Separation Logic for High-Level Synthesis

Felix J. Winterstein* Imperial College London

Samuel R. Bayliss† Imperial College London

George A. Constantinides‡ Imperial College London

Abstract

High-level synthesis (HLS) promises a significant shortening of the FPGA design cycle by raising the abstraction level of the design entry to high-level languages such as C/C++. However, applications using dynamic, pointer-based data structures and dynamic memory allocation remain difficult to implement well, yet such constructs are widely used in software. Automated optimizations that leverage the memory bandwidth of FPGAs by distributing the application data over separate banks of on-chip memory are often ineffective in the presence of dynamic data structures, due to the lack of an automated analysis of pointer-based memory accesses. In this work, we take a step towards closing this gap. We present a static analysis for pointer-manipulating programs which automatically splits heap-allocated data structures into disjoint, independent regions. The analysis leverages recent advances in separation logic, a theoretical framework for reasoning about heap-allocated data which has been successfully applied in recent software verification tools. Our algorithm focuses on dynamic data structures accessed in loops and is accompanied by automated source-to-source transformations which enable automatic loop parallelization and memory partitioning by off-the-shelf HLS tools. We demonstrate the successful loop parallelization and memory partitioning by our tool flow using three real-life applications which build, traverse, update and dispose dynamically allocated data structures. Our case studies, comparing the automatically parallelized to the direct HLS implementations, show an average latency reduction by a factor of 2× *across our benchmarks.*

I. Introduction

High-level synthesis (HLS) and a C/C++ design entry can significantly shorten the design cycle of field-programmable gate array (FPGA) implementations when compared to specifications based on register transfer level (RTL). Examples of state-of-the-art HLS tools are Xilinx Vivado HLS, ROCCC [Villarreal et al.2010] and LegUp [Canis et al.2011], and recent evaluations show that these can deliver a quality of results (QoR), measured in terms of latency and resource utilization, close to hand-written RTL implementations [Meeus et al.2012, BDTI2010]. A crucial task is the extraction of parallelism from a sequential program description while preserving the program semantics, which is usually based on a dependence analysis. Additionally, parallelization requires the memory system to match the computational parallelism. The distributed memory architecture in FPGAs provides impressive memory bandwidth if the program data is partitioned and distributed over multiple on-chip memory blocks. Automatic parallelization for C-to-FPGA compilers therefore requires a memory access and dependence analysis so as to determine data or control dependencies between program fragments. The objective in this work is to implement a static program analysis

*This work is sponsored by the European Space Agency under the Networking/Partnering Agreement No. 4000106443/12/D/JR and by the EPSRC under grant numbers EP/I020357 and EP/I012036.

F. Winterstein, Department of Electrical and Electronic Engineering, Imperial College London, (Current address) Directorate of Human Spaceflight and Operations, European Space Operations Centre, Robert-Bosch-Str. 5, 64293 Darmstadt, Germany,; email f.winterstein12@imperial.ac.uk;

†S. Bayliss, Department of Electrical and Electronic Engineering, Imperial College London, South Kensington Campus, London SW7 2BT, United Kingdom; email: s.bayliss08@imperial.ac.uk.

‡G. Constantinides, Department of Electrical and Electronic Engineering, Imperial College London, South Kensington Campus, London SW7 2BT, United Kingdom; email: g.constantinides@imperial.ac.uk.

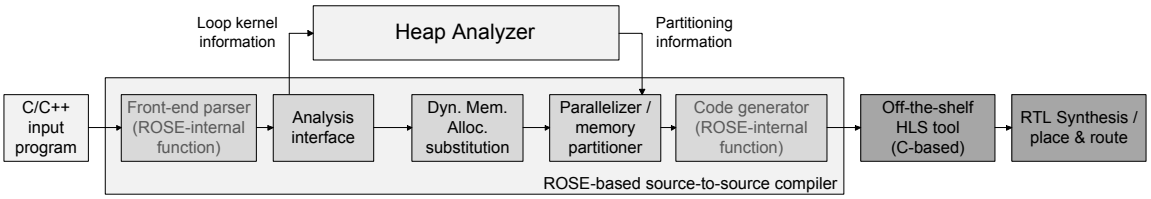


Figure 1: *High-level compilation tool flow.*

and automated code transformations that enable automatic parallelization and distribution of data over separate blocks of on-chip memory.

Our program analysis and code transformations explicitly target programs that use pointers to *heap*-allocated data and dynamic memory allocation, a powerful and widely used feature of high-level programming languages such as C/C++. Automated program transformations which break the monolithic heap memory space into several portions (*heaplets*) and parallelize pointer-manipulating programs are beyond the scope of most current HLS techniques. This gap is mainly due to the difficulty of disambiguating pointer aliases. In [Winterstein et al.2014], we make a step towards closing this gap and present a static analysis for pointer-manipulating programs which determines dependencies between loop iterations accessing heap memory and splits dynamic data structures into disjoint, independent regions. The dependence/disjointness analysis enables automated source-to-source transformations for parallelization and data distribution which can be exploited by a back-end HLS tool. Our source-to-source compiler is based on the ROSE compiler infrastructure [LLNL2014] as shown in Fig. 1. The main contribution of this work is the heap analyzer in Fig. 1. Our departure point from previous work is the use of recent advances in *separation logic* [O'Hearn et al.2001] which allows a formal description of the program state and reasoning about the resources accessed by a program. Separation logic extends classical logic by an operator that explicitly expresses the separation of resources, *i.e.* the non-aliasing property of two pointers. This paves the way for an automated program analysis and can straightforwardly handle dynamic memory allocation in disjoint heaps. This paper is an extended version of the work published in [Winterstein et al.2014]. Our contributions in [Winterstein et al.2014] are:

◦ A separation logic-based parallelization algorithm for pointer-manipulating programs which access dynamic data structures. Our static program analysis handles straight-line code as well as arbitrary *while*-loops and determines whether there is communication-free parallelism in the loop with respect to the accessed dynamic data structures. Starting from the C memory model of a global monolithic heap memory, is determines how to partition the heap and dynamic data structures into disjoint partitions that can be implemented in separate on-chip memory blocks (Section V).

◦ The implementation of an automated source-to-source transformation infrastructure: The source translator ensures synthesizability of code containing unsupported constructs related to dynamic memory allocation (an unsupported feature in tools such as LegUp or Vivado HLS). In a second pass, the disjointness information provided by our analysis is used to split the synthesized heap memory into separate blocks and to split a loop into multiple loops so as to obtain a semantically equivalent parallel implementation. The property of communication-free parallelism ensures that each functional unit only requires access to its own private memory block (Section VI).

◦ The demonstration of our tool flow using three real-life applications as test cases which build,

traverse, update and dispose dynamically allocated data structures. The transformations at source code level allow us to stay as independent as possible of a specific HLS tool. We use Xilinx Vivado HLS as an exemplary back-end tool in our case studies. We also include hand-written HLS and RTL implementations for comparison (Section VII).

This paper extends the previous work in the following ways:

◦ We describe our heap analysis algorithm in more depth. We explain the details of our fix-point calculation and abstraction steps which are central to our technique and which allow us to statically analyze loops whose number of iterations are unknown to the analysis at compile time. We also describe in more detail how the analysis is linked to the source code transformation (Section V). In addition, we give a more detailed introduction to the theoretical background (Section IV). In particular, we extend it to theorem proving in separation logic, a core component of our technique.

◦ We demonstrate the applicability of our technique with additional benchmark applications. These applications use additional types of tree data structures and dynamically construct trees in addition to tree traversals as presented previously. The new applications also show that partitioning can be decoupled from parallelization (Section VII).

◦ We elaborate an execution time analysis of our static heap analyzer and discuss how certain code constructs, such as nested loops, affect the tool running time (Section VII).

II. Related Work

Besides the basic HLS steps, scheduling (the assignment of program operations to time slots), resource allocation and binding (assignment of hardware components to operations), and the generation of control circuits, an HLS tool usually performs several transformations of the input code. Many recent C-to-RTL flows build on standard compiler frameworks such as the LLVM framework [Lattner and Adve2004] (*e.g.* Vivado HLS, ROCCC and LegUp) or GCC (*e.g.* GAUT). The input code passes through standard compiler optimizations, for example dead-code elimination, constant propagation, loop unrolling, and other `-O3` level optimizations, before hardware synthesis. The effect of standard LLVM optimizations on the QoR is explored in [Huang et al.2013], where a 16% average improvement is reported. In contrast, this paper describes an advanced program analysis and HLS-specific code optimizations beyond standard compiler optimizations.

Optimizations based on the *polyhedral model* [Feautrier1991] are among the most popular advanced compiler techniques that have made their way into HLS CAD flows to date. The polyhedral model, an algebraic representation of the loop iteration space, is applied to precisely analyze memory accesses and to determine data dependencies between iterations of loop nests with references to static arrays. Liu *et al.* [Liu et al.2007] have pioneered the use of the polyhedral model for inserting on-chip reuse buffers into the interface of an FPGA accelerator to an external memory. These reuse buffers hold data which are accessed by the loop kernel multiple times. The polyhedral model is used to determine data reuse opportunities and to calculate the reuse volume at compile time. Cong *et al.* [Cong et al.2011] implement bandwidth optimizations though memory partitioning based on a dependence analysis using an integer linear programming (ILP) formulation over the polyhedral model. Bondhugula *et. al.* [Bondhugula et al.2008] describe a scalable ILP-based technique for the aggregation of sets of loop iterations into tiles so as to maximize loop-level parallelism and data locality. Their technique is implemented in a source-to-source translator targeting code optimizations for FPGA-directed HLS [Pouchet et al.2013].

The polyhedral model is applicable to loop nests with static control structures and in which memory access functions and loop bounds are affine combinations of the enclosing loop variables and parameters. The model, however, cannot be directly applied to indirect array references or pointer accesses. Benabderrahmane *et al.* [Benabderrahmane et al.2010] fit the model to indirect array accesses and pointers by conservatively assuming a dependency between all program statements accessing the array or the heap, respectively. In addition, dynamic memory allocation, a widely used feature of high-level programming languages, cannot be captured. In contrast to this, our work targets the same optimizations, automated loop parallelization and the distribution of data over separate memory partitions, but it builds on a logic-based program analysis that explicitly targets pointer-manipulating programs making this work a complement to existing work based on the polyhedral model.

