MATCHUP: Memory Abstractions for Heap Manipulating Programs

Felix Winterstein
European Space Agency
Ground Systems Eng. Dept.
f.winterstein12@ic.ac.uk

Kermin Fleming
Intel Corporation
VSSAD Group
kermin.fleming@intel.com

Hsin-Jung Yang
Massachusetts Institute of Technology, CSAIL
hjyang@csail.mit.edu

Samuel Bayliss
Imperial College London
Circuits and Systems Group
s.bayliss08@ic.ac.uk

George Constantinides
Imperial College London
Circuits and Systems Group
g.constantinides@ic.ac.uk

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 generates parallel application-specific multi-scratchpad architectures including on-chip caches. Our program analysis identifies non-overlapping memory regions, supported by private scratchpads, and regions which are shared by parallel units after parallelization and which are supported by coherent scratchpads 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 dynamic, pointer-based data structures and dynamic memory allocation which, while common in software engineering, remain difficult to analyze and are beyond the scope of the overwhelming majority of HLS techniques to date. We demonstrate our technique with three case studies of applications using dynamically allocated data structures and use Xilinx Vivado HLS as an exemplary HLS tool. We show up to 10× speed-up after parallelization of the HLS implementations and the insertion of the application-specific distributed hybrid scratchpad architecture.

Categories and Subject Descriptors

B.5.2 [**Design Aids**]: Automatic synthesis

General Terms

Algorithms, Design

Keywords

High-Level Synthesis, Caching Schemes, Separation Logic

FPGA'15, February 22–24, 2015, Monterey, California, USA.

http://dx.doi.org/10.1145/2684746.2689073.

1. INTRODUCTION

High-level synthesis (HLS) raises the abstraction level from register transfer level (RTL) to high-level languages, such as C/C++, and can significantly shorten the design cycle when developing applications for field-programmable gate arrays (FPGAs) as compared with an RTL-based specification. State-of-the-art C-to-FPGA tools such as Xilinx Vivado HLS, ROCCC [2] or LegUp [1] can deliver a quality of results (QoR), measured in terms of latency and resource utilization, close to hand-written RTL implementations [12, 17]. The extraction of parallelism is crucial for achieving a good QoR. Computational parallelism also requires that the memory system is not a sequential bottleneck to performance. The distributed memory architecture in FPGAs can provide impressive memory bandwidth if the program data can be partitioned and distributed over multiple on-chip memory banks. Parallel on-chip memory capacity remains a scarce resource and many FPGA applications that process large data sets require access to a large off-chip memory. Accessing external memory, however, can substantially slow down an FPGA accelerator due to memory bandwidth limitations and, in the worst case, the contention on the external memory bus eliminates the gain of parallelization. An application-specific optimization of the on-chip/off-chip memory architecture is thus crucial for mapping a program to an efficient FPGA implementation.

Caching frequently reused data is a common approach to reduce the number of expensive accesses to an external memory. FPGAs allow the implementor to tailor the memory interface according to the requirements of the application. An application-specific optimization of this architecture introduces yet another complex design space and remains a complex task for a developer. Furthermore, automatic cache design in an HLS context requires the extraction of application-specific properties from program descriptions and remains foreign to most HLS flows. This work seeks to bridge this gap. We present an HLS design aid that inserts multiple on-chip caches into the interface to an off-chip memory which results in an application-specific high-performance memory hierarchy. Our technique leverages recent memory abstractions [9, 10], which build an on-chip/off-chip memory hierarchy underneath a uniform interface and which we refer to as *scratchpads* (SPs) in this paper. Each single SP contains an

optional on-chip cache and automatically ensures coherency between the cache contents and data in off-chip memory for an arbitary memory access pattern [9]. SPs also provide an optional mechanism to maintain coherency between the on-chip caches in multiple, parallel SPs [10]. In this work, we leverage a program analysis to determine whether or not such an inter-scratchpad coherency mechanism is required in the generated multi-scratchpad architecture.

This work is based on a static program analysis that extracts memory access information. It focuses on heap-manipulating C/C++ programs, making this work a complement to existing work based on run-time profiling [15], manual code annotations [20] or automated analyses that target explicitly static arrays referenced in static loop nests (leveraging the *polyhedral model*) [11, 16, 19]. Our motivation is driven by the fact that pointer-based memory references and dynamic memory allocation are well established and widely used features of high-level languages such as C++, their analysis and automated program optimizations resulting from it, however, are beyond the scope of the most HLS techniques to date. The memory model in 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 used in this work is based on *separation logic* [18], a theory for reasoning about the behavior of programs that finds its main application in modern software verification tools. We build on a baseline analysis presented in [7] which determines whether (parts of) the accessed heap memory can be completely partitioned into disjoint, non-overlapping portions, which we refer to as *heaplets*. Finding a solution to this question is a pre-requisite for parallelizing the loop by splitting it into parallel sub-loops 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 [7] is limited to cases where the on-chip memory capacity is sufficient and the accessed memory space can be split into independent, private partitions. In this paper we extend it to shared resources and apply it to the synthesis of efficient interfaces to an off-chip memory. To the best of our knowledge, this is the first application of a separation logic-based analysis to an automated optimization of the on-chip/off-chip memory hierarchy for FPGA accelerators. The contributions are:

- In addition to the identification of disjoint heap regions, we extend the baseline analysis in [7] 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.

