Separation Logic-Assisted Code Transformations for Efficient High-Level Synthesis

Felix Winterstein[12]
[1]Ground Station Systems Division
European Space Agency
Robert-Bosch-Straße 5, 64293 Darmstadt, Germany
Email: f.winterstein12@imperial.ac.uk

Samuel Bayliss[2], George A. Constantinides[2]
[2]Department of Electrical and Electronic Engineering
Imperial College London
South Kensington Campus, London, SW7 2AZ
Email: g.constantinides@imperial.ac.uk

***Abstract*—The capabilities of modern FPGAs permit the mapping of increasingly complex applications into reconfigurable hardware. 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++. Applications using dynamic, pointer-based data structures and dynamic memory allocation, however, remain difficult to implement well, yet such constructs are widely used in software. Automated optimizations that aim to leverage the increased 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 non-parallelized HLS implementations, show an average latency reduction by a factor of 2.5 across our benchmarks.**

***Keywords*—FPGA; high-level synthesis; memory system; dynamic data structures; separation logic; static analysis;**

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 with specifications based on register transfer level (RTL). Examples of state-of-the-art HLS tools are Xilinx Vivado HLS, ROCCC [1] or LegUp [2], 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 [3], [4]. 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.

This paper fits in the line of work on HLS design aids that analyze the original program at compile time and perform automated code transformations. The objective is to automate parallelization and distribution of data over separate blocks of FPGA on-chip memory. Significant advancements in this direction have been made in the field of loop analyses using the *polyhedral model* which uses an algebraic representation of the iteration space of static loop nests and is applied to precisely analyze memory accesses made therein. For example, Cong *et al.* [5] implement on-chip memory bandwidth optimizations based on a dependence analysis using the polyhedral model. Recent work by Pouchet *et al.* [6] seeks to maximize loop-level parallelism using loop tiling transformations based on the model. A drawback of the polyhedral model, however, is its restricted applicability to static loop nests which iterate over linear data structures. In particular, the polyhedral model cannot analyze arbitrary memory accesses such as indirect array references or pointer accesses. Approaches for fitting the model to indirect array references [5], [7] over-conservatively assume a dependency between all program statements accessing the array. In addition, dynamic memory allocation, a powerful and widely used feature of high-level programming languages, cannot be captured. In fact, 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 this paper, 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 departure point from previous work is the use of recent advances in *separation logic* [8] which allows a formal description of the program state and reasoning about the resources accessed by a program. Separation logic extends the classical logic by an operator that

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.

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. Separation logic has mainly been leveraged in modern software verification tools. To the best of our knowledge, its application in the context of automated code optimizations for HLS remains unexplored. Our contributions are:

- We present 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, our analysis determines how to partition the heap and dynamic data structures into disjoint partitions that can be implemented in separate on-chip memory blocks.
- We implement 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.
- We demonstrate 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 II of this paper presents a motivating example and outlines the usefulness of our approach. Section III describes the theoretical background. Our parallelization and partitioning algorithm is described in Section IV followed by a description of the implementation of our tool flow in Section V. Section VI presents the case studies and results. Section VII discusses related work and Section VIII concludes the paper.

II. 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, data mining, radar tracking, image color or spectrum quantization applications. K-means clustering performs unsupervised partitioning of a D-dimensional point set $X = \{x_1, ..., x_N\}$ into clusters $S = \{S_1, ..., S_K\}$. Each cluster is represented by its geometrical center. The algorithm considered here, referred to as the *filtering algorithm* [9], uses a tree data structure (a 'kd-tree', [9]) 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) clustering implementations. Besides tree nodes, the algorithm propagates intermediate results (sets of candidate centers) through the call graph. Dynamic memory allocation (requires refactoring of the C source) enables the reuse of memory space and the on-chip heap memory can be tailored to an average-case amount which significantly reduces the required amount of on-chip memory for this application as shown in [10].