While third generation HLS tools, such as Vivado HLS, LegUp and ROCCC, avoid the issue of synthesizing heap-directed pointers into hardware by excluding features such as dynamic memory allocation, there is existing work for second generation HLS flows. Séméria *et al.* [Séméria et al.2000] present an approach for mapping C code with pointers and *malloc* / *free* operations into hardware. Similar to this work they instantiate on-chip allocator blocks using standard allocation schemes and use a pointer analysis to safely map the monolithic heap space to distributed on-chip memory banks. Their approach is based on a pointer analysis by Wilson and Lam [Wilson and Lam1995] that uses a summary of different aliasing cases of the pointer arguments passed to a procedure to identify pointer-induced data dependencies. However, the need for explicit assertions summarizing the aliasing properties of several pointers quickly renders the program analysis unwieldy. Separation logic solves this problem by including a new operator as we discuss in Section IV. Another substantial difference to our approach is their approximate description of data structures (*location sets* [Wilson and Lam1995]), whereas our analysis precisely describes the shape of the heap layout. The approach to synthesis of pointer-based C code programs by Babb *et al.* [Babb et al.1999] also uses an analysis based on location sets. In contrast to both, our approach allows us to partition recursive data structures to increase parallelism.

The work by Ghiya and Hendren [Ghiya et al.1998], in line with this work, uses a shape analysis of the heap layout to establish disjointness of heap-allocated recursive data structures for parallelizing software compilers. This information is used to parallelize loops traversing these data structures, which is similar to our objective. A difference to our work is their analysis which classifies data structures into trees, lists, and general graphs and looks up the known aliasing properties of the link fields. A separation logic-based analysis 'produces' this information itself. Another major difference, of course, is that our work targets HLS CAD flows for hardware synthesis which allows us to build a customized distributed memory architecture based on the heap access analysis.

Formal software verification has been the main application of separation logic [O'Hearn et al.2001]. Only recently, its scope has been extended to data dependence analyses for automatic parallelization. We build on the work by Raza *et al.* [Raza et al.2009] which describes such an analysis and provides the theoretical foundation for our tool as described in Section IV. We modify and extend their method by allowing the analysis to perform semantics-preserving modifications to the program state until the partitioning goal can be proven. We also modify their reasoning in that we present analysis tailored to loop parallelization and the inference of loop-invariant state descriptions which is not covered in [Raza et al.2009]. The work in [Cook et al.2010] also takes Raza's method into an HLS context. The parallelization transformations, however, are not automated and memory partitioning is not addressed. Furthermore, determining disjointness in our tree-based benchmarks

```
1 //main kernel function
2 void filter(treeNode *root) {
3   centerSet* c0 = new centerSet;
4   *c0 = ...;
5   stackRecord *s = push(root, c0, true, NULL);
6   while (s != NULL) {
7     treeNode *u; centerSet *c; bool d;
8     s = pop(&u, &c, &d, s);
9     centerSet cs = *c;
10    if (d) {
11      delete c;
12    }
13    centerSet *cnew = new centerSet;
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    }
21    delete u;
22  }
23 }
24
25 //auxiliary function push (create new list entry at head)
26 stackRecord* push(treeNode *u, centerSet *c, bool d, stackRecord *s){
27   stackRecord *t = new stackRecord;
28   t->u=u; t->c=c; t->d=d; t->n=s;
29   return t;
30 }
31 //auxiliary function pop (delete list head)
32 stackRecord* pop(treeNode **u, centerSet **c, bool *d, stackRecord *s){
33   *u=s->u; *c=s->c; *d=s->d; stackRecord *t=s->n;
34   delete s; return t;
35 }
```

Listing 1: *C-like pseudo code of the (modified) main kernel of the filtering algorithm.*

requires successive unrollings of loop iterations before disjointness can be established, which is not implemented in their technique. Finally, recent work by Botinčan *et al.* [Botinčan et al.2013] describes a technique for separation logic-based parallelization of software threads. Their work is interesting in that they automatically insert synchronization to preserve dependencies in addition to a dependence analysis. Their technique, however, focuses on the theoretical framework whereas we use the theoretical foundations in a demonstrably practical implementation.

III. Motivating Example

Our running example, which we use throughout to illustrate the problem and our approach to solve it, is taken from a high-performance implementation of a K-means clustering algorithm, a technique commonly used in machine learning, radar tracking, image or spectrum quantization applications. K-means clustering aims to partition a set $X = \{x_1, ..., x_N\}$ of points into K clusters, such that each point belongs to the cluster with the nearest mean (represented by its geometrical center). The algorithm considered here, referred to as the *filtering algorithm* [Kanungo et al.2002], uses a tree data structure (a 'kd-tree', [Kanungo et al.2002]) to prune unfavorable candidates for the nearest center to a given data point early in the search process. The tree-based pruning approach allows the algorithm to compute the clustering result significantly faster than other (brute-force)

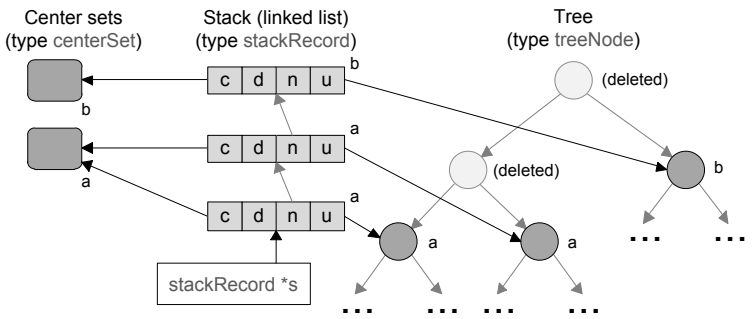


Figure 2: *Snapshot of the pointer-linked data structures accessed by the loop in Listing 1.*

```
1 // main kernel function
2 void filter(treeNode *root) {
3   // ... preamble (pointers access heap partitions a and b)
4   while (s_a != NULL) {
5     // ... loop body (pointers access heap partition a only)
6   }
7   while (s_b != NULL) {
8     // ... loop body (pointers access heap partition b only)
9   }
10 }
```

Listing 2: *Transformed program from Listing 1 (two parallel loop kernels).*

clustering implementations. Besides tree nodes, the algorithm propagates intermediate results (sets of candidate centers) through the call graph. Listing 1 shows C-like pseudo code of the main kernel of the iterative filtering algorithm, the only difference from [Winterstein et al.2013b] being that the tree traversal here is destructive. Fig. 2 shows the three heap-allocated data structures accessed by the loop: the tree, the center sets, and the stack. The stack is implemented as a pointer-linked list whose head is modified by 'push' and 'pop' operations. The stack contains pointers to the tree nodes and center sets. In line 8, pointers to a center set and tree node are fetched from the stack, and pointers to left and right child node as well as a newly allocated center set (line 13) are pushed onto the stack at the end of the loop body (lines 16 - 17) - preceded by a data-dependent conditional (line 15). The kd-tree is traversed in a pre-order fashion and visited nodes are deleted (line 21).

The static program analysis presented in Section V aims to determine the heap-carried data dependencies between loop iterations. Assuming that Fig. 2 describes the current state of the program, we can apply the following program transformations: 1) The remaining tree data structure (dark gray nodes) can be split into two sub-structures (two sub-trees labeled with *a*, one sub-tree labeled with *b*). 2) The linked list can be split into the uppermost node (pointing into the right sub-tree) and the nodes below (pointing into the left sub-tree). The same partitioning is applicable for the pool of center sets. 3) The loop can be split into two loop kernels, each accessing one sub-tree, list segment and group of center sets. The pointers dereferenced in any iteration of a loop will never access the data structures used by the other loop. Hence, once we have established that the loops are 'communication free' with respect to each other, we can split the heap memories into two banks of on-chip memory, each assigned to one loop as shown in Listing 2. A standard HLS tool can use the independence information to instantiate parallel hardware blocks for the loops without the need for arbitration of accesses to a global memory.

The difficult part of the above optimization is the program analysis: Regardless of scope, ev-

ery two heap-directed pointers could potentially alias, *i.e.* reference the same memory cell, which leads to dependencies between expressions that are syntactically unrelated. The difficulty of analyzing these programs increases with linked data structures which contain pointers in their link fields. Separation logic addresses exactly this issue and provides a formalism for straightforwardly expressing the heap layout and alias information at each point of the program execution as described in the next sections.

IV. Background

We briefly describe the underlying theory. A formal introduction is beyond the scope of the paper, but is given in [O'Hearn et al.2001]. The objective of our analysis is to identify disjoint regions in the heap memory that are accessed by different fragments of the program code so as to declare these code fragments as independent (given that no other dependencies exist). In our static analysis, we describe the layout of the heap with a formula at each point of program execution: Informally, it steps through the source code and maintains a formula describing the heap-allocated data structures as well as all points-to information at each program statement. While stepping (symbolically) from one statement to the next, the formula is modified reflecting the heap manipulation, for example a statement may allocate new data, dispose data, or change the data content. The formula maintains information about the layout of the data structure and ignores other properties such as their size. Thus, we refer to this type of analysis as *shape analysis*. Separation logic allows us to express the heap layout in concise formulae and to identify precisely what program statement accessed what part of the formula. The following sub-sections describe the required components of this analysis: The syntax of separation logic formulae (Section I), the formal specification program statements (Section II), symbolically stepping through the source code (Section III), and theorem proving in separation logic (Section IV), which informs us about the 'accessed' portion of the formula.

I. Modelling Program State in Separation Logic

A program modifies the values of program variables and the content of memory cells during execution. The assignment of values to variables and memory cells is referred to as *program state*. Separation logic formally describes the state with two components: The *store* describes the values assigned to variables (*e.g.* $x = 3$ means that variable x currently holds the value 3) and the *heap* describes the values assigned to addressable memory locations (*e.g.* $y \mapsto 4$ means that pointer variable y points to a memory cell containing the value 4). Note that $y \mapsto 4$ implies that the memory location at y is allocated. A program may start with an empty heap memory where nothing is allocated, which is denoted by the *emp* keyword in separation logic formulae. In addition to program variables, the formulae may use auxiliary *primed* variables which only exist in formulae, not in the program code. For example, $z'_1 = 4 \wedge y \mapsto z'_1$ means that there is some heap cell z'_1, containing the value 4 and y points to that cell here, where '$\wedge$' is the classical 'and'-conjunction.