- Even if coherency is ensured, updates to the shared resource may happen in a different order after parallelization compared to the sequential program. This paper presents a *commutativity analysis* for the shared heap update in order to prove that the parallelization is semantics-preserving.

- The framework targets 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 an application-specific parallel multi-scratchpad architecture containing on-chip caches and (if needed) coherency mechanisms: We synthesize parallel private scratchpads for disjoint heap regions and (inherently more expensive) coherent parallel scratchpads for shared regions.

- 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 in our case studies. We use the open-source LEAP infrastructure [8] and implement our test cases on a Virtex 7 FPGA connected to a DDR3 memory.

```
1  //main kernel function
2  void traverse(TR *root, CI *z) {
3    CS* c0 = new CS;
4    *c = ...;
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      TR tn = *u;
10     CS cs = *c;
11     if (d) {
12       delete c;
13     }
14     CS *cnew = new CS;
15     *cnew = subfunction1(cs);
16     if  (tn.left!=NULL) && (tn.right!=NULL)
           && subfunction2(cs) {
17       s = push(tn.left, cnew, true, s);
18       s = push(tn.right, cnew, false, s);
19     } else {
20       delete cnew;
21       CI w = tn.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 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 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: C-like pseudo code from a K-means clustering kernel [5].

Section 2 presents a motivating example for this work. Section 3 describes our program analyses. Section 4 gives a brief overview of the implementation followed by a presentation of our experimental results in Section 5. Section 6 discusses related work and Section 7 concludes the paper.

2. MOTIVATING EXAMPLE

This section reviews a motivating example in the context of our previous work in [7] and explains how the extensions of baseline analysis in [7] are applied to generate a multi-

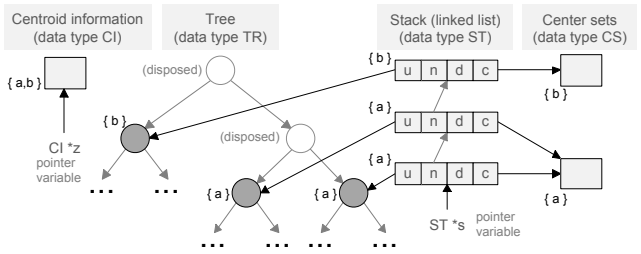

Figure 1: Snapshot of the pointer-linked dynamic data structures accessed by the loop in Listing 1.

scratchpad architecture for both private and shared heap regions. Listing 1 shows C-like pseudo code taken from (a modified version of) a tree-based K-means clustering implementation [5]. The `while`-loop in `traverse` 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 (type `CI`). The tree has been built up from the data set to be clustered. The center sets are intermediate 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. Retrieving these pointers from it and `new`/`delete` operations on its head are performed by the auxiliary functions `push` (lines 5, 17 and 18) and `pop` (line 8). If the data-dependent conditional (line 16) 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.

All data structures accessed by this program are created at run-time using dynamic memory allocation. Allocating memory at run-time 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 scratchpads including on-chip caches which mirror parts of the off-chip data and provide fast data access. We describe the baseline method in [7] and its extensions are described below.

2.1 Memory Partitioning and Parallelization

This section reviews the partitioning method in [7] and demonstrates its application to the novel multi-scratchpad synthesis. Fig. 1 shows an example of the data structures allocated in the heap after executing two `while`-loop iterations. The data structures are grouped according to their types. The loop is split into in parallel sub-loops as shown in Listing 2 (two in this example). If we ignore the centroid data structure (type `CI`) in the heap layout in Fig. 1 for a moment, the method in [7] can prove that the pointers dereferenced in any iteration of a sub-loop never refer to the data structures used by the other loop. We call these loop kernels 'communication free' with respect to each other, which satisfies the 'independence condition' that two parts of a program can operate in parallel if they access different data. A standard HLS tool can use the independence information to instantiate parallel hardware blocks. The analysis partitions

