets and 2), once marked as shared, re-running the cut-point insertion and proof-engine invocations while excluding them from the search for separable heap regions.

In the first phase, we turn a failed proof of complete separability into the detection of shared resources. Returning to our running example, we run the cut-point insertion and fix-point calculation with the objective of splitting the heap into $P = 2$ partitions. After peeling off the first two loop iteration, the function FixpCalc terminates unsuccessfully because it finds that both groups of loop iterations (as explained above) access the centroid information in Fig. 2. Even if more iterations are peeled off, the shared access to this heaplet remains. We use a heuristic to filter such shared resources by declaring all heaplets, which maintain this sharing property after L unrollings, as shared (Line 32). The fix-point calculation is modified in that whenever it detects sharing on a heaplet, it collects the set of program statements that accessed the shared heaplet (each statement in the control flow graph has a unique identifier). During the course of the alternating iteration unrolling, cut-point insertion and fix-point calculation, the analysis builds a set of statement sets accessing shared heaplets ($StmtsS$).

After termination of the inner `do-while`-loop, the analysis is reset. From $StmtsS$, we pick the set $Stmts$ containing the fewest statements accessing shared resources, from which the function ExtrctCutp extracts all cut-points mentioned in at least one of these program statements (C_{shared}). The second phase begins by relaunching the analysis. We pass the set C_{shared} to the modified function CutpInsert which excludes these cut-points during the search for cut-points in the loop pre-state. Similarly during the fix-point calculation we prevent the analysis from complaining about the shared access if the current program statement has been marked as excluded. Finally, we obtain a proof of separability for the tree, the stack and the pool of center sets, and the centroid heaplet is marked as a shared resource. The interface to the shared region in off-chip memory is supported by a coherency protocol. The corresponding program statements accessing the shared resource are Lines 22 and 23. The knowledge of shared resource access made by these statements, discovered by the analysis, is used by the source code transformation to insert `acquireLock` and `releaseLock` commands before and after the critical statements as shown in Listing 2 in order to ensure atomic updates of the shared heap region.

iii. Commutativity Analysis

Parallelization in the presence of shared resources requires a second analysis step after detection of a shared heap region. We must verify that, after parallelization, the program semantics are not altered as a result of the order in which the updates of the shared resource are made by the parallel version being altered. For example, during the execution of the original (unparallelized) loop in Listing 1, the shared centroid information receives all contributions from the right sub-tree before it receives any contribution from the left sub-tree, while it may be updated with data from left and right sub-tree in an arbitrarily interleaved fashion in the parallelized version. Enforcing the original order with barrier synchronization means re-sequentializing the parallelized implementation and is not a viable solution. Instead we want to determine that the modified order of state updates is legal. In the following walk-through, for ease of explanation, we define the function F which reads and writes the shared state (Lines 22 and 23 in Listing 1):

Definition 1 (Update function)**.**

function $F(w)$
$w_{prev} = z \rightarrow \texttt{wgtCent};$
$z \rightarrow \texttt{wgtCent} = w_{prev} + w;$
end function

A commutativity analysis was proposed by Rinard and Diniz [22] and our approach builds on the same basic idea: We say two operations on the program state are commutable if their execution in sequence results in the same program state regardless of their execution order. In our case, F is commutable if $\forall w_1, w_2$, $F(w_1); F(w_2)$ results in the same program state as $F(w_2); F(w_1)$. From the shared resource detection above, we extract the pre- and post-conditions on the program state before and after F's execution:

$$\begin{aligned} &\{w = w'_0 \wedge z \mapsto [\texttt{wgtCent} : w'_1]\} \\ &F \\ &\{w = w'_0 \wedge w'_2 = w'_1 + w'_0 \wedge z \mapsto [\texttt{wgtCent} : w'_2]\} \end{aligned} \tag{1}$$

The extraction phase brings the pre- and post-specification of F into a canonical form $\Pi \wedge \Sigma$, where Π are the pure formulae and Σ are the spatial formulae referring to the shared heap resource. For example, the built-in symbolic execution engine ensures that arithmetic operations in the state formulae appear only in the pure part as demonstrated for the expression $w'_1 + w'_0$ in the bottom row of (1). We test whether F is commutable by symbolically executing two sequences of two calls to F:

$$w = w'_{0,1};\ F(w);\ w = w'_{0,2};\ F(w);\ w = w'_{0,3}; \tag{2}$$
$$w = w'_{0,2};\ F(w);\ w = w'_{0,1};\ F(w);\ w = w'_{0,3}; \tag{3}$$

Note the permuted assignment of symbolic values to w in (3). In order to show that F is commutable, we must prove that the post-states of the sequences in (2) and (3) describe the same program state. Their post-state formulae are:

$$\{w = w'_{0,3} \wedge w'_3 = w'_1 + w'_{0,1} + w'_{0,2} \wedge z \mapsto [\texttt{wgtCent} : w'_3]\} \tag{4}$$
$$\{w = w'_{0,3} \wedge w'_4 = w'_1 + w'_{0,2} + w'_{0,1} \wedge z \mapsto [\texttt{wgtCent} : w'_4]\} \tag{5}$$

The updated shared resource in (4) and (5) is described by $z \mapsto [\texttt{wgtCent} : w'_3]$ and $z \mapsto [\texttt{wgtCent} : w'_4]$, respectively. We want to prove that these predicates describe the same state. We first ask a separation logic theorem prover whether they match which recognizes their equality in shape and creates a new proof obligation: $w'_3 = w'_4$. Next, we combine this verification condition with the remaining pure parts of the formulae and aim to prove:

$$\begin{aligned} &\forall w'_{0,2}, w'_{0,1}. \\ &w = w'_{0,3} \wedge w'_3 = w'_1 + w'_{0,1} + w'_{0,2} \wedge \\ &w = w'_{0,3} \wedge w'_4 = w'_1 + w'_{0,2} + w'_{0,1} \Rightarrow (w'_3 = w'_4) \end{aligned} \tag{6}$$

In the actual verification step, we use satisfiability modulo theories (SMT) solving [23] to decide (6). However, an SMT solver cannot deal with the universal quantification ($\forall$), so we rephrase (6) by negating the verification condition:

$$\begin{aligned} &\exists w'_{0,2}, w'_{0,1}. \\ &w = w'_{0,3} \wedge w'_3 = w'_1 + w'_{0,1} + w'_{0,2} \wedge \\ &w = w'_{0,3} \wedge w'_4 = w'_1 + w'_{0,2} + w'_{0,1} \wedge (w'_3 \neq w'_4) \end{aligned} \tag{7}$$

The solver returns one of three possible results: 1) If (7) is satisfiable, we can find an assignment to the input variables $w'_{0,2}, w'_{0,1}$ of F that makes the two program states after executing the two sequences different: F is not commutable. 2) If (7) is not satisfiable, there is no such assignment: F is commutable. 3) The solver may not be able to decide the question in which case we conservatively assume that F is not commutable. For the running example and with the theory of linear arithmetic of integers it decides that F is commutable. Commutativity has been proven to be an undecidable problem in general [24]. However, it can still be shown for many cases that arise in practice. Next, we describe our compilation flow that uses the information provided by the above static program analyses to generate custom multi-cache architectures.