Pointer variables can have a special value `nil` that corresponds to the `NULL` expression in C/C++. In addition to describing that a memory cell holds a scalar value, we can also use records (*structs* in C/C++): $y \mapsto [\mathtt{f}_1 : x'_1, .., \mathtt{f}_n : x'_n]$ means that y points to a heap-allocated record containing fields with $x'_1, ..., x'_n$ as content. $\mathtt{f}_1, ..., \mathtt{f}_n$ are the field names. For example, the head of the stack in Fig. 2 is described by the formula $s \mapsto [\mathtt{c} : c'_1, \mathtt{d} : d'_1, \mathtt{n} : n'_1, \mathtt{u} : u'_1]$.

Separation logic formulae are generally of the form $\Pi \wedge \Sigma$, where Π is the *pure* part describing the store (*e.g.* $x = 3$) and Σ is the *spatial* part describing the heap (*e.g.* $y \mapsto 4$). We define `Val` the set of values, `Var` the set of program variables, and `Var'` the set of auxiliary primed variables. Def. 1

defines the baseline syntax of the formulae used in our analysis.

Definition 1 (Baseline syntax of separation logic formulae)

$$\begin{array}{rll} E,F ::= & v \in \texttt{Val} \mid x \in \texttt{Var} \mid x'_i \in \texttt{Var'} & \text{expressions} \\ \Pi ::= & \mathit{true} \mid E = F \mid E \neq F \mid \Pi \wedge \Pi & \text{pure formulae} \\ \Sigma ::= & E \mapsto [\mathtt{f}_1 : x'_1, .., \mathtt{f}_n : x'_n] \mid \mathit{emp} \mid \Sigma * \Sigma & \text{spatial formulae} \end{array}$$

Pure formulae contain (in)equalities and the classical conjunction ($\wedge$). Spatial formulae express the following:

- $E \mapsto [\mathtt{f}_1 : x'_1, .., \mathtt{f}_n : x'_n]$ describes a heap-allocated record as discussed above. We use the abbreviation $E \mapsto _$ to denote that E points to 'some' record.
- *emp* denotes an empty heap where nothing is allocated.
- The separating conjunction ($*$) is the core element of separation logic: The formula $\Sigma_0 * \Sigma_1$ means that the heap is split into two disjoint portions h_0 and h_1, where Σ_0 holds for h_0 and Σ_1 holds for h_1. Disjoint heap portions are referred to as *heaplets*. The $*$-connective embeds the non-aliasing property of pointers, *i.e.* $E \mapsto [\mathtt{f} : x'_1] * F \mapsto [\mathtt{f} : y'_1]$ implies $E \neq F$ by definition. Hence, the content of the first heaplet can be modified by a program without any side effects for the second one. The usefulness of the separating conjunction becomes obvious when considering the counterexample in classical logic, $E \mapsto [\mathtt{f} : x'_1] \wedge F \mapsto [\mathtt{f} : y'_1]$: E and F may or may not alias, and expressing the non-aliasing property requires adding the constraint $E \neq F$ to the formula. These constraints are required for each pair of pointers in the program and quickly render an automated analysis unwieldy, especially in the case of pointer-linked data structures.

We refer to 'formula' as 'predicate' in the following. Def. 1 allows us to describe single, heap-allocated data records. To describe more sophisticated data structures such as linked lists or trees, we need to build additional predicates using the $*$-connective. A naive approach of describing a linked list is to mention all nodes in the list: $E \mapsto [\mathtt{n} : x'_1] * x'_1 \mapsto [\mathtt{n} : x'_2] * ... * x'_m \mapsto [\mathtt{n} : \mathtt{nil}]$. This, however, is problematic as the length m of a dynamically allocated linked list is usually unknown at compile time. Instead, we use recursive predicates that describe data structures without knowing their size:

Definition 2 (Example: List segment)

$$ls(E,F) \Leftrightarrow (E = F \wedge \mathit{emp}) \vee (E \neq F \wedge E \mapsto [\mathtt{n} : x'_1] * ls(x'_1, F)\ ) \tag{1}$$

i.e. there is a list segment between pointer E and F if and only if the following condition holds. If $E = F$ this heap portion is empty. Otherwise E points to an element which, in turn, points to a list segment between itself and F.

Definition 3 (Example: Tree)

$$tree(E) \Leftrightarrow (E = \mathtt{nil} \wedge \mathit{emp}) \vee (E \mapsto [\mathtt{l} : x'_1, \mathtt{r} : y'_1] * tree(x'_1) * tree(y'_1)\ ) \tag{2}$$

i.e. there is a tree pointed to by E if and only if the following condition holds. If $E \neq \mathtt{nil}$ it points to an element which contains pointers to left and right sub-tree.

Definition 4 (Example: List with pointers to other heaplets)

$$\begin{aligned} pls(E,F) \Leftrightarrow (\ & E = F \wedge \mathit{emp}\ ) \vee \\ & (\ E \neq F \wedge E \mapsto [\mathtt{u} : u'_1, \mathtt{c} : c'_1, \mathtt{n} : n'_1] * tree(u'_1) * c'_1 \mapsto _ * pls(n'_1, F)\ ) \end{aligned} \tag{3}$$

i.e. there is a list segment as in (1) whose elements also point to a tree and a heap-allocated record.

Note that we omitted additional data fields in the records above for ease of illustration. The above examples demonstrate the ability to describe common data structures; automatic inference of such definitions has been demonstrated by Guo *et al.* in [Guo et al.2007].

II. Programming Language

The next step is to define how program state, expressed in separation logic formulae, is modified during program execution. For didactic purposes, we consider a simple programming language with heap manipulating commands and loops:

Definition 5 (Programming language)

$b ::=$	$E = F \mid E \neq F$	boolean expressions
$A ::=$	$x := E \mid x := [E].f \mid [E].f := F \mid \text{new(x)} \mid \text{delete(E)}$	atomic commands
$C ::=$	$A \mid \textbf{if}\ b\ C_1\ C_2 \mid \textbf{while}\ b\ C \mid C_1; C_2$	commands

E and F are arbitrary expressions containing program variables and values (e.g. $E ::= x$, $E ::= \texttt{nil}$, or $E ::= y + 1$). The term $[E].f$ denotes pointer dereferencing of E and accessing field f of the heap-allocated record pointed to by E.

The program statements (commands) modify the state. The transition of state upon execution of a command is specified by the triple $\{P\}C\{Q\}$: P is the formula describing the pre-condition the state must satisfy for the command to run. If C runs and halts then the post-condition formula Q for the program state is true after execution [O'Hearn et al.2001]. For example, if C is a command that writes the value 5 to the memory cell referenced by y this heap cell must be allocated (pre-condition) and must contain 5 after successful command execution (post-condition): $\{y \mapsto [f : x_1']\}\ [y].f := 5\ \{y \mapsto [f : 5]\}$. Def. 6 specifies a triple for each atomic command of our programming language:

Definition 6 (Specifications for atomic commands [Raza et al.2009])

$$
\begin{array}{rll}
\{\ x = y_1'\ \} & x := E & \{\ x = E[y_1'/x]\ \} \\
\{\ E \mapsto [\mathtt{f} : y_1']\ \} & [E].\mathtt{f} := F & \{\ E \mapsto [\mathtt{f} : F]\ \} \\
\{\ x = y_1' \wedge E \mapsto [\mathtt{f} : z_1']\ \} & x := [E].\mathtt{f} & \{\ x = z_1' \wedge E[y_1'/x] \mapsto [\mathtt{f} : z_1']\ \} \\
\{\ emp\ \} & new(x) & \{\ x \mapsto z_1'\ \} \\
\{\ E \mapsto y'\ \} & delete(E) & \{\ emp\ \}
\end{array}
$$

The term $E[y_1'/x]$ denotes expression E with all occurrences of x replaced by y_1'. Note that specifying pointer-manipulating commands in this way is only possible thanks to separation logic's 'frame rule'. A detailed explanation is given in [O'Hearn et al.2001].

III. Symbolic Execution of Programs

Our static analysis 'symbolically' executes the program by propagating the program state, expressed in separation logic formulae, from one program statement to the next, thereby updating it using the specifications for single commands in Def. 6. The symbolic execution propagates the state through all control flow paths of the program (branching and loops create multiple control flow paths). We build our automated analysis on *coreStar* [Botinčan et al.2011], an open source tool for separation logic-based symbolic execution and theorem proving. At each node in the control flow graph (CFG), coreStar determines the part of the formula describing the current state which matches the pre-condition of the current program statement, and replaces that part with the post-condition

in Def. 6. The other parts, F, of the state formula remain untouched. Formally, before executing the program statement C, it breaks the current program state $\Pi_1 \wedge \Sigma$ into $\Pi_1 \wedge P * F$, where P is the pre-condition of C and F is called the frame. The symbolic execution of C then updates the program state to $\Pi_2 \wedge Q * F$ by replacing P by Q and leaving the frame F untouched. Note that, in a 'correct' program, the symbolic execution always finds a suitable P, whereas failure to do so allows a software verification tool (*e.g.* [Calcagno and Distefano2011]) to find a pointer-related bug. Here, we use separation logic for proving parallelizability instead of correctness, but, as a side effect, our tool also reports a failure in this case.

We have modified coreStar in order to include an extension of the standard symbolic execution called *labelled symbolic execution* [Raza et al.2009] which assigns a unique label to Q, the spatial part of the state formula that was modified, *i.e.* $\Pi_2 \wedge \Sigma \equiv \Pi_2 \wedge \langle Q \rangle_{\{l \in \mathtt{Lab}\}} * F$, with `Lab` being the set of all labels. In the original work in [Raza et al.2009], each program statement C_i is assigned a unique label $l_i \in \mathtt{Lab}$. The technique thus propagates the 'heap footprint' of each statement through the CFG. This tracks the memory accesses made by different parts of the program, a prerequisite for detecting heap-carried dependencies. Our heap access analysis described in the next section is a modified version of labeled symbolic execution in order to detect the presence of communication-free parallelism in loops.