```
1  //main kernel function
2  void traverse(TR *root, CI *z) {
3    ...preamble (pointers access partitions a
         and b)
4    while (s_a != NULL) {
5      //parallel loop kernel a
6      ..access private SP for CS, partition a
7      ..access private SP for ST, partition a
8      ..access private SP for TR, partition a
9      acquireLock();
10     ..access coherent SP for CI, partition a
11     releaseLock();
12   }
13   while (s_b != NULL) {
14     //parallel loop kernel b
15     ..access private SP for CS, partition b
16     ..access private SP for ST, partition b
17     ..access private SP for TR, partition b
18     acquireLock();
19     ..access coherent SP for CI, partition b
20     releaseLock();
21   }
22 }
```

Listing 2: Transformed program from Listing 1.

the remaining tree data structure (dark gray nodes, type `TR`) into two sub-trees labeled with $\{a\}$ and $\{b\}$. It splits the linked list (type `ST`) into the uppermost node and the nodes below, and the pool of center sets (type `CS`) is partitioned accordingly. The generation of the multi-scratchpad architecture in this work uses the heap partitioning information from the baseline analysis. Each of the parallel sub-loops obtains its own interfaces to off-chip memory and the fact that the memory regions can be proven to be non-overlapping allows our setup to instantiate private SPs for each partition without the need to ensure coherency between them, greatly reducing hardware implementation cost.

2.2 Parallel Access to Shared Resources

Our baseline analysis in [7] cannot handle situations including the shared centroid information in Fig. 1. In this work, we present an extension which 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 [10] to this region in each of the sub-loops. The coherent interface consists of two parts: SPs with caches and a coherence mechanism that ensures data coherency between them 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 shared centroid information is updated in line 23:

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

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. 1). However, in the parallelized version z may be updated with data from left and right sub-tree in an

arbitrarily interleaved fashion. In this example, the parallelization is legal because of the commutativity and associativity of the addition[1]. In general, we address this question with a commutativity analysis of the update function.

Listing 2 shows the final result of a source code transformation based on the result of all analyses above. The transformed source code, when run through a back-end HLS tool and RTL implementation, results in a custom configuration of multiple private/coherent scratchpads with a custom degree of parallelism. The on-chip memory blocks in modern FPGAs, a distinguishing feature compared to microprocessors, are aggregated accordingly in order to construct the application-specific parallel caching scheme. The difficult part of these memory system optimizations is the heap analysis: Heap-directed pointer expressions are not restricted to a particular scope and can reference any memory cell in the global heap, which may lead to dependencies between expressions in the program that are syntactically unrelated. Furthermore, the values of pointers change during runtime. A static analysis ruling out pointer aliasing relations is particularly challenging for pointer-linked data structures. Separation logic provides an efficient theoretical framework for expressing the heap layout and alias information during program execution as described below.

3. PROGRAM ANALYSIS

This section describes the program analyses enabling the source 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 [18] and the heap analysis developed previously in [7], which we refer to the 'baseline analysis' and which this work builds on. New extensions of the baseline are described after Section 3.1.

3.1 Background

Our static analysis is based on the *symbolic execution* of an imperative 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 program's control flow graph (CFG). Branching creates multiple control flow paths and loops additionally 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 contraint: $u \neq v \wedge u \mapsto 1 \wedge v \mapsto 6$. These constraints are required for each pair of pointers in the program which quickly limits the applicability of an automated heap analysis to real-life programs that arise in practice.

Separating 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 symbolic execution implements a technique called *labeled symbolic execution* [21]. For each program statement, it assigns a unique label to the accessed heap portions of the current state. In the original work in [21], each program statement C_i is assigned a unique label l_i. The technique thus propagates the 'heap footprint' of each statement through the CFG. Our heap access analyses use modified versions of labeled symbolic execution in order to find disjoint and shared heap regions.

Our baseline analysis for identifying private heap regions and memory partitioning [7] 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 [7]. It starts with a symbolic execution of the loop preamble and a finite number of the first loop iterations (function SymbExeLoopBody). In each step, it explores the separation logic formula describing the pre-state of the loop ($\Pi \wedge \Sigma$), *i.e.* the program state before executing the loop body. Our analysis is based on *cut-points*: A cut-point is a program variable pointing

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

to a heaplet in the symbolic heap [7]. The algorithm inserts cut-points into the loop pre-state formulae (function CutpInsert) while it peels off loop iterations so as to find a valid cut-point assignment. We define a cut-point as valid if 1) it consists of P cut-points, 2) these P cut-points reference heaplets of the same shape, *i.e.* describe the same type of data structure, 3) the built-in proof engine, performing a *fix-point calculation* for the loop-under-analysis (function FixpCalc) proves that the initial partitioning of the heap-allocated data structures is maintained for all subsequent loop iterations. Note that, for ease of explanation, we omit cases in Algorithm 1 where multiple alternative cut-point insertions for the same number of unrolled iterations are available and need to be checked as discussed in [7].

The proof of loop-invariant resource separation is generated by assigning a state to each inserted cut-point (function AssignCPStates). The fix-point calculation assigns footprint labels to the accessed heaplets according to the current cut-point state, which changes once a heaplet referenced by a different cut-point is accessed during the symbolic execution. Complete partitioning of the heap accessed by the loop-under-test is proven by the absence of non-singleton label sets attached to the heaplets in the state formulae. A detailed description of the technique is given in [7]. If we ignore the centroid information in the motivating example, starting from the pre-state in Fig. 1 and with a second cut-point s_b (in addition to s) referencing the uppermost stack record in Fig. 1, the proof of complete separability is generated by our baseline analysis, a prerequisite for parallelization.

3.2 Detecting Private and Shared Resources

The baseline analysis in [7] is limited to cases where the accessed memory space can be split into independent, private partitions. We aborted the analysis reporting a failed proof after a fixed parameter of L unrollings if the program state cannot be completely partitioned. Here, we relax the constraint that the inherent parallelism of the application needs to be communication-free. Algorithm 1 shows the extended baseline analysis to identify disjoint and shared resources. If we include the centroid information in our motivating example and run the disjointness analysis, the proof engine 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. We run the cut-point insertion and fix-point calculation with the objective of splitting the heap into P partitions as in [7], as shown in the inner `do-while`-loop in Algorithm 1. After peeling off the first loop iteration of the motivating example, the function FixpCalc terminates unsucessfully because it finds non-singleton label sets attached to a center set and the centroid information. After unrolling two iterations, the sharing of a center set disappears and the centroid information remains as the only shared resource. We use a heuristic approach to filter shared resources by declaring all heaplets which have a non-singleton label sets after L unrollings as

Algorithm 1 Detecting Private and Shared Resources.