Listing 1 C-like pseudo code taken from the (modified) main kernel of the filtering algorithm [9].

```
1:  function PUSH(treeNode *u, centerSet *c, bool d, stackRecord *s)
2:      t = new stackRecord;
3:      t → n = s; t → u = u; t → c = c; t → d = d;
4:      return t;
5:  end function
6:  function POP(treeNode **u, centerSet **c, bool *d, stackRecord *s)
7:      t = s → n; *u = s → u; *c = s → c; *d = s → d;
8:      delete s; return t;
9:  end function
10: function FILTER(treeNode *root)        ▷ main kernel function
11:     c_init = new centerSet;
12:     *c_init = ... ;
13:     s = PUSH(root, c_init, true, nil);
14:     while s != nil do
15:         s = POP(&u, &c, &d, s);
16:         data = *c;
17:         data_new = f1(data);    ▷ some computation using data
18:         if d then
19:             delete c;
20:         end if
21:         c_new = new centerSet ;
22:         *c_new = data_new;
23:         b = u → left!=nil & u → right!=nil & f2(data_new);
24:         if b then                  ▷ data dependent condition b
25:             s = PUSH(u → right, c_new, true, s);
26:             s = PUSH(u → left, c_new, false, s);
27:         else
28:             delete c_new;
29:         end if
30:         delete u;
31:     end while
32: end function
```

Listing 1 shows C-like pseudo code taken from the main kernel of the iterative filtering algorithm, the only difference from [10] being that the tree traversal here is destructive. Fig. 1 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. At the beginning of the loop body, pointers to a center set and tree node are fetched from the stack (line 15), and pointers to left and right child node as well as a newly allocated center set are pushed onto the stack at the end of the loop body (line 25 and 26) - preceded by a data-dependent conditional (line 24). The kd-tree is traversed in a pre-order fashion and visited nodes are deleted (line 30).

The static program analysis presented in Section IV aims to determine the heap-carried data dependencies between loop iterations. Assuming that Fig. 1 describes the current state of the program, we can apply the following program transformations: 1) The remaining tree data structure (dark

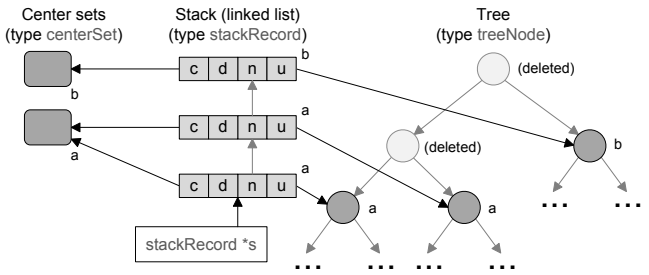

Fig. 1: Snapshot of the pointer-linked dynamic data structures accessed by the loop in Listing 1.

Listing 2 Transformed program from Listing 1.

```
1: function FILTER(treeNode *root)
2:     ... preamble (pointers access heap partitions a and b)
3:     while s_a != nil do
4:         ... loop body (pointers access heap partition a only)
5:     end while
6:     while s_b != nil do
7:         ... loop body (pointers access heap partition b only)
8:     end while
9: end function
```

gray nodes) can be split into two sub-structures (two sub-trees labelled with a, one sub-tree labelled 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, after α-renaming of the variable s, into two loops, 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, every two heap-directed pointers 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.

III. Background

We briefly describe the underlying theory. A formal introduction is beyond the scope of the paper, but is given in [8].

A. 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 use auxiliary *primed* variables which are only used in formulae (*e.g.* $z'_1 = 4 \wedge y \mapsto z'_1$ also means that y points to a heap cell with value 4, 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 single 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. 1 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)**.**

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

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 allow describing data structures without knowing their size:

Definition 2 (Example: List segment)**.**

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

i.e. there is a list segment between pointer E and F. 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)**.**

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

i.e. there is a tree pointed to by E. 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)**.**

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

i.e. there is a list segment as in (1) whose elements contain pointers 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 three examples demonstrate the ability to describe the ability to describe common data structures; automatic inference of such definitions has been demonstrated by Guo *et al.* in [11].

B. 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 expr.
$A ::=$	$x := E \mid x := [E] \mid [E] := F \mid$ new(x) $\mid$ delete(E)	atomic cmnds.
$C ::=$	$A \mid \textbf{if}\ b\ C_1\ C_2 \mid \textbf{while}\ b\ C \mid C_1; C_2$	commands

E and F are arbitrary expressions containing program variables and values (e.g. $E ::= x$, $E ::= \mathtt{nil}$, or $E ::= y + 1$). The term $[E]$ denotes pointer dereferencing of 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 [8]. 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): $\{y \mapsto x'_1\}[y] := 5\{y \mapsto 5\}$. Def. 6 specifies a triple for each atomic command of our programming language:

Definition 6 (Specifications for atomic commands [12])**.**

$\{\ x = y'_1\ \}$	$x := E$	$\{\ x = E[y'_1/x]\ \}$
$\{\ E \mapsto [\mathtt{f} : y'_1]\ \}$	$[E].\mathtt{f} := F$	$\{\ E \mapsto [\mathtt{f} : F]\ \}$
$\{\ x = y'_1 \wedge E \mapsto [\mathtt{f} : z'_1]\ \}$	$x := [E].\mathtt{f}$	$\{\ x = z'_1 \wedge E[y'_1/x] \mapsto [\mathtt{f} : z'_1]\ \}$
$\{\ emp\ \}$	$new(x)$	$\{\ x \mapsto z'_1\ \}$
$\{\ E \mapsto y'\ \}$	$delete(E)$	$\{\ emp\ \}$

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 [8].

C. 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* [13], 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. We have modified coreStar in order to include a technique called *labelled symbolic execution* [12] which assigns a unique label to Q, the spatial part of the state formula that was modified, *i.e.* $\Pi \wedge \Sigma \equiv \Pi \wedge \langle Q \rangle_{\{l \in \mathtt{Lab}\}} * F$, with Lab being the set of all labels. The technique thus propagates the 'heap footprint' of each statement through the CFG. This allows the description of 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 builds on a modified version of labelled symbolic execution in order to detect the presence of communication-free parallelism in loops.

IV. Partitioning and Parallelisation

Our analysis for memory partitioning and parallelization is hypothesis-based: We start with the hypothesis that the heap accessed by the loop iterations 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. It consists of two 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.

A. 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.*

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,

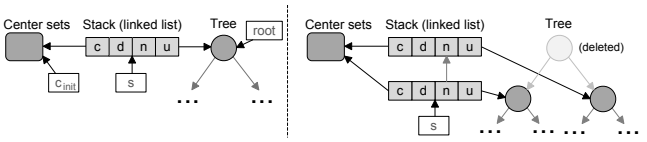

Fig. 2: Loop body pre-state before execution of the first iteration (left) and the second iteration (right).

i.e. 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 no cut-point. The goal is to obtain P cut-points in the pre-state of a loop iteration. 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:

$$s = s_0' \wedge root = u_0' \wedge c_{init} = c_0' \qquad (4)$$
$$\wedge\, tree(u_0') * c_0' \mapsto \_ * s_0' \mapsto [\texttt{u} : u_0', \texttt{c} : c_0', \texttt{n} : \text{nil}]$$

Fig. 2, left, depicts (4), 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 (4) 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 $A \equiv (4)$. 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. 2, right):

$$s = s_2' \wedge tree(u_1') * tree(u_2') * c_1' \mapsto \_ \qquad (5)$$
$$* \, s_2' \mapsto [\texttt{u} : u_1', \texttt{c} : c_1', \texttt{n} : s_1'] * s_1' \mapsto [\texttt{u} : u_2', \texttt{c} : c_1', \texttt{n} : \text{nil}]$$

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: (5) 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 (5) because of the lack of a second predicate $c_x \mapsto \_$ (the pointer aliasing is illustrated in Fig. 2, right). Hence, we abandon the inserted cut-point, peel off another loop iteration, and reach the pre-state of the third iteration:

$$s = s_4' \wedge tree(u_3') * tree(u_4') * tree(u_2') \qquad (6)$$
$$* \, c_2' \mapsto \_ * c_1' \mapsto \_ * s_4' \mapsto [\texttt{u} : u_3', \texttt{c} : c_2', \texttt{n} : s_3']$$
$$* \, s_3' \mapsto [\texttt{u} : u_4', \texttt{c} : c_2', \texttt{n} : s_1'] * s_1' \mapsto [\texttt{u} : u_2', \texttt{c} : c_1', \texttt{n} : \text{nil}]$$

The formula describes the program state shown in Fig. 1. We repeat the cut-points insertion. Our tool explores all possible cut-points assignments (there are multiple options now) and launches the proof engine 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 unrollings.

B. Proving Communication-free Parallelism

We start with (6) 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-points states' depending on whether we are currently accessing data structures 'belonging' to cut-point s (a) or s_b (b). We switch to a different cut-points state once we accessed a heaplet pointed to by a different cut-point variable. We assign label $a \in \text{Lab}$ to all heaplets accessed during execution in cut-points state a, and similarly for b. We count *pointer dereferencing* and *delete* as an access.