IV. Theorem Proving

Automated theorem proving is the work horse in our tool flow. The symbolic execution engine uses it to infer the frame portion F at each CFG node as described above. A detailed description of frame inference is beyond the scope of this paper, but is given in [Berdine et al.2005]. It is also used to prove implications described in the next sections. In all cases, the theorem prover tries to verify an *entailment* of the form $S_1 \vdash S_2$ which is interpreted as "S_1 entails S_2" or "from S_1 I can derive S_2", with S_1 and S_2 being formulae in separation logic of the form $\Pi \wedge \Sigma$. The theorem prover in coreStar builds on the proof technique in [Berdine et al.2005]. The basic idea is to reduce an entailment $S_1 \vdash S_2$ to an axiom $\Pi \wedge \mathit{emp} \vdash \mathtt{true} \wedge \mathit{emp}$, with an arbitrary pure formula Π. The proof of the original entailment is successful if the reduction is successful. The latter is made by applying a sequence of inference rules of the form:

$$\frac{\text{premise}}{\text{conclusion}}.$$

An inference rule asserts that "if the premise holds then the conclusion holds". During the proof search, the theorem prover applies its inference rules upwards, i.e. the premise of the previous rule application becomes the conclusion of the current rule application until the axiom is reached or a contradiction is found. Inference rules rewrite the separation logic formulae. For example, we can inform the prover that the following entailment is valid:

$$x \mapsto [\mathtt{n} : x_1'] * ls(x_1', \mathtt{nil}) \vdash ls(x, \mathtt{nil}) \tag{4}$$

(i.e. if x points to the first element in a linked list, then x itself points to a linked list).
To this end, we must provide two inference rules:

$$\frac{ls(E,F) \vdash ls(E,F)}{E \mapsto [\mathtt{n} : x_1'] * ls(x_1', F) \vdash ls(E,F)} \quad \frac{Q_1 \vdash Q_2}{Q_1 * S \vdash Q_2 * S} \tag{5}$$

The first rule is an 'abstraction rule' that says that a singleton head node of a linked list can be folded into an inductive *ls* predicate from Def. 2. The second is a 'subtraction rule' that removes

identical heaplets on both sides of the entailment. Given these rules, the theorem prover will derive

$$\dfrac{\dfrac{emp \vdash emp}{ls(x,\texttt{nil}) \vdash ls(x,\texttt{nil})}\text{subtraction}}{x \mapsto [\texttt{n} : x_1'] * ls(x_1',\texttt{nil}) \vdash ls(x,\texttt{nil})}\text{abstraction} \tag{6}$$

Starting from the initial state $E \mapsto [\texttt{n} : x_1'] * ls(x_1', F) \vdash ls(E, F)$ in the bottom row, (6) shows the application of both inference rules in (5) from bottom to top. The top row is equivalent to $\texttt{true} \wedge emp \vdash \texttt{true} \wedge emp$ which is an axiom. Hence, (6) tells us that (4) can be derived from an axiom and therefore is a valid entailment. The example we give in (4) is a logical implication: The left hand side implies the right hand side.

V. Partitioning and Parallelization

Our semantics-preserving parallelization is based on the rationale that two program fragments can run in parallel if they access disjoint regions in memory (global variables being a special case of memory resources). We can then place each of these regions in physically separated on-chip memory banks without the need for cross-communication between functional units and each bank. Our memory partitioning and parallelization analysis is hypothesis-based: The user specifies a value P. This value corresponds to the hypothesis that the heap accessed by the loop kernel can be split into P disjoint parts and the loop can be split into P parallel loops. The algorithm then tries to verify the hypothesis.

Proving the hypothesis is implemented in two main phases: searching for a necessary condition for the hypothesis to be true and, starting from the program state satisfying this condition, proving that the hypothesis is valid in all iterations. In the first phase, our tool symbolically executes the loop preamble and a finite number of loop iterations. During this process, it examines the separation logic formulae describing the accessed heap to determine whether the heap can be split into P parts of identical shape, which is our necessary condition for partitioning. If such an initial partitioning can be established the tool instruments the formulae with *cut-points* (markers) that mark the beginning of each partition. After the initial partitioning and instrumentation, the second phase is to prove that this partitioning is maintained not only in a finite number of iterations at loop start-up but in all loop iterations. Maintaining the partitioning in this case means that loop iterations (or parts of the loop body) are assigned to a heap partition and no iteration accesses the heap associated with a different partition than its 'own'. We use cut-points and heap footprint labels to assign heap partitions to loop iterations. Failing to prove the partitioning property in all iterations restarts the first phase: Generally, there are multiple options for the initial partitioning of the program state into P portions. If the first option failed, the analysis tries the next one until we either obtain a successful proof or all options have been tested. Using the motivating example from Section III, we first describe the initial partitioning and cut-point insertion followed by the proof of disjointness in all iterations.

I. Inserting Cut-points

Our analysis tries to split up spatial formulae at *cut-points*:

Definition 7 (Cut-point) *A cut-point is a program variable pointing to a heaplet in the program state formula.*

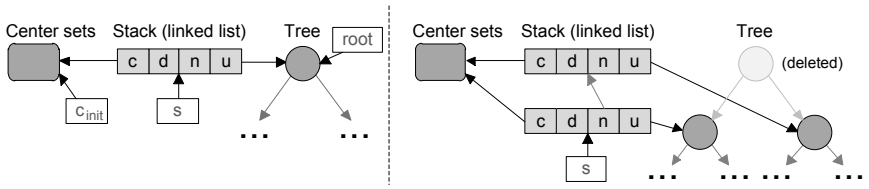


Figure 3: *Pre-state before execution of the first (left) and the second loop iteration (right).*

The program can only interact with heap-allocated data via pointers (program variables). Useful heap partitioning requires the program to have access to each partition via pointers, *e.g.* given $ls(u, x'_1) * ls(x'_1, v) * ls(v, \texttt{nil})$, the program can access the first and third list segment via cut-points u and v, as opposed to the second list segment since x'_1 is not a cut-point. The goal is to obtain P cut-points in the pre-state of a loop iteration (*i.e.* the state before the loop body executes). We symbolically execute the program fragment before the loop in Listing 1 to determine the program state just before the loop body executes for the first time:

$$s = s'_0 \wedge root = u'_0 \wedge c_{init} = c'_0 \wedge tree(u'_0) * c'_0 \mapsto _ * s'_0 \mapsto [\texttt{u} : u'_0, \texttt{c} : c'_0, \texttt{n} : \text{nil}] \tag{7}$$

Fig. 3, left, depicts (7), which contains the stack record (pointed to by s'_0), the tree, and a center set (pointed to by c'_0). Each heap predicate in (7) is also referenced by a cut-point. We consider the program variable s first and select the predicate $m_1 \equiv s'_0 \mapsto [\texttt{u} : u'_0, \texttt{c} : c'_0, \texttt{n} : \texttt{nil}]$. Next, we try to find another predicate m_2 of the same shape as m_1 in the formula. To this end, we create a template $m_2 \equiv t'_0 \mapsto [\texttt{u} : t'_1, \texttt{c} : t'_2, \texttt{n} : t'_3]$ and set $A \equiv (7)$. We then ask coreStar's theorem prover whether it can match two predicates in A with $m_1 * m_2$. If the prover is successful, A contains the desired second predicate m_2 and we can extract it from the proof. If it is unsuccessful, we modify A by symbolically executing the next iteration, which is the case in this example. The loop pre-state after unrolling is (depicted in Fig. 3, right):

$$\begin{aligned} &s = s'_2 \wedge tree(u'_1) * tree(u'_2) * c'_1 \mapsto _ \\ &* s'_2 \mapsto [\texttt{u} : u'_1, \texttt{c} : c'_1, \texttt{n} : s'_1] * s'_1 \mapsto [\texttt{u} : u'_2, \texttt{c} : c'_1, \texttt{n} : \text{nil}] \end{aligned} \tag{8}$$

Now the matching is successful. We introduce a second cut-point s_b and let it point to the only possible candidate m_2: $s = s'_2 \wedge s_b = s'_1$, which satisfies the necessary condition for partitioning: (8) contains $P = 2$ heaplets m_1 and m_2, of the same shape and referenced by cut-points. Next, we ask our proof engine described in the next section to prove that, in all subsequent loop iterations, the spatial part of the state can be split into $P = 2$ partitions, each of which being assigned either to cut-point s or s_b. As explained in the next section, this proof fails for (8) because of the lack of a second predicate $c_x \mapsto _$ (the pointer aliasing is illustrated in Fig. 3, right). Hence, we abandon the inserted cut-point, peel off another loop iteration, and reach the pre-state of the third iteration:

$$\begin{aligned} &s = s'_4 \wedge tree(u'_3) * tree(u'_4) * tree(u'_2) \\ &* c'_2 \mapsto _ * c'_1 \mapsto _ * s'_4 \mapsto [\texttt{u} : u'_3, \texttt{c} : c'_2, \texttt{n} : s'_3] \\ &* s'_3 \mapsto [\texttt{u} : u'_4, \texttt{c} : c'_2, \texttt{n} : s'_1] * s'_1 \mapsto [\texttt{u} : u'_2, \texttt{c} : c'_1, \texttt{n} : \text{nil}] \end{aligned} \tag{9}$$

The formula describes the program state shown in Fig. 2. We repeat the cut-point insertion. Our tool explores all possible cut-point assignments (there are multiple options now) and launches the proof engine in the next section for each candidate assignment. Assume we have assigned the second cut-point to the heaplet pointed to by s'_1: $s = s'_4 \wedge s_b = s'_1$. Starting from this pre-state, our proof engine can now successfully prove the parallelization hypothesis of $P = 2$. The next section explains how it works. Note that, for other programs, we may not find a successful proof in which case we abort after $L_{\max}$ unrollings.