```
Input:
Loop body specification (code)
Initial state formula (Π ∧ Σ)_initial
Output:
Assignment of pointer statements to heap partitions
Number of initial unrollings
Shared/private predicate for pointer statements
Variables:
it: Iteration counter (number of iterations to be unrolled)
C: Set of cut-points
C_shared: Set of cut-points referencing shared heaplets
S_cutpoints: Set of cut-point states
Π ∧ Σ: Loop pre-state formula
StmtS: Set of statement sets accessing shared heaplets
procedure HeapAnalysis
    C_shared ← ∅
    do
        it ← 0
        C ← ∅
        Π ∧ Σ ← (Π ∧ Σ)_initial
        StmtsS ← ∅
        success ← false
        do
            while C not valid do
                Π ∧ Σ ← SymbExeLoopBody(Π ∧ Σ, it)
                Π ∧ Σ, C ← CutpInsert(Π ∧ Σ, C_shared)
                it ← it + 1
            end while
            S_cutpoints ← AssignCPStates(C)
            success, Stmts_shared ← FixpCalc(
                Π ∧ Σ, C, S_cutpoints, C_shared)
            StmtsS ← StmtsS ∪ {Stmts_shared}
        while (not success) and (it < L)
        if it = L then
            C_shared ← ExtrctCutp( argmin_{Stmts∈StmtsS} |Stmts|)
        end if
    while not success
    ... generate analysis output
end procedure
```

shared. 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 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 adding a partition label to a heaplet 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 heap region residing in off-chip memory is then supported by a coherency protocol. The corresponding program statements accessing the shared resource are lines 22 and 23. The notion of these statements, extracted from the anaysis, 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.

3.3 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 because the order in which the updates of the shared resource are made by the parallel version is 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 subtree 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 [25] 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 symbolic execution and detection of the shared resources as above, we extract the pre- and post-conditions on the program state:

$$\{w = w'_0 \wedge z \mapsto [\texttt{wgtCent} : w'_1]\} \quad (1)$$
$$F$$
$$\{w = w'_0 \wedge w'_2 = w'_1 + w'_0 \wedge z \mapsto [\texttt{wgtCent} : w'_2]\}$$

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 by creating a fresh primed variable w'_2. We test whether F is commutable by symbolically executing two sequences of two calls to F:

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

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:

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

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:

$$w'_3 = w'_4 \quad (6)$$

Next, we combine the verification condition (6) with the remaining pure parts of the formulae and aim to prove:

$$\forall w'_{0,2}, w'_{0,1}. \quad (7)$$
$$w = w'_{0,3} \wedge w'_3 = w'_1 + w'_{0,1} + w'_{0,2} \wedge$$
$$w = w'_{0,3} \wedge w'_4 = w'_1 + w'_{0,2} + w'_{0,1} \Rightarrow (w'_3 = w'_4)$$

In the actual verification step, we use satisfiability modulo theories (SMT) solving [4] to decide (7). However, an SMT solver cannot deal with the universal quantification ($\forall$), so we rephrase (7) by negating the verification condition:

$$\exists w'_{0,2}, w'_{0,1}. \quad (8)$$
$$w = w'_{0,3} \wedge w'_3 = w'_1 + w'_{0,1} + w'_{0,2} \wedge$$
$$w = w'_{0,3} \wedge w'_4 = w'_1 + w'_{0,2} + w'_{0,1} \wedge (w'_3 \neq w'_4)$$

The solver returns one of three possible results: 1) If (8) is satisfiable, we can find an assignment to the input variables $w'_{0,2}, w'_{0,1}$ of F that makes the program states after executing both sequences different: F is not commutable. 2) If (8) 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 shown to be an undecidable problem in general [14]. However, it can still be shown for many cases that arise in practice. Our analysis is conservative: If we cannot decide it, we declare the update function non-commutative and abort the parallelization.

3.4 Robustness of the Heap Analysis

The heap analysis is the core element of our framework. We briefly discuss its performance and identify weaknesses motivating future research. An advantage of our technique is that it can, beyond straightline code and deterministic static control parts such as unrollable `for`-loops, handle `while`-loops enclosing data-dependent conditionals, and with data-dependent loop condition and unknown iterations count. This feature requires us to describe data structures of unknown size to ensure convergence of the fix-point calculation. We achieve this with recursive predicates as discussed in [7]. For instance, an acyclic linked list is described by:

$$ls(x, y) \Leftrightarrow (x = y \wedge emp) \vee$$
$$(x \neq y \wedge x \mapsto [\texttt{n} : x'_1] * ls(x'_1, y)\ ) \quad (9)$$

Our analysis is based on symbolic execution which mimics the actual program execution and heap accesses. In contrast to a *reachability analysis* [22], another common approach to

disjointness detection, which relies on the reachability properties of certain pointer data structures (*e.g.* left and right sub-tree), the separation logic-based heap footprint analysis can also partition cyclic data structures, such as a circular linked list or a doubly linked list.