V. Code Generation

The tool flow of the multi-cache synthesis consists of three main parts: 1) The analysis extension builds on the baseline heap analyzer in [9]. It also interfaces to the Z3 SMT solver [23]. 2) The modified source-to-source translator, which implements the loop parallelization and pointer access transformations, is implemented as a custom transformation pass in the LLVM infrastructure [11]. The code generation includes directives for instructing Vivado HLS to generate bus interfaces for memory access. 3) We use LEAP [13] to embed the C/C++-based HLS kernels in an environment that constructs the on-chip/off-chip memory hierarchy (through LEAP Memories) as explained in Section II.

The source-to-source transformation replaces heap memory with arrays located in off-chip memory by default (a portion of them then resides on-chip via caches) and each heap access becomes an access to the external memory bus. Our translator turns pointer dereferencing into array-based bus accesses and instantiates a memory interface for each data structure type and each of the P heap partitions (private and shared). The extended heap analyzer provides information on whether the memory bus points to a private or a shared heap region. We insert a generic Verilog wrapper for each interface which acts as a bridge between Vivado's native bus protocol and the LEAP interface. Vivado's scheduler ensures that, when the HLS kernel issues a memory request, it stalls execution until the memory request has been serviced by the LEAP Memory.

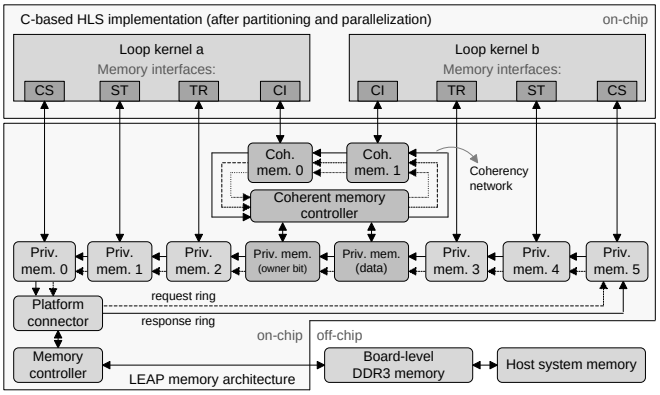

Figure 3: *Parallelized HLS implementation of the filtering algorithm with a hybrid cache architecture (private interfaces for center sets (CS), stack records (ST), tree nodes (TR), coherent interfaces for the centroid information (CI)).*

```
1 requestLock(access_critical_region0);
2 waitForLock(); //stalls until lock is acquired
3 ...issueMemoryRequest //set memory fence
4 releaseLock(access_critical_region0);
```

Listing 3: *Lock-synchronized shared memory access.*

Fig. 3 shows the integration of our running example after heap partitioning and parallelization with $P = 2$ into the LEAP memory hierarchy. Each loop kernel (we omit the preamble here) has an interface to the memory system for each type of heap-allocated data-structure: center sets (CS), stack records (ST), tree nodes (TR) and centroid information (CI). An additional coherency network is instantiated for the CI ports (shared memory). For shared heap regions, the source translator inserts synchronization signals to ensure atomic updates to the shared heap cell. Listing 3 shows an example. The function argument `access_critical_region0` translates into a Boolean signal in the generated RTL code and triggers lock acquisition and release. LEAP's lock service ensures that no access to heap region 0 is granted before the lock is acquired (only one requestor can own the lock). The memory fence instruction ensures that the memory transaction has been completed before releasing the lock.

The on-chip caches of the LEAP Private and Coherent Memories are direct-mapped and we use a write-back policy. The presence of a coherency/synchronization mechanism is the only variable parameter in our cache architecture implementation above. The next section describes how our design aid automatically determines how much on-chip memory should be used for the implementation of each individual private cache.

VI. Custom Cache Sizing

Our cache sizing technique uses up the left-over BRAM and enlarges the private on-chip caches. The size of each private cache is set individually in order to obtain a size distribution across the parallel caches that is tailored to the memory access pattern of a particular application. It is important to note that our technique does not rely on successive synthesis and place-and-route cycles, but instead estimates the cache performance for different sizes with a pre-RTL, dynamic program analysis of the input code to an HLS tool. Our approach relies on a prediction of the performance of each cache from the application's reference stream, and finds a size configuration that maximizes the aggregate performance subject to a resource constraint. We adopt here a run-time profiling approach for capturing the memory reference trace in order to ensure wide applicability. Especially in heap-manipulating programs, the absolute data structure size is often unknown at compile time. Our dynamic analysis can handle such programs at the expense of relying on a representative input data set provided by the user. Using a data set which is not representative of the typical use case may affect the performance of the cache system, but does not compromise functional correctness of the implementation.

To give a more concrete example of our technique, we consider a two-cache system consisting of private caches. Our compilation flow above generates such a system, for example, from applications which use a tree data structure and a stack to implement a depth-first tree traversal. Assuming we have only run the transformation of pointer references and cache insertion for such an application without asking for additional parallelization, the hardware implementation has a private cache for stack records (ST) and tree nodes (TR). The RTL design for the modified source code is generated with an HLS tool, for example Xilinx Vivado HLS, which also provides information of the BRAM resources consumed by the HLS core itself. In this case, the core uses 112 36k-RAM blocks which leaves 918 left-over blocks in a Virtex 7 device (xc7vx485tffg1761-2) to be used by the platform surrounding the HLS core. With a conservative 40%-margin, 550 RAM blocks (2200 kB[2]) can be repurposed as cache memories.