II. Proving Communication-free Parallelism

The starting point for the proof engine is the program state obtained after the initial unrolling of a finite number of loop iterations above. In our example, we start with (9) and the two cut-points s and s_b, and aim to split the heap accessed during the loop iterations into two portions a and b. During symbolic execution of the loop body, we distinguish between two 'cut-point states' depending on whether we are currently accessing data structures 'belonging' to cut-point s (portion a) or s_b (portion b). Our tool constantly tracks the current cut-point state during symbolic execution of loop iterations. We switch to a different cut-point state once we have accessed a heaplet pointed to by a different cut-point variable as the one assigned to the current state. We assign label $a \in \text{Lab}$ to all heaplets accessed during execution in cut-point state a (cut-point s), and similarly for b (cut-point s_b). We count *pointer dereferencing* and *delete* as an access. Our label assignment and cut-point state propagation through the program's CFG are implemented as add-ons to coreStar. Tracking the cut-point state together with footprint label assignment to heaplets allows the analysis to assign heap partitions to loop iterations.

The parallelization goal is to partition the loop iteration space into two groups labeled a and b, and we try to establish the fact that a heaplet accessed by an iteration in cut-point state a (of group a) is never accessed by another iteration of group b. In other words, we try to prove that the separation of the accessed heap into a and b is invariant in each subsequent loop iteration. If the number of iterations was known at compile time, we could symbolically execute all iterations to prove this property. However, in general, this number is not statically determinable because of the data dependent loop condition (Listing 1, line 15). Hence, we perform a *fix-point calculation* [Magill et al.2006] for proving that the separation property is loop invariant. Our fix-point calculation adopts and modifies a technique described by Magill *et al.* [Magill et al.2006] and works as follows:

1. Start with the pre-state of the loop M_0^{pre} equal to (9) with cutpoints s_a and s_b inserted.

2. Symbolically execute $\{M_i^{pre} \wedge b\}$ '*loop body*' $\{M_{i+1}^{post}\}$, b is the loop condition, i is the iteration counter and M_{i+1}^{post} describes the program state after the loop body in iteration i has been executed. We attach labels a or b to heaplets corresponding to the current cut-point state. If we find both labels a and b on a heaplet, it means that this heaplet has been accessed by at least one iteration of cut-point state a and one of state b: The separation into disjoint partitions is not maintained and we abort, report a failed proof and restart the cut-point insertion to obtain a different initial partitioning. If only either a or b are attached to any heaplet we continue with the next step.

3. Absorb singleton heaplets in M_{i+1}^{post} in the recursive predicates of Def. 2-4. For example, a formula can be rewritten so that the head node of a linked list and the tail list can be merged into one linked list. Formally, if M_{i+1}^{post} is $s_a \mapsto [\texttt{n} : x_1'] * ls(x_1', \texttt{nil})$ we can fold $s_a \mapsto [\texttt{n} : x_1']$ into the ls-predicate resulting in $M_{i+1,folded}^{post} \equiv ls(s_a, \texttt{nil})$. This step is called *abstraction* as we lose some information here: Instead of knowing that the heap contains a linked list with at least one entry, we now know that it contains a linked list which possibly can be empty. However, the information of having at least one node in the list is not required by our analysis because we are interested in the shape of the heap layout only. We maintain a set of abstraction rules which we provide to the theorem prover as described in Section IV and which define what is a valid abstraction. Our abstraction rules forbid folding across program variables, *i.e.* $s_a \mapsto [\texttt{n} : x] * ls(x, \texttt{nil})$ does not get merged into $ls(s_a, \texttt{nil})$ because x is a program variable. Note that this also prevents folding across cut-points. The set of footprint labels attached to a predicate resulting from merging two predicates is the union of both original label sets. The abstraction step prevents accumulating singleton heaplets

such as $s_a \mapsto [\mathtt{n} : x'_1]$ during repeated execution of the loop body and is crucial for convergence of the fix-point calculation.

4. The fix-point calculation terminates if M^{post}_{i+1} implies a post-state of one of the previous iterations $M^{post}_k, k = 0, ..., i$. Formally, we ask coreStar's theorem prover to decide $M^{post}_{i+1} \vdash \bigvee_{k=0..i} M^{post}_k$ (the right hand side is the disjunction of all previous post-states). If the implication does not hold we set $M^{pre}_{i+1} := M^{post}_{i+1}$ and continue with step 2).

For our example, we reach a fix-point after 7 iterations of steps 1) to 4). Note that, for another candidate for the cut-point assignment ($s_b = s'_3$ instead of $s_b = s'_1$) as discussed above, the fix-point calculation would have been aborted because we had eventually reached the state $\langle c'_2 \mapsto _ \rangle_{\{a,b\}}$. The runtime complexity of the analysis is dominated by the number of disjunctive clauses that are generated when branch instructions are symbolically executed. In the worst case, this number grows exponentially with the number of conditionals, hence grows exponentially with the number of fix-point iterations if such statements are in the loop body. However, we do not see an exponential growth in our case studies due to clause merging and folding terms into recursive predicates. Furthermore, Magill's folding heuristic works well in practice, but it cannot guarantee convergence of the fix-point calculation and hence an upper bound on its iterations in general due to the incompleteness of the heuristic.

The successful fix-point calculation tells us that the heap accessed by the loop, after peeling off a finite number of initial loop iterations, can be partitioned into two disjoint regions labeled *a* and *b*. Furthermore, it tells us that the partitioning will be maintained for all following loop iterations, each of which will either access heap portion *a* or *b*, but not both. A code transformation can now split the original code into two code fragments, each having access to its own heap partition as shown in Listing 2. What remains is to assign all heap-manipulating program statements in the loop preamble and initially unrolled iterations to the correct partitions. This is described in the following section.

III. Assigning Heap Partition Information to Statements

After the analysis has determined that the loop can be split into two loops with access to their private heap partitions, we must ensure that the pointers used in the preamble and unrolled iterations refer to the correct memory partition. For example, the predicate $s'_4 \mapsto [\mathtt{u} : u'_3, \mathtt{c} : c'_2, \mathtt{n} : s'_3]$ in (9) gets attached the partition label *a* during the loop analysis: $\langle s'_4 \mapsto [\mathtt{u} : u'_3, \mathtt{c} : c'_2, \mathtt{n} : s'_3] \rangle_{\{a\}}$. The heaplet described by this predicate, however, was allocated (*new*-statement, Listing 1, line 17) and written to (pointer dereferencing, also line 17) in the second iteration that was peeled off during the cut-point insertion. Consequently, we must attach the partition information to these program statements as well.

We link the partition assignment to heap-manipulating program commands with a combination of our labeled symbolic execution (footprint labels according to the cut-point state) with the standard labeled symbolic execution in [Raza et al.2009] (a unique footprint label for each program statement). Recall that (9) describes the program state just before launching the fix-point calculation. During the fix-point calculation, we record each heaplet the first time it gets assigned a label. Recording on first label assignment is necessary because, for instance, we may lose track of the predicate $c'_2 \mapsto _$ in (9) as it will be disposed (Listing 1, line 11) during the course of fix-point calculation before we even access $c'_1 \mapsto _$ for the first time. After a successful fix-point calculation,

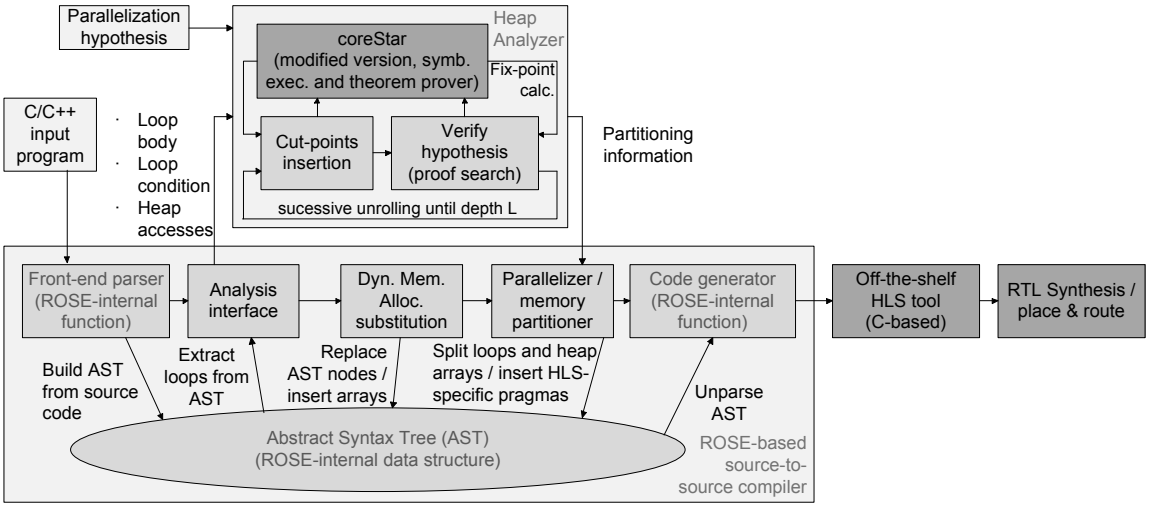


Figure 4: *CAD flow including the heap analyzer, source translator and third party tools for HLS and RTL implementation.*

we stitch together all snapshots resulting in a labeled version of (9):

$$
\begin{aligned}
& s_a = s'_4 \wedge s_b = s'_1 \wedge \langle tree(u'_3)\rangle_{\{a\}} * \langle tree(u'_4)\rangle_{\{a\}} * \langle tree(u'_2)\rangle_{\{b\}} \\
& * \langle c'_2 \mapsto _\rangle_{\{a\}} * \langle c'_1 \mapsto _\rangle_{\{b\}} * \langle s'_4 \mapsto [\mathtt{u} : u'_3, \mathtt{c} : c'_2, \mathtt{n} : s'_3]\rangle_{\{a\}} \\
& * \langle s'_3 \mapsto [\mathtt{u} : u'_4, \mathtt{c} : c'_2, \mathtt{n} : s'_1]\rangle_{\{a\}} * \langle s'_1 \mapsto [\mathtt{u} : u'_2, \mathtt{c} : c'_1, \mathtt{n} : \mathrm{nil}]\rangle_{\{b\}}
\end{aligned} \tag{10}
$$

During the symbolic execution of the loop preamble and iteration unrolling prior to the fix-point calculation, we also record the program statements that accessed each of the heaplets in (10) by assigning a second set of footprint labels (FT) as in the standard label assignment in [Raza et al.2009]. This set contains a unique label for each accessing statement, e.g. $FT = \{l_2, l_3, l_7\}$ for statements 2, 3 and 7. With these two label sets we obtain a mapping

$$m : \mathtt{Lab} \rightarrow \{a, b\} \tag{11}$$

where Lab is the set of all unique labels assigned to heap-manipulating program commands in the loop preamble and unrolled iterations. This mapping allows us to assign the correct heap partition information to each pointer access. This information is used by the source-to-source transformation for correct code instrumentation. The above analysis provides both memory partitioning information (by labels assigned to heaplets) and the legality of parallelization (by a successful fix-point calculation). Algorithm 1 summarizes our heap analysis. Next, we explain how this information is used in a source-to-source translator for automated parallelization and partitioning.