Folding singleton heaplets into recursive predicates is essential for the successful termination of the loop analysis. For example, our analysis automatically folds

$$s \mapsto [\mathtt{n} : s_1'] * s_1' \mapsto [\mathtt{n} : s_0'] * s_0' \mapsto [\mathtt{n} : \mathtt{nil}] \qquad (10)$$

into $ls(s, \mathtt{nil})$. The recursive predicates are defined in logic rules used by the built-in theorem prover which automatically searches for applicable rules. We define a set of predicates for common data structures such as trees, lists, lists with additional pointers to singleton heaplets and sub-trees. These allow us to cover a large range of pointer-based programs. However, we may find applications using more exotic structures for which no folding rule in our current set applies. This limitation can be removed by integrating algorithms for automatic inference of recursive predicates, such as [24] in our tool. The decision under what conditions the folding is triggered builds on a heuristic [23] which 'gobbles up' heaplets by recursive predicates if their pointers are primed variables which do not appear in any other part of the formula except of the predicates involved in the folding. The heuristic works well in practice and we are not aware of a theorem prover implementing a more robust technique. However, in general, we cannot rule out situations where the folding fails due to the incompleteness of the heuristic. A code example where this is the case in given in [23]. The same holds for Algorithm 1 itself which uses a heuristic to distinguish private from shared heap regions. Our analysis may thus indicate sharing of a heaplet which in reality is private to a particular code section. Note that this does not compromise correctness but only performance as our tool instantiates an unnecessary coherency mechanism in this case. We argue that the benchmark applications in Section 5 are representative of common pointer-based programs.

The scalability of the analysis is determined by the fix-point calculation which performs repeated symbolic executions of the loop body until convergence. Non-deterministic branching (*e.g.* data dependent conditionals) in the loop body results in several disjunctive clauses describing the loop state as all control flow paths must be analyzed. In the worst case the number of these clauses can grow exponentially with the number of fix-point iterations. However, absorbing heaplets in recursive predicates contains the growth of the state formula. For moderate parallelism degrees, we observe a maximum of 28 disjunctive clauses which results in up to 7 minutes for the fix-point calculation.

The next section describes our compilation flow that uses the information provided by the above program analyses to generate application-specific multi-scratchpad architectures.

4. CODE GENERATION

We implemented a prototype tool of our technique consisting of three parts main parts: 1) The heap analyzer implements the analyses described in the previous section and interfaces the Z3 SMT solver [4]. 2) The source-to-source translator is built on the ROSE compiler infrastructure [3]

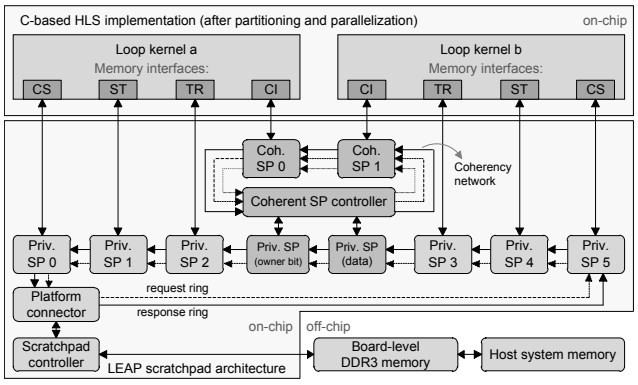

Figure 2: Parallelized HLS implementation of the filtering algorithm with a hybrid cache architecture.

and implements the loop parallelization and pointer access transformations. The heap analyzer and source translator are extensions of the baseline framework described in [7]. 3) We leverage the open-source LEAP (Latency-insensitive Environment for Application Programming) framework [8] to embed the C/C++-based HLS kernels in 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 the underlying FPGA device, on-board memory and the host system an FPGA card is plugged into. In particular, our setup uses LEAP's *scratchpads* (SPs), a memory interface abstraction for FPGA applications. SPs provide a simple read-request, read-response and write memory interface to the connected application. Internally, Leap scratchpads instantiate a memory hierarchy: an optional on-chip cache, board-level off-chip memory and finally the main memory of the attached host system as shown in Fig. 2. SPs 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. The framework provides two types of SPs: 1) *Private scratchpads* [9] are instantiated when memory spaces are known to be disjoint from all regions accessed by other memory interfaces. 2) If several memory interfaces refer to a shared memory region we instantiate *coherent scratchpads* [10]. The latter feature consists of distributed caches backed by a coherence protocol. Multiple coherent SPs appear as independent interfaces to the application, while they are internally connected via a ring network that ensures coherency between them. The shared memory abstraction by coherent scratchpads hides the internals of the coherency mechanism. Coherent SPs are more expensive (in terms of FPGA resources) and slower (in terms of response time) than their private counterparts.

Our source translator replaces the basic C++ routines for dynamic memory allocation with custom implementations to ensure synthesizability by off-the-shelf HLS tools. Occurrences of `new` and `delete` statements are grouped according to the type of their operand and custom allocator functions are instantiated for each type. The fixed-size allocator is a standard implementation using a *free-list* which keeps track of occupied memory space [6]. Heap memory is replaced by arrays located in off-chip memory by default (a portion of