The parallelization goal is to partition the loop iteration space into two groups labelled a and b, and we try to establish the fact that a heaplet accessed by an iteration in cut-points state a (of group a) is never accessed by another iteration of group b. In other words, we try to prove that the separation 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 24). Hence, we perform a *fix-point calculation* for proving that the separation property is loop invariant. Our fix-point calculation adopts and modifies a technique described by Magill *et al.* [14] and works as follows:

1. Start with the pre-state of the loop M_0^{pre} equal to (6) with cutpoints s_a and s_b inserted as described above.
2. Symbolically execute $\{M_i^{pre} \wedge b\}$ '*loop body*' $\{M_{i+1}^{post}\}$, b is the loop condition, i is the iterations counter. We attach labels to heaplets as described above. If we find both labels a and b on a heaplet, we abort and report a failed proof.
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 $M_{i+1,folded}^{post} \equiv ls(s_a, \texttt{nil})$. The folding prevents accumulating singleton heaplets such as $s_a \mapsto [\texttt{n} : x_1']$ during repeated execution of the loop body which is required for convergence of the fix-point calculation.
4. The fix-point calculation terminates if M_{i+1}^{post} implies a post-state of one of the previous iterations $M_k^{post}, k = 0, ..., i$. Formally, we ask coreStar's theorem prover to decide $M_{i+1}^{post} \Rightarrow \bigcup_{k=0..i} M_k^{post}$ (the right hand side is the disjunction of all previous post-states). If the implication does not hold we set $M_{i+1}^{pre} = M_{i+1}^{post}$ and continue with step 2).

Note that, for another candidate for the cut-points 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 $\langle c_2' \mapsto \_ \rangle_{\{a,b\}}$. A successful fix-point calculation tells us *that* it is possible to partition the heap given the assignment of cut-points, but it does not tell us *how* we would partition (6). This is because, for instance, we may lose track of the predicate $c_2' \mapsto \_$ as it will be disposed (Listing 1, line 19) during the course of fix-point calculation before we even access $c_1' \mapsto \_$ for the first time. We can solve this issue by recording each heaplet at the time it gets first assigned a label. At the end of the analysis, we stitch together

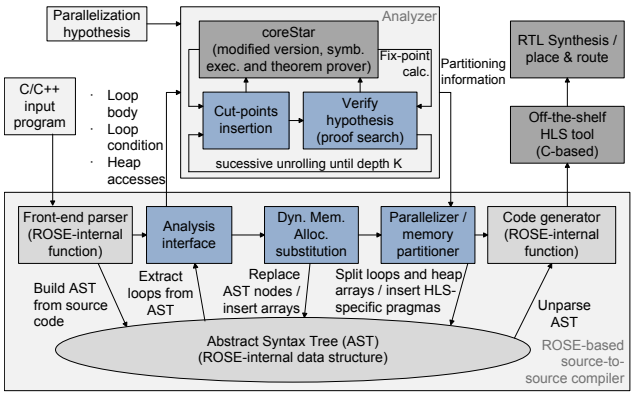

Fig. 3: CAD flow including the heap analyzer, source translator and third party tools for HLS and RTL implementation.

all snapshots resulting in a labelled version of (6):

$$
\begin{aligned}
& s_a = s'_4 \wedge s_b = s'_1 \wedge \langle tree(u'_3)\rangle_{\{a\}} * \langle tree(u'_4)\rangle_{\{a\}} \qquad (7)\\
& * \langle tree(u'_2)\rangle_{\{b\}} * \langle c'_2 \mapsto \_\rangle_{\{a\}} * \langle c'_1 \mapsto \_\rangle_{\{b\}}\\
& * \langle s'_4 \mapsto [\mathtt{u} : u'_3, \mathtt{c} : c'_2, \mathtt{n} : s'_3]\rangle_{\{a\}}\\
& * \langle s'_3 \mapsto [\mathtt{u} : u'_4, \mathtt{c} : c'_2, \mathtt{n} : s'_1]\rangle_{\{a\}}\\
& * \langle s'_1 \mapsto [\mathtt{u} : u'_2, \mathtt{c} : c'_1, \mathtt{n} : \mathrm{nil}]\rangle_{\{b\}}
\end{aligned}
$$

(7) tells us that the heap, at this point, can be partitioned into two disjoint regions labelled a and b. Furthermore, the successful fix-point calculation 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.