VI. Implementation

Our tool flow consists of three parts: the heap analyzer, a source-to-source compiler, and a back-end HLS and FPGA synthesis tool. Fig. 4 shows the complete tool flow.

I. Heap Analyzer

Our heap analyzer connects to the analysis interface of the source translator and implements the two-step analysis described above. It is written in OCaml and is largely based on our modified

ALGORITHM 1: Heap partitioning analysis

Input:
Loop body specification (code),
Initial state formula $(\Pi \wedge \Sigma_{\{FT\}})^{\text{initial}}$ (from symbolic execution of loop preamble),
Parallelization hypothesis P;
Output:
Validity of parallelization hypothesis (*success*),
Number of initial unrollings required (*it*),
Assignment of pointer statements to heap partitions (m);
Data:
it ; `// Iteration counter (number of iterations to be unrolled)`
C ; `// Set of cut-points`
$S_{\text{cutpoints}}$; `// Set of cut-point states`
$\Pi \wedge \Sigma_{\{FT\}}$; `// state formula in separation logic (attached footprint label set` FT`)`
$\Pi \wedge \Sigma_{\{CS\}}$; `// state formula in separation logic (attached cut-point state set` CS`)`
m ; `// label mapping` m`:` $FT \rightarrow CS$

$it \leftarrow 0$;
$C \leftarrow \emptyset$;
$\Pi \wedge \Sigma_{\{FT\}} \leftarrow (\Pi \wedge \Sigma_{\{FT\}})^{\text{initial}}$;
$success \leftarrow$ `false`;
repeat
  while not checkIfValidCutpInsertion$(\Pi \wedge \Sigma_{\{FT\}}, C)$ **do**
    $\Pi \wedge \Sigma_{\{FT\}} \leftarrow \text{SymbExec}(\Pi \wedge \Sigma_{\{FT\}}, it)$; `// peel off` it `loop iterations (Section I)`
    $\Pi \wedge \Sigma_{\{FT\}}, C \leftarrow \text{CutpInsert}(\Pi \wedge \Sigma_{\{FT\}}, P)$; `// insert` P `cut-points (Section I)`
    $it \leftarrow it + 1$;
  end
  $S_{\text{cutpoints}} \leftarrow \text{AssignCPStates}(C)$; `// assign states to cut-points (Section II)`
  $\Pi \wedge \Sigma_{\{CS\}}, success \leftarrow \text{FixpCalc}(\Pi \wedge \Sigma_{\{FT\}}, C, S_{\text{cutpoints}})$; `// fix-point calculation (Section II)`
  $m \leftarrow \text{GetLabelMapping}(\Pi \wedge \Sigma_{\{CS\}}, \Pi \wedge \Sigma_{\{FT\}})$; `// obtain label mapping (Section III)`
until *success* **or** $it \geq L_{\max}$;
return *success*, it, m;

version of coreStar [Botinčan et al.2011] which we extended to include labeled symbolic execution and cut-point processing. coreStar mainly consists of a symbolic execution engine and a theorem prover. The former is performed on the control flow graph of the program which is built internally. It operates on an intermediate representation of the input program in `coreStarIL`, which the programming language in Def. 5, together with the specifications in Def. 6, can be straightforwardly translated to. The theorem prover is generic in that it leaves the definition of the logic theory to the user. Our heap analyzer currently uses 122 logic rules as described in Section IV which define pure and spatial predicates, such as those in Def. 2-4, and how footprint labels are propagated. These rules also define, for example, under what conditions a points-to predicate describing a singleton list node can be 'gobbled up' by an existing linked list predicate in order to ensure convergence of the fix-point calculation.

There are four elements in the heap analyzer which are currently not yet automated. Firstly, the parallelization hypothesis is specified by the user as discussed above. Secondly, the case studies in the next section are still based on a manual translation from the input code into `coreStarIL`. An automatic translation from the LLVM intermediate representation (which C++ code can be compiled

to) into `coreStarIL` is a purely syntactic transformation and is currently under development. Thirdly, although this case does not occur in our case studies, scaling the tree-based benchmarks to parallelization degrees $P > 2$ (non-power-of-two values are possible) would require some guidance to select a good partitioning as described in Section VIII. Finally, in some case studies, a context assertion, *i.e.* a state formula before entering the program (fragment) under analysis must be provided. The integration of a technique for a compositional analysis that infers the context assertion automatically [Calcagno and Distefano2011] is under development.

II. Source-to-source Compiler

Our source translator is built on the ROSE source compiler infrastructure [LLNL2014] which provides a library of C++ functions for source code analysis and transformation. Our code analysis and transformations are a collection of C++ classes which traverse and modify the abstract syntax tree (AST) of the input program: The analysis interface determines the type of heap-allocated data through the syntax analysis of *new* and *delete* statements, and finds loops in the syntax tree and extracts the body, condition and context.

The subsequent replacement of the standard C++ dynamic memory allocation ensures synthesizability by off-the-shelf HLS tools. The heap is replaced by arrays and the corresponding pointers are converted to integer variables. Occurrences of *new* and *delete* statements are grouped according to the type of their operand and custom allocator functions are instantiated for each type as a replacement. Dynamic type casts are currently not supported. Our fixed-size allocator is a standard implementation using a *free-list* which keeps track of occupied memory space [Winterstein et al.2013b]. It is implemented in a C header file which contains template functions for dereferencing, allocation and disposal. Dereferencing of heap-directed pointers is substituted using an auxiliary static pointer variable added by the tool as shown in Listing 3. We stress that this work focuses on memory partitioning and parallelization and is therefore orthogonal to work that determines a bound on the amount of allocated heap memory. Cook *et al.* [Cook et al.2009] describe a technique for finding parametric worst-case bounds on the heap consumption based on a separation logic-driven analysis which can be used for this purpose in our benchmarks.

In the last step of the transformation, the memory partitioner / parallelizer receives information from the heap analyzer that a parallelization is legal and how the heap arrays have to be partitioned. In addition to heap-carried dependencies, we need to take 'store dependencies' between normal program variables into account. For these, we use standard data flow analyses such as the definition-usage analysis which determines the variable write-read relation between CFG nodes in the program. We include the DEF-USE analysis provided by the ROSE library in the tool. The parallelization analysis, if successful, has divided the loop iterations into P independent groups, where P is the degree of parallelization. Additionally, several loop iterations may have been peeled off by the analysis as it is the case in our motivating example described above. Our source transformation removes the original loop from the AST and inserts two sections of code: 1) The original loop body guarded by an if-conditional with the loop condition representing the iterations that have been unrolled during the analysis. 2) P loops of the same type and with the same loop condition as the original one, each containing the fragment of the loop body which accesses one of the independent groups. Some HLS tools, such as Vivado HLS, require code fragments to be wrapped in functions in order to schedule parallel execution of them. The last step of the parallelization transformation is an 'outlining' step that wraps the sub-loops into functions and inserts calls at the original source code position.

```
1 // ... substitution of u->d = d;
2 // ... u is the new pointer variable (integer type), heap_0_1 the new heap array
3 u_ptr = make_pointer(heap_0_1, u); // auxiliary variable u_ptr
4 u_ptr->d = d;
```

Listing 3: *Substituting pointer accesses to heap-allocated data..*

The arrays representing the heap memory are partitioned accordingly. The heap analysis tells us what heap partition a pointer accesses. The partition index is added to the substitution of pointer accesses (*e.g.* Listing 3: *heap*_0_*x*, where *x* is the partition index). We finally customize the dynamic memory allocator according to the parallelization: Each of the *P* new loops accesses its own disjoint heap region. Consequently, we can restrict the scope of *new* / *delete* operations that are made by a loop to its heap array partition and instantiate an allocator, including the freelist, for each partition.

VII. Case Studies

We test the tool flow in Fig. 4 using C++ implementations taken from real-world applications. We use Xilinx Vivado HLS 2014.1 as a back-end HLS tool and Xilinx Vivado 2014.1 for RTL synthesis. However, since our optimizations are at source code level, our tool can be also used in combination with a different HLS tool. Our benchmark applications are:

Merger - The program maintains four linked lists whose nodes are sorted according to a key. It repetitively reads four key-value pairs from its interface and performs a sorted insertion in each list for each pair. After a constant number of pairs has been received, it repeatedly deletes the head node of that list which contains the smallest key until all lists are empty. The output is a sorted sequence of all key-value pairs. A distinguishing feature of this applications is that the loop under analysis contains a sub-loop. During each symbolic execution of an outer loop iteration the proof engine requires a few inner iterations to converge to a loop invariant for the inner sub-loop. We consider this benchmark a representative example from the class of list processing progams.

Tree Deletion - This application performs a full traversal a pointer-linked tree data structure and deletes the visited tree nodes after some computation using the node data.