Our technique then estimates the performance of the caches from the memory reference trace, which is obtained from running the HLS input program with a representative input data set provided by the user. The reference stream, together with the knowledge of the cache type (direct-mapped, set-associative, fully associative) allows us to model the aggregate hit rate of the multi-cache system. For $K = 2$ private caches as in this example, there is no interaction between the caches

[2]We use 32 kbits in a Xilinx 36K-RAM block to store user data

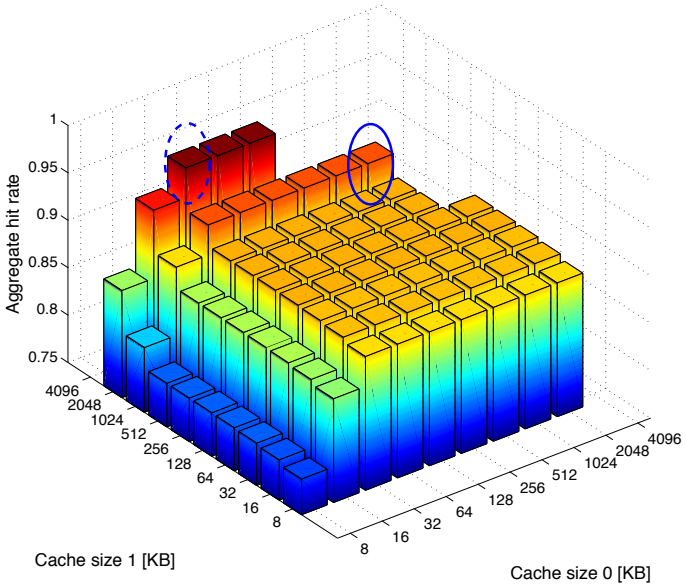

Figure 4: *Aggregate hit rate estimate for a two-cache system with an* 2200 *kB on-chip memory constraint.*

and the aggregate hit rate is given by:

$$\eta = \frac{\sum_{i=0}^{K-1} h_i(B_i)}{\sum_{i=0}^{K-1} t_i}, \tag{8}$$

where h_i is the number of hits in cache i of size B_i, t_i is the total number of accesses to cache i. Fig. 4 shows the aggregate hit rate for the two direct-mapped caches over different feasible size configurations. The design space spans hit rates from 79% to 97%. The hit rate of Cache 0 (for stack records of type `ST`) reaches its maximum at 32 kB and then plateaus. The reason for the steep improvement with low sizes and early saturation is the high locality of the memory accesses made to the stack-like linked list and the fact that just 32 kB of cache memory is sufficient to keep the entire data structure on-chip. For tree nodes (Cache 1), a 2 MB cache is needed to fit all tree data. Clearly, spending the same amount of memory resources on both caches is sub-optimal.

The advantage of our technique over a one-size-fits-all cache scaling becomes obvious when we take the memory resource constraint of 2200 kB into account. With a fixed size for all caches, on this grid, we could implement caches with a maximal capacity of 1024 kB each, which corresponds to the bar marked with the solid-line blue ellipse in Fig. 4. A cache sizing tailored to the access pattern of the application allows us to decide that a size of 32 kB for Cache 0 and 2048 kB for Cache 1 maximizes the hit rate while still satisfying the resource constraint. This design point is marked with the dashed blue ellipse in Fig. 4. In general, implementations (including those parallelized by our CAD flow) will use more than two parallel caches, and the disparity between fixed-size and application-specific cache sizing will be larger.

Replacing a fixed-size scaling with a specific size distribution relies on the ability to predict the performance of each cache from the reference stream, and to find a size assignment maximizing the aggregate performance subject to a memory constraint. Our cache sizing flow has three components: 1) It determines unused BRAM resources, which requires an estimation of the memory resources used by the HLS core itself. 2) It predicts the hit/miss counts of each cache for different sizes. 3) The amount of spare BRAM and the cache performance estimates are combined into an optimization problem which finds a variable size configuration in the multi-cache system that maximizes the aggregate hit rate.

i. On-chip Memory Utilization Estimation

We obtain high-level estimates of the BRAM consumption from the HLS tool to determine the left-over RAM resources. Here, we use Vivado HLS, which provides estimates of the number of LUTs, FFs, DSP slices and RAM blocks consumed by the HLS core. Compared to LUTs, FFs and DSP slices, the predicted amount of memory is relatively accurate. Once the tool decided which variables in the code go into BRAM, a conservative estimate can be easily made. We evaluated the accuracy of the BRAM estimation in [12] by comparing the high-level estimates with post placement and routing (PAR) results. The only cases where the high-level prediction deviates from the implementation post PAR were observed when the downstream RTL synthesis tool performed bit truncations that affected operands stored in memory. However, in these cases, the high-level estimate is always higher than the actual usage, which results in a slightly conservative but safe estimate. We also include a 10% security margin in the left-over portion used for cache construction. We use a cache banking technique in [19], implemented in the context of this work, to address a potential clock rate degradation due to large on-chip RAMs. Each cache memory is divided into smaller parallel banks with pipeline buffers at the input and output, relieving the timing pressure on cross-chip routing paths. This multi-cycle banked cache structure ensures that the critical path lies within the HLS core instead of the cache in all our benchmark designs.

ii. Cache Performance Estimation

We build our sizing technique on top of the multi-cache generator above. We instrument the transformed program with profiling instructions that fill trace buffers, which maintain the memory reference trace for each bus interface to external memory. The profiling run takes user-provided input data. Hence, we may miss corner cases with this dynamic program analysis. However, since cache size is only a performance-related parameter, the functional cor-

rectness of the optimisation is not compromised. The trace buffers are empty at program start-up. On each access to external memory in the program, the instrumentation code adds the memory address. In this way, we build up reference streams of length M_i:

$$(a_{0,i}, \ldots, a_{M_i-1,i}), \tag{9}$$

where i is the index of the memory interface. The memory is divided up into *blocks*, some of which will have copies in the cache. The block width L is equal to the cache line size. For a data width smaller than L the *block reference streams*

$$(\lfloor \tfrac{a_{0,i}}{L} \rfloor, \ldots, \lfloor \tfrac{a_{M_i-1,i}}{L} \rfloor) \tag{10}$$

give us the dynamic trace of memory accesses at the granularity of the cache line size. The line size is a fixed parameter in our analysis. If the user data width is larger, a cache access is split into multiple sequential chunks in our implementation. We model this by expanding the block reference stream (10) accordingly in a post-processing step. The cache size remains the only variable parameter in the hit rate estimation. Other parameters such as associativity are fixed but must be taken into account.