```
requestLock(access_critical_region0);
waitForLock(); //stalls until lock has been
    acquired
...issueMemoryRequest //set memory fence
releaseLock(access_critical_region0);
```

Listing 3: Lock-synchronized shared memory access.

them resides on-chip via caches) and each heap access becomes an access to the external memory bus. The 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 heap analyzer provides information whether the memory bus points to a private or a shared heap region. We insert a generic Verilog wrapper for each interface that acts as a bridge between Vivado's native bus protocol and the LEAP memory 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 SP.

Fig. 2 shows the integration of our running example after heap partitioning and parallelization with $P = 2$ into the LEAP framework and memory hierarchy. Each loop kernel (we omit the preamble here) has an interface to the memory system for the 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 in order to ensure fine-grain atomic updates to the shared heap cell. Listing 3 shows an example. The pass-by-reference argument `access_critical_region0` translates into a boolean signal in the generated RTL code and triggers lock acquisition and releasing. The lock service provided by LEAP 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.

5. EXPERIMENTS

We run our experiments with three C++ applications that traverse, update, allocate and dispose dynamic data structures in heap memory. Our benchmark applications are:
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 scratchpads are inserted in the parallelized implementation.
Tree Deletion - This application traverses binary tree structure and deletes the visited tree nodes after some computation at each node. Our analysis splits the tree and stack (a linked list implementing the tree traversal) into P disjoint sub-structures after peeling off the first loop iteration. During tree traversal, the program updates a running minimum which is heap-allocated and detected as a shared resource. Commutativity of the min-reduction can be shown. P coherent scratchpads 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 scratchpads and a lock service.

Table 1: Parallelization and caching.

P	N_c	Slices	DSP	BRM	Clock	Lat.	S
Merger (4×2048 random input key-value pairs)							
scratchpads without on-chip caches							
1	0	18080	3	42	10.0ns	1257.9ms	**1.0**
4	0	19338	3	62	10.0ns	533.8ms	2.4
scratchpads with on-chip caches (32 KBytes)							
1	1	20807	3	62	10.0ns	757.6ms	1.7
4	4	22961	3	72	10.0ns	115.3ms	10.9
Tree Deletion (2048 tree nodes)							
scratchpads without on-chip caches							
1	0	24711	15	52	10.0ns	6575.2us	**1.0**
2	0	26127	15	61	10.0ns	3601.8us	1.8
4	0	29293	15	91	10.0ns	2207.9us	3.0
scratchpads with on-chip caches (32 KBytes)							
1	3	31754	18	82	10.0ns	5928.7us	1.1
2	6	37740	19	111	10.0ns	2604.3us	2.5
4	12	41860	23	202	10.5ns	711.0us	9.2
Filter (32767 kd-tree nodes, $K = 128$ clusters)							
scratchpads without on-chip caches							
1	0	26508	39	69	10.0ns	135.8ms	**1.0**
2	0	30594	103	92	10.0ns	81.9ms	1.7
4	0	38362	235	125	10.0ns	61.6ms	2.2
scratchpads with on-chip caches (32 KBytes)							
1	4	34085	41	102	10.0ns	93.4ms	1.5
2	8	42410	110	147	10.0ns	53.8ms	2.5
4	16	51901	244	272	11.1ns	41.8ms	3.2

We use Xilinx Vivado HLS 2014.1 as a back-end C-to-FPGA tool. LEAP supports Altera FPGA boards as well as several boards with Xilinx FPGAs (Nallatech ACP, XUPV5, HTG-V5, ML605, VC707). Recently, support has been added for Xilinx VC709 boards with two board-level DDR3 memory modules. Here, the target platform is a VC707 evaluation board (Virtex 7 FPGA, 1GB on-board 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 2013.09 and Xilinx ISE 14.5. The on-chip caches of the private and coherent scratchpads are direct-mapped with write-back policy and we set their size to 32 KBytes with 64 bit block size by default. We report FPGA slices, DSP48 slices, 36K Block RAMs (BRM), achieved clock period and total latency (cycle count $\times$ clock period) for the complete FPGA designs (HLS core and multi-scratchpad architecture). Table 1 quantifies the acceleration and resource consumption of parallelization and the multi-scratchpad architecture. N_c is the number of inserted SPs.

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

P	N_c	Slices	DSP	BRAM	Clock	Latency	Area-time product
Merger (4×2048 random input key-value pairs)							
4	4	31825 (38.6%)	8 (166.7%)	76 (5.6%)	11.1 ns (11.0%)	138.9 ms (20.5%)	4421.0 slices×s (67.0%)
Tree Deletion (2048 tree nodes)							
2	6	38296 (1.5%)	25 (31.6%)	132 (18.9%)	11.1 ns (11.0%)	4353.6 us (67.2%)	166.7 slices×s (69.6%)
4	12	52223 (24.8%)	25 (8.7%)	206 (2.0%)	11.8 ns (12.4%)	1311.1 us (84.4%)	68.5 slices×s (130.0%)