Filter - This is the motivating example in Section III which is taken from the direct implementation of the filtering algorithm for efficient *K*-means clustering [Kanungo et al.2002]. Our tool splits the loop in Listing 1 and partitions the heap memory with degree *P*. The code fragment is embedded in a larger program which includes tree build-up and center processing to form a complete clustering application. This example is interesting in that it is more complicated than a usual toy example: Loop iterations allocate and dispose center sets, preceded by a data-dependent conditional, which carry a heap dependence between some iterations. Our analysis figures out that there are no heap-carried dependencies between iterations which access tree nodes without a parent-child relation.

Reflect Tree - The application traverses a binary tree in pre-order fashion and recursively swaps the left and right child pointer of each node, thus producing a mirrored tree. It also performs some computation at each node and updates the data fields of the tree nodes.

Build Tree - The application builds up a kd-tree [Kanungo et al.2002] in the heap by recursively sub-dividing a set of 3-dimensional data points. This benchmark is a special case: We force (overriding the complaints of the DEF-USE analysis) our tool to partition the heap and split up the build-tree kernel so that it builds up a sub-tree in each partition. However, Vivado HLS does not schedule parallel execution of the duplicated code fragments because they share a common access to the input data set.

Table 1: *Implementation results and comparison.*

	P	Slices	DSP	BRAM	Clock	Cycles	S
Merger (4×2048 random input key-value pairs)							
Base line (reference)	1	547	0	96	8.2 ns	12711k	1.0
Blind parall. (without analysis)	4	842	0	96	7.5 ns	12710k	1.0
Autom. parall. (with analysis)	4	1026	0	96	6.4 ns	3304k	**3.8**
Tree Deletion (16383 tree nodes)							
Base line (reference)	1	2284	9	515	6.9 ns	1900k	1.0
Blind parall. (without analysis)	2	3177	12	515	5.4 ns	1860k	1.0
Autom. parall. (with analysis)	2	4637	27	515	5.1 ns	1027k	**1.8**
Filter (16384 3-dim. data points, 32767 tree nodes, $K = 128$ clusters)							
Base line (reference)	1	2449	20	314	6.0 ns	998k	1.0
Blind parall. (without analysis)	2	2990	40	312	5.8 ns	1029k	1.0
Autom. parall. (with analysis)	2	5004	80	317	6.3 ns	572k	**1.7**
Reflect Tree (16383 tree nodes)							
Base line (reference)	1	1313	12	291	5.0 ns	754k	1.0
Blind parall. (without analysis)	2	1449	15	291	5.7 ns	762k	1.0
Autom. parall. (with analysis)	2	2449	36	291	5.8 ns	407k	**1.8**
Build Tree (16383 tree nodes)							
Base line (reference)	1	1495	0	240	5.1 ns	4423k	1.0
Blind parall. (without analysis)	2	1921	0	257	5.6 ns	4743k	0.9
Autom. parall. (with analysis)	2	3039	0	273	5.6 ns	4145k	**1.1**
Octree Traversal (16383 tree nodes)							
Base line (reference)	1	1895	12	483	5.5 ns	1271k	1.0
Blind parall. (without analysis)	2	2051	21	483	5.7 ns	1282k	1.0
Autom. parall. (with analysis)	2	3368	36	483	5.3 ns	684k	**1.8**

Octree Traversal - The program traverses a heap-allocated octree and permutes all of the eight child pointers at each node. It additionally performs some computation at each node. Octrees are popular data structures in graphics applications such as ray tracing.

The target device is a Virtex 7 FPGA (Xilinx VC707 evaluation board, xc7vx485tffg1761-2) and all results are taken from placed and routed designs. We report resource utilization in slices, DSP slices (DSP) and 36K-Block RAMs (BRAM). We also report the achieved clock speed (target 5ns) and the number of clock cycles required for task completion which we determine via simulations of the generated RTL designs. The RTL test benches for the benchmarks are fed with application-specific input data. For each test case, Table 1 shows the implementation results for three cases: The *base line* case shows the implementation if the tool only ensures synthesizability (syntactical substitution of

dynamic memory allocation and heap-directed pointers, no heap analysis) without parallelization. The second case shows the results of "blind" loop unrolling. Instead of using our source-to-source compiler, we use the standard Vivado directive for partial loop unrolling here which instantiates P parallel loop kernels. We call this case "blind parallelization" because it is not guided by our heap analysis and no heap partitioning is performed by Vivado HLS. The third row shows results if the tool flow uses the heap analyzer for memory partitioning and parallelization using our source transformation (automatic parallelization, degree P), an optimization that cannot be done by Vivado HLS itself as shown in the previous case and as explained in [Winterstein et al.2013b]. The speed-up S relates the cycle count of the automatically parallelized benchmarks to that of the base line case.

Vivado HLS does not schedule parallel execution of the loop kernels without explicit heap partitioning (second case). Including a directive for implementing dual-port memories to increase the number of access ports did not have any influence on the scheduling in our cases. Our heap analysis detects the independence of the four linked lists in the **Merger** benchmark and parallelizes the application. The speed-up in terms of cycle count is close to the maximum speed-up of $P = 4$. The analysis also partitions the data structures of **Filter** and **Tree Deletion** which enables successful parallelization ($S > 1.7$ compared to the base case). As opposed to the **Merger** benchmark, the tree-based applications require unrolling of one or two loop iterations until disjointness of sub-structures can be determined (Section II) which explains the resource overhead compared to the base case (especially noticeable in terms of DSP slices). All other tree-based benchmarks require one loop iteration to be peeled off before the parallelization is successful. We observe an expected acceleration of $S \approx 1.8$ in terms of cycle count reduction, except for **Build Tree**. The heap in **Build Tree** can be partitioned by our tool, but the program cannot be parallelized because it contains a non-heap allocated array that is accessed by both sub-loops. In such cases, the user may opt not to use our technique, based on two parallelizability checks prior to RTL implementation. Firstly, the ROSE-internal DEF-USE analysis will report the additional data dependency. Secondly, tools like Vivado HLS provide information about the scheduling result, from which the user can see whether or not the new loop kernels have been scheduled for parallel implementation.

For the benchmarks **Merger** and **Filter**, we include an additional case study by adding two reference designs for comparison shown in Table 2: hand-optimized HLS designs using Vivado HLS [Winterstein et al.2013b] and hand-written RTL designs in VHDL [Winterstein et al.2013a]. Comparing resources, clock frequency and cycle count, we observe further improvements obtained from manual source code refactoring: In the hand-optimized HLS design of **Filter**, we manually flattened loop nests in order to enable efficient pipelining [Winterstein et al.2013b] of the tree traversal loop, an optimization beyond the scope of this paper. This loop contains two sub-loops with variable bounds and code at each loop-level. It is not a perfectly or semi-perfectly nested loop, which prevents the application of VivadoâĂŹs loop flattening directive. Without loop flattening, only the inner loops can be pipelined, which results in less speed-up compared to the manually flattened loop. The manual HLS design remains 3× slower than the RTL implementation because the tree traversal must be distributed over a producer and a (flattened) consumer loop, while it is implemented in a single pipeline in the RTL design. A detailed discussion of these implementations is given in [Winterstein et al.2013b]. Furthermore, the use of bit width customizations of data items and pointers in the manual designs, which reduces the memory consumption, is beyond the scope of this work.

We ran our case studies on a machine with 16GB of memory and an Intel i7-3770 processor, 3.40GHz. The overall tool running time depends on several factors and varies significantly across our benchmarks. Table 3 shows the tool running time broken down into cut-point insertion,

Table 2: *Comparison with hand-written HLS/RTL designs.*

	Slices	DSP	BRAM	Clock	Cycles	S
Merger (4×2048 random input key-value pairs)						
Base line (ref.)	547	0	96	8.2 ns	12711k	1.0
Autom. parall.	1026	0	96	6.4 ns	3304k	**3.8**
Hand-writ. HLS	832	0	60	5.8 ns	3341k	3.8
Hand-writ. RTL	888	0	52	5.0 ns	2197k	5.8
Filter (16384 3-dim. data points, 32767 tree nodes, $K = 128$ clusters)						
Base line (ref.)	2449	20	314	6.0 ns	998k	1.0
Autom. parall.	5004	80	317	6.3 ns	572k	**1.7**
Hand-writ. HLS	6062	36	264	5.9 ns	165k	6.0
Hand-writ. RTL	6449	40	236	5.0 ns	54k	18.4

Table 3: *Tool running time.*

	Cut-point Insertion	Fix-point Calculation	Source Transf.	Total (analysis)	HLS + RTL Impl.
Merger	264.9s	479.1s	15.4s	759.4s	394.6s
Tree Deletion	0.5s	1.8s	13.4s	15.7s	862.3s
Filter	2.1s	93.0s	14.3s	109.4s	1110.7s
Reflect Tree	0.4s	5.8s	13.6s	19.8s	747.8s
Build Tree	0.6s	2.1s	14.4s	17.1s	642.0s
Octree Traversal	50.1s	72.2s	13.1s	135.4s	871.2s

fix-point calculation and source code transformation. We also show the time spent on HLS and RTL implementation. The source translator's running time, whose largest components are repeated AST traversals and the DEF-USE analysis, is similar for all benchmarks. The running time of the heap analyzer for **Merger** and **Octree Traversal** is longer than for the other benchmarks because the symbolic execution of the loop body is substantially slower. In the former case this is due to the inner sub-loop analysis described above and the latter generates a large number of predicates describing the octree nodes. The running time of the heap analyzer for **Tree Deletion**, **Reflect Tree** and **Build Tree** is small because the abstraction rules applied during fix-point calculation quickly merge predicates into 'small' formulae specifying the program state. This ensures convergence after few iterations and fast symbolic execution. The abstraction rules cannot be applied as aggressively in the **Filter** benchmark due to the structure of the loop body. This results in many disjunctive formulae (up to 11) which are carried from one fix-point iteration to the next and slow down the symbolic execution.