The *stack distance* metric [25, 26] counts the number of unique references 'between' accesses to the same address. A fully associative cache with LRU replacement policy produces a miss if this number exceeds the number of cache lines B. The stack distance thus provides an exact model of such a cache. The stack distance distribution of a reference stream allows us to count cold misses (cache misses due to empty cache at program start-up) and capacity misses (misses due to line eviction because the cache is full) in fully associative caches. In lower-associativity caches, additional conflict misses occur (eviction due to intervening references although the cache is not full) which the stack distance approach can only approximate [25, 26]. The prediction accuracy worsens with decreasing associativity.

Because we target direct-mapped caches and our goal is an accurate prediction, we devise a precise hit rate determination for direct-mapped caches. For each reference r and the previous reference r' to the same block address, we examine the intervening references between r' and r. A conflict miss occurs if at least one intervening reference accesses the same cache line, which is determined by reference modulo the cache size B. Algorithm 2 shows Matlab-like pseudo code of the hit rate estimator for direct-mapped caches (size B). It predicts the the number of hits (n_{hit}) and misses (n_{miss}) of the cache dependent on its size. We validate our cache model in [12] with measurements of the actual hit/miss rates. Additionally, we compared the stack distance-based approximation in [25] ($h_{\text{est}}^{\text{SD}}$, error^{SD}) with our estimator.

Algorithm 2 Hit rate of a private, direct-mapped cache.

```
1: Input:
2: Block reference stream S
3: Number of cache lines B
4: Output:
5: Miss count n_miss
6: Hit count n_hit
7: function ESTIMATE_HITRATE(S)
8:     S_u ← unique(S)                          ▷ keep unique block references
9:     n_miss, n_hit ← 0
10:    for all r ∈ S_u do
11:        I ← findAll(S = r)                   ▷ get indices of entries equal to r
12:        c ← r mod B                          ▷ cache line accessed by r
13:        n_miss ← n_miss + 1                  ▷ first access is always a cold miss
14:        for j = 1 ... length(I) − 1 do       ▷ loop over remaining accesses
15:            R' ← S(I(j − 1) + 1 : I(j) − 1)  ▷ intervening refs
16:            C' ← R' mod B                    ▷ intervening cache line refs
17:            if find(C' = c) = ∅ then
18:                n_hit ← n_hit + 1            ▷ hit
19:            else
20:                n_miss ← n_miss + 1          ▷ conflict miss
21:            end if
22:        end for
23:    end for
24:    return n_miss, n_hit
25: end function
```

Ours matches exactly the measured hit/miss counts, *i.e.* Algorithm 2 models our direct-mapped caches perfectly. The approximation by Brehob and Enbody [25] tends to underestimate the hit rate of direct-mapped caches, an observation also made in [25]. The high-level hit rate prediction allows us to compare the performance of cached memory interfaces with different block reference streams) relative to the other caches and select a configuration of cache sizes that maximizes the aggregate hit rate. The next section describes how our technique finds such a configuration.

iii. Optimization Strategy

Our compiler generates K caches as described above. With Algorithm 2, we can estimate the performance of each independent cache $h_i(B)$, $i = 0 \ldots K-1$ once we have obtained the corresponding reference streams. We assign different sizes to the caches in such a way that the aggregate hit rate is maximized. To this end, we assign to each cache a set of N cache sizes $\mathcal{B}_i = \{B_0, B_1, \ldots, B_{N-1}\}$ and compute the hit rate relative to the total number of accesses for each size. We cast the search for the best size assignment for each cache into an optimization problem and define the following variables:

$p_{ij} = h_i(B_j)$	the profit (hit rate of cache i)
$w_{ij} = bram_i(B_j)$	the cost (block RAM consumption of cache i)
C	the global constraint on the available block RAM resources
$x_{ij} \in \{0, 1\}$	a binary variable,

where $i = 0 \ldots K-1$ iterates over caches and $j = 0 \ldots N-1$ iterates of cache sizes. We phrase the maximisation problem as a *Multiple-Choice Knapsack Problem* (MCKP) [27]

as follows:

$$\begin{aligned} &\text{maximise} && \textstyle\sum_{i=0}^{K-1} \sum_{j=0}^{N-1} p_{ij} x_{ij} \\ &\text{subject to} && \textstyle\sum_{i=0}^{K-1} \sum_{j=0}^{N-1} w_{ij} x_{ij} \leq C \\ &\text{and} && \textstyle\sum_{j=0}^{N-1} x_{ij} = 1, \;\; i = 0 \ldots K-1 \end{aligned} \tag{11}$$

The objective in (11) maximizes the aggregate hit rate of K caches. The first constraint enforces memory resource limits and the second constraint ensures that, for each cache, exactly one size from the set $\mathcal{B}_i$ is selected. We solve the MCKP with an algorithm by Pisinger *et al.* [27] based on dynamic programming. The next section evaluates the HLS design aid.

VII. Experiments

We run our experiments with the three C++ applications from that traverse, update, allocate and dispose dynamic data structures in heap. All applications perform pointer-chasing and are therefore sensitive to the memory access latency.

Merger. The program builds up four linked lists from scratch performing a sorted insertion of input values, and subsequently merges and disposes the four lists to produce a single sorted output stream. The linked lists are disjoint, the parallelized program does not access shared heap memory as determined by our analysis. Four private caches are inserted in the parallelized implementation.

Reflect Tree. The application traverses a binary tree and recursively swaps the left and right child pointer of some nodes to produce a partially mirrored tree. The HLS core consists of P parallel units, each of which has two private memory interfaces and one interface to shared memory which holds a running minimum. P coherent caches and a lock service are instantiated for the shared heap region.

Filter. This is our running example. The tree, center sets and linked list data structures are partitioned and supported by private caches and the traversal loop is parallelized. The shared heap-allocated running sum is supported by coherent caches and a lock service.