Filter (32767 kd-tree nodes, $K = 128$ clusters)							
2	8	45985 (8.4%)	108 (-1.8%)	169 (15.0%)	11.1 ns (11.0%)	84.7 ms (57.3%)	3894.1 slices×s (70.6%)
4	16	64330 (23.9%)	242 (-0.8%)	278 (2.2%)	11.8 ns (6.3%)	63.0 ms (50.7%)	4051.7 slices×s (86.8%)

For each benchmark, we set the unparallelized ($P = 1$) design with no caches as a reference (top row for each benchmark). The ratio S is the speed-up of each configuration compared to the reference case ($S = 1$).

As expected, the speed-up by parallelization is moderate (2.2× - 3.0×) if the memory interface is not supported by caches. Adding single caches to the unparallelized implementations brings a speed-up of 1.1× - 1.7×. We observe latency improvements when the applications are parallelized and multiple caches are inserted. The speed-up for Merger and Tree Deletion is significant in this case (10.9× and 9.2×) because the heap-allocated data, after partitioning by our analysis, fit almost entirely in the on-chip caches, as opposed to the tree data structure used by the filtering algorithm which achieves a moderate speed-up of 3.2× due to the sub-tree size and limited reuse of tree node data.

Our analysis determines that Merger requires P private SPs, while Filter and Tree Deletion require a hybrid architecture consisting of private and coherent SPs. We compare the implementation results of our application-specific architectures to an 'all-shared' scenario where no knowledge of disjoint heap regions is available to generate the multi-scratchpad 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 SPs must be supported with a coherency network by default. We focus on the second point 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 SPs. 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) SP architecture in Table 1 which uses knowledge of private and shared heap regions ($\frac{AC-HY}{HY}$ in %). The AC versions use more logic and have longer latencies. We compare the efficiency of the implementations by the area-time product. Our disjointness analysis brings an overall improvement of the slices-latency product of 67.0% to 130.0% (84.8% on average).

6. RELATED WORK

There are several approaches to optimizing the on-chip/off-chip memory hierarchy in an HLS context. Cheng *et al.* [15] use run-time profiling to group program operations accessing the same memory addresses into partitions and to instantiate separate on-chip caches assigned to disjoint memory regions accessed by the groups. This approach does not make any assumptions about the type of program to analyze. A disadvantage, however, is that it has to rely on representative test inputs and a simulation environment and requires a mechanism to take corner cases into account that have been missed during simulation. Similar to this work, the CHiMPS framework is a C-to-FPGA flow that generates a distributed multi-cache architecture [20]. A main difference to our work is that, sidestepping the coherence problem, shared memory regions are not supported by caches, while we address this issue with a sharing analysis and coherent caches. Another main difference is that independent memory regions must be manually indicated with source code annotations as opposed to an automated analysis in this work.

Significant advancements in the direction of 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 polyhedral model is applied to precisely analyze the accesses to static arrays referenced in such loop nests. For example, Liu *et al.* [16], and Bayliss and Constantinides [11] use the polyhedral model to determine the addresses of reused data items and buffer them in on-chip memories, whereas Pouchet *et al.* [19] present an on-chip buffer insertion in combination with automatic loop-level parallelization. The 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. The focus of our work on heap-allocated data-structures significantly increases the body of code for which automated parallelization and automatic memory-system optimizations can be applied.

7. 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 applications. We target applications that perform computation on large heap-allocated data structures and that require access to an off-chip memory. We leverage a separation logic-based static program analysis in [7] 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 paral-

lelize the application. Our extended analyzer also detects heap regions that are shared by multiple accessors in the parallelized implementation. An additional commutativity 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-scratchpad architecture 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.