The above analysis provides both memory partitioning information (by labels assigned to heaplets) and the legality of parallelization (by a successful fix-point calculation). Next, we explain how this information is used in a source-to-source translator for automated parallelization and partitioning.

V. Implementation

Our tool flow consists of three parts: a module for the heap analysis, a source-to-source compiler, and a back-end HLS and FPGA synthesis tool. Fig. 3 shows the complete tool flow.

A. 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 version of coreStar [13] which we extended to include labelled symbolic execution and cut-points 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 ∼60 logic rules 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.

Listing 3 Substituting pointer accesses to heap-allocated data.

```
1: // ... substitution of u → d = d_0;
2: // ... u is the new pointer variable, 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_0;
```

B. Source-to-source Compiler

Our source translator is built on the ROSE source compiler infrastructure [15]. The ROSE infrastructure provides a library of C++ functions for source code analysis and transformation which we use to build our customized translator. 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 [10]. 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 which addresses the problem of determining a bound on the amount of allocated heap memory. Cook *et al.* [16] 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* (DEF-USE) 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. 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.

VI. Case Studies

We test the tool flow in Fig. 3 using three C++ implementations taken from real-world applications. We use Xilinx Vivado HLS 2012.2 as a back-end HLS tool and Xilinx Vivado 2012.2 (RTL flow) 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.

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. The tree deletion is a common operation for example in the destructor function of C++ classes which use the tree structure.

Filter - This is the motivating example in Section II which is taken from the direct implementation of the filtering algorithm for efficient K-means clustering [9]. 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.

The target device is a Virtex 7 FPGA (7vx485t-1) and all results are taken from placed and routed designs. We report resource utilization in slices, DSP slices (DSP) and 36K-Block RAMs (BRM). We also report the achieved clock speed 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 I shows the implementation results for two cases: The first shows the implementation if the tool only ensures synthesizability (substituting dynamic memory allocation) without parallelization (base line), which does not require the separation logic-based heap analysis. The second row shows implementation results if the tool flow uses the heap analyzer for memory partitioning and parallelization (automatic parallelization, degree P), an optimization that cannot be done by Vivado HLS itself as shown in [10]. For the benchmarks **Merger** and **Filter**, we add two reference designs for comparison: hand-optimized HLS designs using Vivado HLS and hand-written RTL designs in VHDL. The ratio R relates the cycle count of the automatically parallelized benchmarks to that of the base line and hand-crafted designs. Our 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 the filtering algorithm and the tree deletion which enables successful parallelization ($R \approx 1.8$ 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 V-B) which explains the resource overhead compared to the base case. Comparing resources, clock frequency and cycle count, however, we observe further improvements obtained from manual source code refactoring: In the hand-optimized HLS design of the filtering algorithm, we manually flattened loop nests in order to enable efficient pipelining [10], an optimization beyond the scope of this paper. Our tool currently uses standard C data types as opposed to bit width customizations of data items and pointers in the hand-written HLS and RTL implementations which results in increased memory usage and more expensive arithmetic operations compared to the customized case.

TABLE I: Implementation results and comparison.

	P	Slices	DSP	BRM	Clock	Cycles	R
Merger (4×2048 random input key-value pairs)							
Base line (no par.)	1	574	0	96	9.0 ns	21167k	1.0
Autom. paralleliztn.	4	965	0	96	8.7 ns	5483k	**3.9**
Hand-optim. HLS	4	764	0	60	8.3 ns	3407k	6.2
Hand-crafted RTL	4	932	0	52	5.0 ns	2197k	9.6
Tree Deletion (16383 tree nodes)							
Base line (no par.)	1	1521	6	241	5.2 ns	901k	1.0
Autom. paralleliztn.	2	4069	18	245	6.0 ns	487k	**1.8**
Filter (16384 3-dim. data points, 32767 kd-tree nodes, $K = 128$ cluster centers)							
Base line (no par.)	1	2736	17	432	6.1 ns	957k	1.0
Autom. paralleliztn.	2	6734	68	439	7.0 ns	527k	**1.8**
Hand-optim. HLS	2	5492	38	280	5.5 ns	165k	5.8
Hand-crafted RTL	2	6449	40	236	5.0 ns	54k	17.7

TABLE II: Running time of the fix-point computation.