We use Xilinx Vivado HLS 2014.1 as a back-end C-to-FPGA tool. We implement our benchmarks on a VC707 evaluation board (Virtex 7 FPGA, xc7vx485tffg1761-2, 1GB DDR3 SDRAM). We build the Bluespec-based LEAP framework with Bluespec 2014-07-A. The generated RTL code is integrated into the framework with Bluespec's `import BVI` statement. The complete FPGA designs are implemented in a hybrid flow with Synopsys Synplify Premier 2014.03.1 for synthesis and Xilinx Vivado 2014.4 for placement and routing. We report FPGA slices, DSP slices, 36k-BRAMs (18k-blocks count as 0.5 36k-blocks) and total latency (cycle count × clock period) for the complete FPGA designs (HLS core and multi-cache architecture). We fix the parallelization degree to $P = 4$ for all benchmarks in this evaluation. The latency results are normalized differently depending on the benchmark: latency per input sample for **Merger**, latency per full tree traversal for **Reflect Tree**, and latency per clustering iteration for **Filter**. We separate this evaluation into two parts: The first part focuses on the performance gained by inserting our multi-cache system and the benefits of specializing it by inserting coherent caches only if necessary. The second part of this evaluation section discusses the automatic scaling of private caches.

Table 1: *Parallelization and caching (cache size 1 kB).*

P: parallelisation degree; N_c: number of caches; S: speed-up over baseline							
P	N_c	LUT	FF	DSP	BRAM	Latency	S
Merger (250000 random input key-value pairs)							
LEAP Memories without on-chip caches							
1	0	58709	59029	21	571.5	18.0 ms	**1**
4	0	67867	67130	19	586.5	5.9 ms	3.1
LEAP Memories with on-chip caches (1 kB)							
1	1	64860	64820	24	583.5	19.4 ms	0.9
4	8	91401	88830	38	634.5	6.4 ms	2.8
Reflect Tree (36862 tree nodes)							
LEAP Memories without on-chip caches							
1	0	64471	65953	37	231.5	547.5 ms	**1**
4	0	95483	99269	97	360.5	194.0 ms	2.8
LEAP Memories with on-chip caches (1 kB)							
1	3	70662	72437	46	243.5	320.6 ms	1.7
4	12	118226	123215	129	408.5	79.9 ms	6.9
Filter (32767 kd-tree nodes, 128 clusters)							
LEAP Memories without on-chip caches							
1	0	72980	74050	57	275	897.3 ms	**1**
4	0	128163	128587	179	486.5	415.8 ms	2.2
LEAP Memories with on-chip caches (1 kB)							
1	4	83077	83493	67	296	464.7 ms	1.4
4	16	163849	164620	218	558.5	145.6 ms	6.2

i. Hybrid Multi-Cache Architectures

Table 1 quantifies the acceleration and resource consumption of parallelization and the multi-cache architecture N_c is the number of inserted caches. The default size of all caches is 1 kB. For each benchmark, we set the unparallelized ($P = 1$) design with no caches as a baseline reference (top row for each benchmark). The ratio S is the speed-up of each configuration compared to the baseline reference case ($S = 1$).

Adding single caches to the unparallelized implementations ($P = 1$) brings a speed-up of 1.7× and 1.4× for **Reflect tree Filter**, respectively. Parallelization with $P = 4$ results in 2.2× to 2.8× speed-up over the unparallelized baseline if the memory interface is not supported by caches. We observe further latency improvements when these parallelized applications are supported by multiple caches, which provides an overall acceleration of 6.2× to 6.9× for the tree-based benchmark. The small

caches mostly reduce the memory access time for the stack and center set data structures, as opposed to the tree data structures which are substantially larger. As we shall see in Section ii, the system performance is further improved by cache scaling. **Merger** is an extreme case in this evaluation because inserting the small 1 kB caches slightly slows down the implementations for both $P = 1$ and $P = 4$. The reason is the size of the data structures: only 2048 list elements fit in the caches, which is a small fraction of the entire data structure, resulting in a poor hit rate. The improvement of the memory access latency by the caches thus does not outweigh the small overhead in terms of cycle count because of the buffered banked cache memories [19]. Parallelization improves the net speed-up, but we shall see in the next sections that scaled-up caches further improve the overall latency significantly.

In addition to aggregate latency, we evaluate the benefit of cache architecture specialization. Our analysis determines that **Merger** requires P private memories, while **Reflect Tree** and **Filter** require a hybrid architecture with private and coherent caches. We compare the results of our application-specific systems to an 'all-coherent' scenario where no knowledge of disjoint heap regions is available to generate the multi-cache system. Firstly, such a scenario requires a commutativity analysis for safe parallelization for all heap updates which significantly increases the burden of analysis. Secondly, all LEAP Memories must feature a coherency mechanism by default. We focus on the second aspect here and quantify the additional cost of such an all-coherent architecture in terms of loss of efficiency: Table 2 lists the implementation results for the designs with all-coherent memories. Each row also shows the increase in resource consumption, latency and the slices-latency product of the all-coherent (AC) default compared to the corresponding hybrid (HY) architecture in Table 1 which uses knowledge of private and shared heap regions ($\frac{AC-HY}{HY}$ in %). We measure the resource consumption in terms of logic slices here to obtain a metric which combines LUTs and FFs.

The AC versions use more logic and have longer latencies. The resource overhead is especially noticeable for DSP slices (70.5% up to 281.6%), but is also substantial for logic slices (19.5% to 25.4%). The area overhead is particularly large for **Merger**, because the application-specific memory architecture does not use a coherency network at all. The access latencies due to the additional coherency network are notably longer. Finally, we compare the efficiency of the implementations by the area-time product. For $P = 4$, our disjointness analysis and the ability to instantiate cheap private caches whenever possible brings an overall improvement of the slices-latency product of 53.8% to 93.5% (70.9% on average).

The results above quantify the advantage of a specialized application-specific multi-cache system. The following sections discuss the performance and trade-offs when scaling up the private caches in the above hybrid multi-cache system.

ii. Performance after Custom Cache Scaling

Our technique improves the aggregate hit rate of the multi-cache architecture. The following results show the impact of the custom cache sizing on the overall execution latency and on the FPGA resource usage once we scale the private caches of the hybrid multi-cache systems. For ease of comparison, we include the uncached case and the case with small default size caches (both from Section i). We compare four cases:

Case 1. No caches (as in Table 1)

Case 2. Small fixed cache size of 1 kB (as in Table 1)

Case 3. A fixed size for all caches but scaled up to the maximum possible size

Case 4. A variably-sized multi-cache system as delivered by our technique in Section VI

The clock rate target is set to 100 MHz in all cases and all designs meet this clock constraint. All caches have a line width of 64 bits. Table 3 shows the timing as well as the utilization of LUTs, FFs, DSP slices and 36k-RAM blocks. We also show the aggregate hit rate (measured) of all private caches and the execution latency. The parallelization degree is $P = 4$ in all cases. We compare the speed-up S with respect to the base case in Section i ($P = 1$, no caches).