In our experiments with three heap-manipulating C++ benchmark applications, we observe a speed-up of up to 10.9× after parallelization and generation of a multi-scratchpads architecture compared to the unparallelized application and uncached access to the off-chip memory. We also quantify the benefit of extracting application-specific knowledge about disjoint and shared heap memory regions: Our hybrid multi-scratchpads architecture consisting of private and coherent scratchpads outperforms a default all-coherent version by 84.8% on average in terms of the area-time product.

Future work will extend our static program analysis to detect data reuse during program execution. The outcome of this analysis is a binary decision whether to insert a cache. Beyond data reuse detection, we plan to integrate a prediction of memory access patterns into our analysis framework and to use this information to approximate the cache hit probability as another application-specific feature taken into account to optimize the cache architecture. Knowledge about access patterns will also be used to implement application-specific prefetching and request merging. We also plan to implement a quantification of the average amount of heap consumption to generate cache sizing information.

8. REFERENCES

[1] High-Level Synthesis with LegUp. [Online]. Available: `http://legup.eecg.utoronto.ca/`

[2] ROCCC 2.0. [Online]. Available: `http://www.jacquardcomputing.com/roccc/`

[3] ROSE compiler infrastructure. [Online]. Available: `http://rosecompiler.org/`

[4] Z3: An Efficient SMT Solver. [Online]. Available: `http://z3.codeplex.com/`

[5] F. Winterstein, S. Bayliss, and G. Constantinides. FPGA-Based K-Means Clustering using Tree-Based Data Structures. In *Proc. Field Programmable Logic and Appl.*, pages 362–365, 2013.

[6] F. Winterstein, S. Bayliss and G.A. Constantinides. High-level synthesis of dynamic data structures: A case study using Vivado HLS. In *Proc. Field-Programmable Technology*, pages 362–365, 2013.

[7] F. Winterstein, S. Bayliss and G.A. Constantinides. Separation logic-assisted code transformations for efficient high-level synthesis. In *Proc. Field-Program. Cust. Comput. Machines*, pages 1–8, 2014.

[8] K. Fleming, H.-J. Yang, M. Adler and J. Emer. The leap fpga operating system. In *Proc. Field Programmable Logic and Appl.*, pages 1–8, 2014.

[9] M. Adler, K. Fleming, A. Parahsar, M. Pellauer and J. Emer. Leap scratchpads: automatic memory and cache management for reconfigurable logic. In *Proc. Field Program. Gate Arrays*, pages 25–28, 2011

[10] H.-J. Yang, K. Fleming, M. Adler and J. Emer. LEAP shared memories: automating the construction of fpga coherent memories. In *Proc. Field-Programmable Custom Comput. Machines*, pages 117–124, 2014.

[11] S. Bayliss and G. Constantinides. Optimizing sdram bandwidth for custom fpga loop accelerators. In *Proc. Field Programmable Gate Arrays*, pages 195–204, 2012.

[12] BDTI. An independent evaluation of: the autoesl autopilot high-level synthesis tool, 2010.

[13] C. Calcagno and D. Distefano. Infer: an automatic program verifier for memory safety of C programs. In *Proc. NASA Formal Methods*, pages 459–465, 2011.

[14] A. Charlesworth. The undecidability of associativity and commutativity analysis. *ACM Trans. on Program. Lang. and Systems*, 24(5):554–565, Sept. 2002.

[15] S. Cheng, M. Lin, H. J. Liu, S. Scott, and J. Wawrzynek. Exploiting memory-level parallelism in reconfigurable accelerators. In *Proc. Field-Program. Cust. Comput. Machines*, pages 157–160, 2012.

[16] 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 Trans. on Computer-Aided Design of Integrated Circuits and Systems*, 28(3):305–315, Mar. 2009.

[17] 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*, pages 1 – 21, Aug. 2012.

[18] P. O'Hearn, J. Reynolds, and H. Yang. Local reasoning about programs that alter data structures. In *Proc. Computer Science Logic*, pages 1–19, 2001.

[19] L.-N. Pouchet, P. Zhang, P. Sadayappan, and J. Cong. Polyhedral-based data reuse optimization for configurable computing. In *Proc. Field Programmable Gate Arrays*, pages 29–38, 2013.

[20] 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*, 37(3):395, June 2009.

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

[22] L.J. Hendren, A. Nicolau. Parallelizing Programs with Recursive Data Structures. In *IEEE Trans. Parallel Distrib. Syst.*, 1(1):35–47, Jan. 1990.

[23] S. Magill, A. Nanevski, E. Clarke, and P. Lee. Inferring invariants in separation logic for imperative list-processing programs. In *SPACE*, 2006.

[24] B. Guo, N. Vachharajani, and D. I. August. Shape analysis with inductive recursion synthesis. In *ACM SIGPLAN Notices*, 42(6):256–265, Jun. 2007.

[25] M. C. Rinard and P. C. Diniz. Commutativity analysis: a new analysis technique for parallelizing compilers. *ACM Transactions on Programming Languages and Systems*, 19(6):942–991, Nov. 1997.