(Intel i7-3770 CPU, 3.40GHz, 16GB)	Merger	Tree Deletion	Filter
No. of fix-point iterations	5	3	7
Running time	195.8 s	1.1 s	109.7 s

The fix-point computation of the heap analyzer dominates the tool running time. It increases significantly if the analyzed loop body contains complicated control flow and if many fix-point iterations are required. The **Merger** benchmark contains nested *if-else* constructs and sub-loops and has the longest running time. Table II shows the timing results.

VII. Related Work

A large body of work on advanced HLS techniques for parallelization and memory partitioning is based on the polyhedral model [5], [6], which is among the most popular and powerful abstractions for the analysis of loops to date, but which is not

applicable to our benchmarks.
Notable is the work by Ghiya and Hendren [17] in that a pointer analysis is used to establish disjointness of heap-allocated 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. The other major difference, of course, is that our work targets HLS for FPGAs which allows us to build a customized distributed memory architecture based on the heap access analysis.
We build on the work by Raza *et al.* [12] which provides the theoretical foundation for our tool as described in Section III. The work in [18] 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 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.* [19] 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.

VIII. Conclusion

We present a proof of concept tool flow that automatically parallelizes loops in pointer-manipulating C/C++ programs and distributes dynamically 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 three 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 $1.8\times - 3.9\times$ in terms of cycle count compared to the non-parallelized implementations.

A. 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 automize the translation leveraging the LLVM [20] 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 of comparing partitioning alternatives, which we want to address in future work. We also plan to extend the analysis and the tool to automatically insert on-chip buffers in the interface to an external memory.

References

[1] "ROCCC 2.0 — Jacquard Computing." [Online]. Available: http://www.jacquardcomputing.com/roccc/

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

[3] W. Meeus, K. Van Beeck, T. Goedemé, J. Meel, and D. Stroobandt, "An Overview of Todays High-Level Synthesis Tools," *Design Automation for Embedded Systems*, pp. 1 – 21, Aug. 2012.

[4] BDTI, "An Independent Evaluation of: The AutoESL AutoPilot High-Level Synthesis Tool," 2010. [Online]. Available: http://www.bdti.com/Resources/BenchmarkResults/HLSTCP/AutoPilot

[5] J. Cong, W. Jiang, B. Liu, and Y. Zou, "Automatic memory partitioning and scheduling for throughput and power optimization," *ACM Trans. on Design Autom. of Electron. Syst.*, vol. 16, no. 2, pp. 1–25, Mar. 2011.

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

[7] M. Benabderrahmane, L. Pouchet, A. Cohen, and C. Bastoul, "The polyhedral model is more widely applicable than you think," in *Proc. of the Int. Conf. on Compiler Construction*, 2010, pp. 283–303.

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

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

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

[11] B. Guo, N. Vachharajani, and D. I. August, "Shape analysis with inductive recursion synthesis," *ACM SIGPLAN Notices*, vol. 42, no. 6, p. 256, Jun. 2007.

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

[13] M. Botinčan, D. Distefano, M. Dodds, R. Grigore, and M. J. Parkinson, "coreStar: the core of jStar," *Boogie*, pp. 65–77, 2011.

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

[15] "ROSE compiler infrastructure." [Online]. Available: http://rosecompiler.org/

[16] B. Cook, A. Gupta, S. Magill, A. Rybalchenko, J. Simsa, S. Singh, and V. Vafeiadis, "Finding heap-bounds for hardware synthesis," in *Formal Methods in Computer-Aided Design*, 2009, pp. 205–212.

[17] R. Ghiya, L. Hendren, and Y. Zhu, "Detecting parallelism in C programs with recursive data structures," *Compiler Construction*, 1998.

[18] B. Cook, S. Magill, M. Raza, J. Simsa, and S. Singh, "Making fast hardware with separation logic," 2010, unpublished. [Online]. Available: http://www.cs.cmu.edu/ smagill/papers/fast-hardware.pdf

[19] M. Botinčan, M. Dodds, and S. Jagannathan, "Proof-directed parallelization synthesis by separation logic," *ACM Trans. on Programming Languages and Systems*, vol. 35, no. 2, pp. 1–60, Jul. 2013.

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