In addition to more BRAM, we observe a sudden increase in LUT, FF and DSP utilization once caches are included in the LEAP Memories. LUTs and FFs increase only marginally when scaling the caches up, leaving the BRAM usage as the limiting factor. The hit rate and latency improvements for **Merger** are substantial and grow steadily with larger cache sizes. There is a significant asymmetry between the linked lists in the application and the large improvement of the variable sizing over a fixed sizing (Cases 3 and 4) is due to the fact that larger caches support longer lists. The overall speed-up after parallelization, private cache insertion and custom cache sizing is $S = 15.2$ over the baseline.

For the tree-based benchmarks, we see a different characteristic of the latency improvement. Even small caches lift the aggregate hit rate above 90%. This reflects the behavior in Fig. 4: the stack data structures are very small (but heavily accessed) compared to the tree structure in the average case and a small cache is sufficient to keep all data on-chip. Consequently, the optimization algorithm in Section iii opts to use more

Table 2: *Cost increase of all-coherent default compared to application-specific hybrid cache architectures.*

P: parallelisation degree; N_c: number of caches							
P	N_c	Slices	DSP	BRAM	Clock period /ns	Latency /ms	Area − time product
Merger (250000 random input key-value pairs)							
4	8	42875 (25.4%)	145 (281.6%)	642 (1.2%)	10.0 (0.0%)	7.83 (22.6%)	335.7 slices · s (53.8%)
Reflect Tree (36862 tree nodes)							
4	12	52665 (20.4%)	220 (70.5%)	504 (23.4%)	10.2 (2.2%)	128.5 (60.7%)	6765.5 slices · s (93.5%)
Filter (32767 kd-tree nodes, 128 clusters)							
4	16	65412 (19.5%)	375 (72.0%)	644 (15.3%)	10.1 (0.9%)	208.2 (43.0%)	13615.8 slices · s (70.9%)

Table 3: *Latency and resources after custom cache scaling.*

$P = 4$; S: speed-up over unparallelized, uncached baseline in Table 1							
Case	LUT	FF	DSP	BRAM	Hit rate	Latency	S
Merger (250000 random input values, baseline latency: 18.0 ms)							
1	67867	67130	19	586.5	0	5.9 ms	3.1
2	91401	88830	38	634.5	5.31%	6.4 ms	2.8
3	93528	89184	38	858.5	79.24%	2.4 ms	7.4
4	92871	89064	39	874.5	99.12%	1.2 ms	15.2
Reflect tree (36863 tree nodes, baseline latency: 547.5 ms)							
1	95483	99269	97	360.5	0	194.0 ms	2.8
2	118226	123215	129	408.5	90.12%	79.9 ms	6.9
3	136087	125046	126	743.5	95.50%	68.7 ms	8.0
4	119284	123253	128	736.5	98.27%	57.4 ms	9.5
Filter (32767 kd-tree nodes, 128 clusters, baseline latency: 897.3 ms)							
1	128163	128587	179	486.5	0	415.8 ms	2.2
2	163849	164620	218	558.5	94.12%	145.6 ms	6.2
3	168416	165278	221	886.5	96.19%	140.4 ms	6.4
4	164818	164168	219	878.5	98.72%	138.2 ms	6.5

Table 4: *Power and energy measurements.*

R: energy reduction compared to Case 1						
Case	P_{FPGA} /W	P_{SDRAM} /W	E_{FPGA} /mJ	E_{SDRAM} /mJ	E_{total} /mJ	R
Merger (250000 random input values)						
1	1.78	1.11	10.40	6.40	16.88	**1**
2	2.13	1.09	13.61	6.99	20.60	0.8
3	2.58	1.05	6.30	2.55	8.85	1.9
4	2.57	1.01	3.05	1.19	4.24	4.0
Reflect tree (36863 tree nodes)						
1	2.13	1.15	412.30	222.46	634.76	**1**
2	2.34	1.04	186.68	83.24	269.92	2.4
3	3.06	1.07	210.40	73.37	283.77	2.2
4	3.26	1.14	187.09	65.56	252.64	2.5
Filter (32768 kd-tree nodes, 128 clusters)						
1	2.25	1.31	936.46	542.55	1479.01	**1**
2	2.77	1.05	402.50	152.94	555.44	2.7
3	3.27	1.03	459.55	144.19	603.74	2.5
4	3.53	1.08	488.11	148.73	636.84	2.3

memory resources for the large tree structure. For **Reflect tree**, this improves the aggregate hit rate by 3% to 4% compared to a homogeneous maximum sizing. Although the hit rates for **Filter** and **Reflect tree** are similar, the latency improvement from cache scaling for **Filter** is small. This is mainly due to high core-internal computation between memory accesses, which makes the effect of a shorter access time to the tree data less significant. The overall improvement execution time after parallelization with $P = 4$, hybrid cache insertion and custom cache scaling is 9.5× and 6.5× over the unparallelized and uncached baseline implementation for **Reflect tree** and **Filter**, respectively.

iii. Energy Consumption

We quantify the impact of our cache insertion and scaling on the overall energy consumption. To this end, we measure the instantaneous power consumption of the FPGA and the board-level SDRAM while the applications are running. We collect power figures for three out of the 12 power rails on the VC707 board: `VCCINTFPGA` is the main supply of the FPGA and `VCCBRAM` is an additional block RAM supply. We combine both to obtain the main supply of the FPGA. The third rail is `VCC1V5`, a supply of the SDRAM. No other rail notably changes its power levels during execution of our applications. We integrate power over the three latencies defined in the previous section; we show the energy per input value for **Merger**, the energy per completed tree traversal for **Reflect tree** and the energy per clustering iteration for **Filter**. Table 4 shows the main energy consumption of the FPGA (E_{FPGA}), the energy attributed to the SDRAM (E_{SDRAM}) and the total energy for the four cases above. We also show the energy improvement R compared to Case 1 (uncached parallel implementation). Table 4 also shows the mean power consumptions P_{FPGA} and P_{SDRAM}.