VIII. Conclusions

This paper presents an extended version of our work in [Winterstein et al.2014]. We describe our tool flow that automatically parallelizes loops in pointer–manipulating C/C++ programs and distributes

heap-allocated, pointer-linked data structures over separate banks of on-chip block memory in order to leverage the memory-level parallelism in FPGAs. The core of our tool flow is the heap analyzer for proving communication-free parallelism in loops. We develop and implement a hypothesis-based algorithm for the disjointness/dependence analysis which draws on several existing techniques developed in the separation logic framework: symbolic execution, heap footprint analysis and loop invariant synthesis. The outcome of the analysis is information about the legality of parallelization and an assignment of heaplets to on-chip memory partitions. The analysis is accompanied by automated code transformations which ensure the synthesizability of the pointer-manipulating program by standard HLS tools, and implement the parallelization and memory partitioning. Our source code translator performs transformations at human-readable C code level which allows us to stay as independent as possible of a specific HLS tool. We demonstrate the successful parallelization and memory partitioning by our tool flow using six real-life applications and using Xilinx Vivado HLS as an exemplary back-end tool. The HLS implementations parallelized by our tool achieve the expected acceleration by a factor of $1.7\times - 3.8\times$ in terms of cycle count compared to the non-parallelized implementations.

I. Future Directions

Our tool flow performs the core tasks, analysis and source code transformation, automatically. The loop body extracted from the AST, however, is currently manually translated from C into the `coreStarIL` representation. We plan to automate the translation leveraging the LLVM [Lattner and Adve2004] framework as an intermediate step. Another aspect is to improve the analysis. Our cut-point insertion greedily searches for a necessary condition for parallelization. If we were to parallelize the motivating example with a degree of four, our analysis would split the left sub-tree twice instead of splitting each left and right sub-tree once which is better in terms of acceleration. Currently, our analysis thus lacks the ability to compare partitioning alternatives, which we want to address in future work.

References

[Babb et al.1999] Jonathan Babb, Martin Rinard, Andras Moritz, Walter Lee, Matthew Frank, Rajeev Barua, and Saman Amarasinghe. 1999. Parallelizing applications into silicon. In *Proceedings of the Symposium on Field-Programmable Custom Computing Machines*. IEEE, Napa Valley, CA, 70–80.

[BDTI2010] BDTI. 2010. An Independent Evaluation of: The AutoESL AutoPilot High-Level Synthesis Tool. (2010). Retrieved March 2, 2014 from `http://www.bdti.com/Resources/BenchmarkResults/HLSTCP/AutoPilot`

[Benabderrahmane et al.2010] Mohamed-Walid Benabderrahmane, Louis-Noel Pouchet, Albert Cohen, and Cé Bastoul. 2010. The polyhedral model is more widely applicable than you think. In *Proceedings of the International Conference on Compiler Construction*. Springer-Verlag, Paphos, Cyprus, 283–303.

[Berdine et al.2005] Josh Berdine, Cristiano Calcagno, and Peter O'Hearn. 2005. Symbolic execution with separation logic. In *Proceedings of the Asian Conference on Programming Languages and Systems*. Springer-Verlag, Tsukuba, Japan, 52–68.

[Bondhugula et al.2008] Uday Bondhugula, Albert Hartono, J. Ramanujam, and P. Sadayappan. 2008. A Practical Automatic Polyhedral Parallelizer and Locality Optimizer. *SIGPLAN Not.* 43, 6 (June 2008), 101–113.

[Botinčan et al.2011] Matko Botinčan, Dino Distefano, Mike Dodds, Radu Grigore, and Matthew J. Parkinson. 2011. coreStar: the core of jStar. *Boogie* (2011), 65–77.

[Botinčan et al.2013] Matko Botinčan, Mike Dodds, and Suresh Jagannathan. 2013. Proof-directed parallelization synthesis by separation logic. *ACM Transactions on Programming Languages and Systems* 35, 2 (July 2013), 1–60.

[Calcagno and Distefano2011] Cristiano Calcagno and Dino Distefano. 2011. Infer: an automatic program verifier for memory safety of C programs. In *Proc. Third Int. Conf. on NASA Formal Methods*. Springer-Verlag, Pasadena, CA, 459–465.

[Canis et al.2011] Andrew Canis, Jongsok Choi, Mark Aldham, Victor Zhang, Ahmed Kammoona, Jason H. Anderson, Stephen Brown, and Tomasz Czajkowski. 2011. LegUp: High-level Synthesis for FPGA-based Processor/Accelerator Systems. In *Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*. ACM, Charlotte, NC, 33–36.

[Cong et al.2011] Jason Cong, Wei Jiang, Bin Liu, and Yi Zou. 2011. Automatic memory partitioning and scheduling for throughput and power optimization. *ACM Transactions on Design Automation of Electronic Systems* 16, 2 (March 2011), 1–25.

[Cook et al.2009] Byron Cook, A. Gupta, S. Magill, A. Rybalchenko, J. Simsa, S. Singh, and V. Vafeiadis. 2009. Finding heap-bounds for hardware synthesis. In *Proceedings of the Conference on Formal Methods in Computer-Aided Design*. IEEE, Austin, TX, 205–212.

[Cook et al.2010] Byron Cook, Stephen Magill, Mohammad Raza, Jiri Simsa, and Satnam Singh. 2010. Making fast hardware with separation logic. (2010). `http://www.cs.cmu.edu/~smagill/papers/fast-hardware.pdf` unpublished.

[Feautrier1991] Paul Feautrier. 1991. Dataflow analysis of array and scalar references. *International Journal of Parallel Programming* 20, 1 (1991), 23–53.

[Ghiya et al.1998] Rakesh Ghiya, L Hendren, and Yingchun Zhu. 1998. Detecting parallelism in C programs with recursive data structures. *IEEE Transactions on Parallel and Distributed Systems* 1 (1998), 35–47.

[Guo et al.2007] Bolei Guo, Neil Vachharajani, and David I. August. 2007. Shape analysis with inductive recursion synthesis. *ACM SIGPLAN Notices* 42, 6 (June 2007), 256.

[Huang et al.2013] Qijing Huang, Ruolong Lian, A. Canis, Jongsok Choi, R. Xi, S. Brown, and J. Anderson. 2013. The Effect of Compiler Optimizations on High-Level Synthesis for FPGAs. In *Proceedings of Field-Programmable Custom Computing Machines*. IEEE, Seattle, WA, 89–96.

[Kanungo et al.2002] Tapas Kanungo, David M. Mount, Nathan S. Netanyahu, Christine D. Piatko, Ruth Silverman, and Angela Y. Wu. 2002. An efficient k-means clustering algorithm: analysis and implementation. *IEEE J PAMI* 24, 7 (July 2002), 881–892.

[Lattner and Adve2004] Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis Transformation. In *Proceedings of the International Symposium on Code Generation and Optimization*. IEEE Computer Society, Palo Alto, CA, 75–86.

[Liu et al.2007] Qiang Liu, George a. Constantinides, Konstantinos Masselos, and Peter Y.K. Cheung. 2007. Automatic On-chip Memory Minimization for Data Reuse. In *Proceedings of the International Symposium on Field-Programmable Custom Computing Machines*. IEEE, Napa, CA, 251–260.

[LLNL2014] LLNL. 2014. ROSE compiler infrastructure. (2014). `http://rosecompiler.org/`

[Magill et al.2006] Stephen Magill, A Nanevski, Edmund Clarke, and Peter Lee. 2006. Inferring invariants in separation logic for imperative list-processing programs. In *Proceedings of the Third Workshop on Semantics, Program Analysis, and Computing Environments forMemory Management (SPACE).* ACM, Charlotte, SC, 47–60.

[Meeus et al.2012] Wim Meeus, Kristof Van Beeck, Toon Goedemé, Jan Meel, and Dirk Stroobandt. 2012. An Overview of Today's High-Level Synthesis Tools. *Design Automation for Embedded Systems* (Aug. 2012), 1 – 21.

[O'Hearn et al.2001] Peter O'Hearn, John Reynolds, and Hongseok Yang. 2001. Local Reasoning About Programs That Alter Data Structures. In *Proceedings of the 15th International Workshop on Computer Science Logic (CSL '01)*. Springer-Verlag, Paris, France, 1–19.

[Pouchet et al.2013] Louis-Noel Pouchet, Peng Zhang, P Sadayappan, and Jason Cong. 2013. Polyhedral-based data reuse optimization for configurable computing. In *Proceedings of the International Symposium on Field Programmable Gate Arrays*. ACM, Monterey, CA, 29–38.

[Raza et al.2009] Mohammad Raza, Cristiano Calcagno, and Philippa Gardner. 2009. Automatic Parallelization with Separation Logic. In *Proceedings of the International Symposium on Programming Languages and Systems*. Springer-Verlag, York, UK, 348–362.

[Séméria et al.2000] Luc Séméria, Koichi Sato, and Giovanni De Micheli. 2000. Resolution of dynamic memory allocation and pointers for the behavioral synthesis form C. In *Proceedings of the Design, Automation, and Testing in Europe Conference*. ACM, Paris, France, 312–319.

[Villarreal et al.2010] Jason Villarreal, Adrian Park, Walid Najjar, and Robert Halstead. 2010. Designing Modular Hardware Accelerators in C with ROCCC 2.0. In *Proceedings of the Symposium on Field-Programmable Custom Computing Machines*. IEEE, Charlotte, NC, 127–134.

[Wilson and Lam1995] Robert P. Wilson and Monica S. Lam. 1995. Efficient context-sensitive pointer analysis for C programs. In *Proceedings of the Conference on Programming Language Design and Implementation*. ACM, La Jolla, CA, 1–12.

[Winterstein et al.2013a] Felix Winterstein, Samuel Bayliss, and George A. Constantinides. 2013a. FPGA-based K-means Clustering using Tree-based Data Structures. In *Proceedings of the International Conference on Field Programmable Logic and Applications*. IEEE, Porto, Portugal, 1–6.

[Winterstein et al.2013b] Felix Winterstein, Samuel Bayliss, and George A. Constantinides. 2013b. High-level synthesis of dynamic data structures: A case study using Vivado HLS. In *Proceedings of the International Conference on Field-Programmable Technology*. IEEE, Kyoto, Japan, 362–365.

[Winterstein et al.2014] Felix Winterstein, Samuel Bayliss, and George A Constantinides. 2014. Separation logic-assisted code transformations for efficient high-level synthesis. In *Proceedings of the International Symposium on Field-Programmable Custom Computing Machines*. IEEE, Boston, MA, 1–8.