For large caches, the extra power consumption is significant (up to 102%). The latency reduction must be large enough to counter this effect and improve E_{FPGA} and E_{total}. Large caches always improve the energy consumption with respect to a cacheless memory interface in our implementations. In all benchmarks, the application-specific cache sizing outperforms fixed sizing in terms of energy reduction.

iv. Analysis Complexity and Performance

We evaluate the tool execution time for the largest benchmark design points on an Intel i7-3770 machine with 16 GB memory. Table 5 shows the lines of C++ code (LOC) and the time for heap analysis (including the symbolic execution), for solving the MCKP and FPGA

Table 5: *Analysis time, compile time and FPGA usage.*

Benchmark	LOC	Heap analysis	MCKP	Compilation	FPGA usage
Merger	291	1546 s	< 1 s	6308 s	43.7%
Reflect tree	570	27 s	< 1 s	3748 s	55.9%
Filter	783	705 s	< 1 s	8628 s	76.4%

compilation. It also includes the resource usage in terms of FPGA slices. In the worst case, the time complexity of the current symbolic execution is exponential in the number of analyzed branching statements in the program, which may limit scalability. However, our analysis merges heuristically singleton heaplets into generalized formulae as described in [9], which results in a linear growth observed in our benchmarks.

State merging is also required for the convergence (termination) of our analysis. The decision under what conditions the merging is triggered builds on a heuristic, which works well in practice for common data structures such as trees and linked lists. However, for programs using more exotic data structures we cannot rule out that merging fails (non-convergence of the analysis) due to the incompleteness of the heuristic. Algorithm 1 tries to provably distinguish private from shared heap regions only up L peeled off iterations. This incompleteness may thus result in indicating sharing of a heaplet which in reality is private to a particular code section. Note that this does not compromise the soundness of our transformation but only performance as our tool would simply not parallelize the application or instantiates an unnecessary coherency mechanism in this case. Our analysis currently does not support pointer arithmetic.

VIII. Conclusion

Mapping dynamic memory operations to FPGAs is difficult, both in terms of analysis and implementation. In this work, we present an HLS design aid for synthesizing pointer-based C/C++ programs into efficient FPGA implementations. We target applications that perform computation on large heap-allocated data structures and that require access to an off-chip memory. We leverage the separation logic-based static program analysis in [9] to determine whether different program parts access disjoint, non-overlapping regions in the monolithic heap space in which case we trigger automated source-to-source transformations that automatically parallelize the application. Our extended analyzer also detects heap regions that are shared by multiple accessors in the parallelized implementation. An additional commutativity and associativity analysis decides whether the parallelization in the presence of shared memory regions is semantics-preserving. The information provided by the heap analyses is used to optimize the interface between the parallelized HLS kernel and an off-chip memory: we generate an application-specific multi-cache system, where disjoint heap partitions are mirrored in private, independent on-chip caches and interfaces to shared heap regions are supported where necessary with on-chip caches backed by (inherently more expensive) coherency mechanisms and a synchronization service. We observe a speed-up of up to 6.9× after parallelization and insert of a multi-cache system compared to the unparallelized and uncached application. Our hybrid multi-cache system outperforms a default all-coherent version by 69.3% on average in terms of the area-time product.

We combine the hybrid cache synthesis with custom cache sizing, which automatically uses up the left-over BRAM to scale up the size of the private on-chip caches. The size of each private cache is set individually in order to reach a size distribution across the parallel caches that maximizes the aggregate hit rate. The pre-synthesis cache performance estimation is based on a high-level cache model and on the memory reference trace of the application obtained from automated profiling. We cast the cache size assignment into a Multiple-Choice Knapsack Problem to find the best size distribution for a given reference trace. The overall reduction of execution time after parallelization, insertion of the hybrid multi-cache system and custom cache scaling is up to 15.2× (9.8× on average) over an unparallelized and uncached implementation. Although the insertion of large on-chip caches has a significant impact on the power consumption of the FPGA, we show that our variably-sized multi-cache configuration reduces the total energy by 2.5× (on average) compared to a cacheless memory interface.

There are two important extensions planned for future work. The current cache scaling targets private, independent caches. Future work will focus on a model of the coherency protocol in a cache architecture consisting of coherent caches, which must take additional invalidation and owner misses due to interfering accesses by other caches into account. Secondly, our energy measurements in Table 4 suggest that the optimal cache sizing changes when we optimize for energy instead of aggregate hit rate. Future work will address the development of an energy model that can be used to minimize the energy consumption of our multi-cache system.

Acknowledgment

This work was supported by the EPSRC (EP/I012036/1, EP/I020357/1, EP/K015108/1, EP/K034448/1), the Royal Academy of Engineering and Imagination Technologies.

References

[1] W. Meeus, K. Van Beeck, T. Goedemé, J. Meel, and D. Stroobandt, "An Overview of Today's High-Level Synthesis Tools," *Design Automation for Embedded Systems*, pp. 1 – 21, Aug. 2012.

[2] F. Winterstein, S. Bayliss, and G. Constantinides, "High-level synthesis of dynamic data structures: A case study using Vivado HLS," in *Proc. Int. Conf. on Field-Programmable Technology*, 2013, pp. 362–365.

[3] S. Cheng, M. Lin, H. J. Liu, S. Scott, and J. Wawrzynek, "Exploiting Memory-Level Parallelism in Reconfigurable Accelerators," in *Proc. Int. Symp. on Field-Programmable Custom Computing Machines*, 2012, pp. 157–160.

[4] A. Putnam, S. Eggers, D. Bennett, E. Dellinger, J. Mason, H. Styles, P. Sundararajan, and R. Wittig, "Performance and power of cache-based reconfigurable computing," *ACM SIGARCH Computer Architecture News*, vol. 37, no. 3, p. 395, Jun. 2009.

[5] Q. Liu, G. Constantinides, K. Masselos, and P. Cheung, "Combining Data Reuse With Data-Level Parallelization for FPGA-Targeted Hardware Compilation: A Geometric Programming Framework," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 28, no. 3, pp. 305–315, Mar. 2009.

[6] S. Bayliss and G. Constantinides, "Optimizing SDRAM bandwidth for custom FPGA loop accelerators," in *Proc. Int. Symposium on Field Programmable Gate Arrays*, 2012, pp. 195–204.

[7] L.-N. Pouchet, P. Zhang, P. Sadayappan, and J. Cong, "Polyhedral-based data reuse optimization for configurable computing," in *Proc. Int. Symp. on Field Programmable Gate Arrays*, 2013, pp. 29–38.

[8] P. O'Hearn, J. Reynolds, and H. Yang, "Local reasoning about programs that alter data structures," in *Proceedings of the 15th International Workshop on Computer Science Logic - CSL'01*, 2001, pp. 1–19.

[9] F. J. Winterstein, S. R. Bayliss, and G. A. Constantinides, "Separation logic for high-level synthesis," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 9, no. 2, pp. 10:1–10:23, Dec. 2015.

[10] F. Winterstein, K. Fleming, H.-J. Yang, S. Bayliss, and G. Constantinides, "MATCHUP: Memory Abstractions for Heap Manipulating Programs," in *In Proc. ACM/SIGDA Int. Symp. on Field-Programmable Gate Arrays*, 2015, pp. 136–145.

[11] "The LLVM Compiler Infrastructure." [Online]. Available: http://llvm.org/

[12] F. Winterstein, K. Fleming, H.-J. Yang, J. Wickerson, and G. Constantinides, "Custom-sized caches in application-specific memory hierarchies," in *In Proc. Int. Conf. on Field Programmable Technology*, 2015, pp. 144–151.

[13] K. Fleming, H.-J. Yang, M. Adler, and J. Emer, "The LEAP FPGA Operating System," in *Proc. Int. Symp. on Field Programmable Logic and Appl.*, 2014, pp. 1–8.

[14] M. Adler, K. E. Fleming, A. Parashar, M. Pellauer, and J. Emer, "Leap Scratchpads: Automatic Memory and Cache Management for Reconfigurable Logic," in *Proc. Int. Symp. on Field Programmable Gate Arrays*, 2011, pp. 25–28.

[15] H.-J. Yang, K. Fleming, M. Adler, and J. Emer, "LEAP Shared Memories: Automating the Construction of FPGA Coherent Memories," in *Int. Symp. on Field-Programmable Custom Computing Machines (FCCM)*, May 2014, pp. 117–124.

[16] V. Mirian and P. Chow, "Fcache: A system for cache coherent processing on fpgas," in *Proc. ACM/SIGDA Int. Symp. on Field Programmable Gate Arrays*, 2012, pp. 233–236.

[17] E. Matthews, N. C. Doyle, and L. Shannon, "Design Space Exploration of L1 Data Caches for FPGA-Based Multiprocessor Systems," in *Proc. Int.Symp. on Field-Programmable Gate Arrays*, 2015, pp. 156–159.

[18] J. Choi, K. Nam, A. Canis, J. Anderson, S. Brown, and T. Czajkowski, "Impact of Cache Architecture and Interface on Performance and Area of FPGA-Based Processor/Parallel-Accelerator Systems," in *Int. Symp. on Field-Programmable Custom Computing Machines*, 2012, pp. 17–24.

[19] H.-J. Yang, K. Fleming, M. Adler, F. Winterstein, and J. Emer, "Scavenger: Automating the construction of application-optimized memory hierarchies," in *In Proc. Field Programmable Logic and Applications*, 2015, pp. 1–8.

[20] T. Kanungo, D. Mount, N. Netanyahu, C. Piatko, R. Silverman, and A. Wu, "An efficient k-means clustering algorithm: analysis and implementation," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 24, no. 7, pp. 881–892, Jul. 2002.

[21] M. Raza, C. Calcagno, and P. Gardner, "Automatic parallelization with separation logic," in *Programming Lang. and Syst.*, 2009, pp. 348–362.

[22] M. C. Rinard and P. C. Diniz, "Commutativity analysis: a new analysis technique for parallelizing compilers," *ACM Transactions on Programming Languages and Systems*, vol. 19, no. 6, pp. 942–991, Nov. 1997.

[23] "Z3: An Efficient SMT Solver." [Online]. Available: http://z3.codeplex.com/documentation/

[24] A. Charlesworth, "The undecidability of associativity and commutativity analysis," *ACM Transactions on Programming Languages and Systems*, vol. 24, no. 5, pp. 554–565, Sep. 2002.

[25] M. Brehob and R. Enbody, "An analytical model of locality and caching," Department of Computer Science, Michigan State University, Tech. Rep., 1996.

[26] K. Beyls and E. H. DâĂŹHollander, "Reuse distance as a metric for cache behavior," in *In Proc. IASTED Conf. on Parallel and Distributed Computing and Systems*, 2001, pp. 617–662.

[27] D. Pisinger, "A minimal algorithm for the 0-1 knapsack problem." *Operations Research*, vol. 45, pp. 758–767, 1994.

Felix Winterstein Felix Winterstein (S'13-M'16) has a Master's degree in electrical engineering from RWTH Aachen University, Germany, and a Ph.D. degree from Imperial College London. He was an engineer at the European Space Agency from 2010 to 2015 and is currently a research associate in the Circuits and Systems Group at Imperial College London. His research interests are high-level synthesis and memory system optimizations for reconfigurable computing.

Kermin E. Fleming Kermin E. Fleming has a Master's degree from Carnegie Mellon University and a Ph.D. degree from Massachusetts Institute of Technology (MIT). He is currently a member of the Software Services Group at Intel. His work is focused mainly on tools and techniques for implementing high-performance reconfigurable systems and architectures.

Hsin-Jung Yang Hsin-Jung Yang received her Bachelor's degree in Electrical Engineering from National Taiwan University, Taiwan, in 2010, and her Master's degree in Electrical Engineering and Computer Science (EECS) from Massachusetts Institute of Technology (MIT), USA, in 2012. She is currently pursuing her Ph.D. degree in EECS at MIT. Her current research interests include high-performance architectural design for reconfigurable computing, with an emphasis on memory system optimizations.

George A. Constantinides George A. Constantinides (S'96-M'01-SM'08) received the Ph.D. degree from Imperial College London in 2001. Since 2002, he has been with the faculty at Imperial College London, where he is currently Professor of Digital Computation and Head of the Circuits and Systems research group. He was the general chair of the ACM International Symposium on Field-Programmable Gate Arrays in 2015. He serves on several program committees and has published over 150 research papers in peer refereed journals and international conferences. Dr Constantinides is a Senior Member of the IEEE and a Fellow of the British Computer Society.