Improving High-Level Synthesis with Decoupled Data Structure Optimization

Ritchie Zhao, Gai Liu, Shreesha Srinath, Christopher Batten, Zhiru Zhang
School of Electrical and Computer Engineering, Cornell University, Ithaca, NY
{rz252, gl387, ss2873, cbatten, zhiruz}@cornell.edu

Abstract
Existing high-level synthesis (HLS) tools are mostly effective on algorithm-dominated programs that only use primitive data structures such as fixed-size arrays and queues. However, many widely used data structures such as priority queues, heaps, and trees feature complex member methods with data-dependent work and irregular memory access patterns. These methods can be inlined to their call sites, but this does not address the aforementioned issues and may further complicate conventional HLS optimizations, resulting in a low-performance hardware implementation. To overcome this deficiency, we propose a novel HLS architectural template in which complex data structures are decoupled from the algorithm using a latency-insensitive interface. This enables overlapped execution of the algorithm and data structure methods, as well as parallel and out-of-order execution of independent methods on multiple decoupled lanes. Experimental results across a variety of real-life benchmarks show our approach is capable of achieving very promising speedups without causing significant area overhead.

1. Introduction
Over the past decade, the benefits provided by traditional technology scaling has gradually diminished due to challenges with power consumption and physical design at the newest technology nodes. As a result, general-purpose processors and software are no longer seen as a sustainable solutions for future computing needs. Engineers and researchers are increasingly exploring specialized hardware accelerators in order to obtain the performance and energy efficiency necessary for applications which traditionally lay in the software-only domain. High-level synthesis (HLS) is a key enabler of this trend, allowing designers to automatically synthesize hardware from high-level specifications written in a software programming language and making microarchitectural optimizations (such as pipelining or memory banking) accessible in the form of pragmas. HLS design methodologies can markedly reduce the development effort and cost of creating specialized hardware compared to register-transfer level (RTL) flows [5].

A common design pattern in software engineering is to separate a program into algorithms and data structures. To maximize the performance of an application, both parts must be well-optimized; for instance, achieving the best asymptotic runtime of Dijkstra’s algorithm requires a priority queue with efficient push and pop methods. To hide design complexity, software programmers often leverage readily available libraries of commonly used data structures which offer a set of carefully crafted methods to achieve high performance and code reusability.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.

DAC ’16, June 05-09, 2016, Austin, TX, USA
2016 ACM. ISBN 978-1-4503-4236-0/16/06$15.00
DOI: http://dx.doi.org/10.1145/2897937.2898030

void bintree::insert (key, value) {
    NodeType curr = root;
    // data dep. variable-bound loop
    while (curr kk !curr.insert_at(key)) {
        curr = curr.choose_child(key);
    }
}

NodeType bintree::find (key) {
    NodeType cur = root;
    // data dep. variable-bound loop
    while (curr kk curr->key != key) {
        curr = curr.choose_child(key);
    }
    return curr;
}

Figure 1: Self-balancing binary tree methods – (a) insert adds a key/value pair to the tree, then rebalances it; (b) find returns the node containing the input key or NULL if not found. For brevity, we use the function choose_child to choose the left or right child based on the input key, and insert_at to determine whether to insert at the current node based on the input key. insert_node performs the actual insert operation.

However, current HLS tools primarily focus on algorithm-dominated programs that only utilize primitive data structures such as fixed-size arrays or queues (e.g., DSP applications). These primitive data structures typically only provide very simple read and write methods, which can easily be inlined to the main algorithm without complicating any of the key HLS optimizations such as pipelining and unrolling. In contrast, more complex data structures (e.g., hash tables, priority queues, and trees) usually contain methods that exhibit data-dependent work and irregular memory access patterns. For these methods, inlining does not significantly improve the generated hardware.

For example, Figure 1 shows the insert and find methods of a self-balancing binary tree. The code snippets show that both methods contain a variable-bound loop, which complicates unrolling. In addition, the insert method mutates the tree in nontrivial ways in the rebalancing step, and to avoid violation of data dependences we cannot easily overlap other invocations of insert with itself or find. These properties make it very difficult to optimize any HLS code utilizing the binary tree structure. Traditional HLS techniques are typically unable to extract coarse-grained parallelism across these methods calls invoked within a sequential monolithic program, resulting in poorly optimized hardware. This has prevented complex data structures from gaining widespread use within the HLS programming space, even though current tools make it possible to write synthesizable code for such objects.

To address the challenges of complex data structure synthesis, we propose to decouple the data structure methods from the algo-
structure is then defined as one whose key methods are complex. We first define a complex data structure method as one which is (1) long latency and/or variable latency, (2) difficult to pipeline due to variable-bound loops and/or memory dependences. A complex data structure is then defined as one whose key methods are complex.

In this section, we introduce several important concepts and terms that will be used in the subsequent discussions.

2.1 Complex Data Structures in HLS
We first define a complex data structure method as one which is (1) long latency and/or variable latency, (2) difficult to pipeline due to variable-bound loops and/or memory dependences. A complex data structure is then defined as one whose key methods are complex.

The rest of this paper is organized as follows: Section 2 provides an overview of complex data structures and methods; Section 3 presents our methodology and the architectural template for specialized container units. We report experimental results for various data structures in Section 5, discuss related works in Section 6, and conclude the paper in Section 7.

2. Preliminaries
In this section, we introduce several important concepts and terms that will be used in the subsequent discussions.

2.2 Accessor and Mutator Methods
It is useful to classify various data structure methods based on certain properties which affect how they may be optimized during synthesis. We identify two broad classes of methods, borrowing terminology used in object-oriented software programming:

- **Accessor Methods** — methods that only read and never write to the data structure, and can thus be executed in parallel and/or out-of-order with respect to other accessor methods.
- **Mutator Methods** — methods that change the data structure and must be executed serially and in-order with respect to other (even non-mutator) methods.

In this study, we assume that complex mutator methods cannot be overlapped with other methods, since they modify memory and can create dependences (e.g., read-after-write). A sophisticated compiler or expert designer may be able to guarantee safety for the concurrent execution of a mutator and other methods, in which case our architecture can be extended to handle such calls.
of the mutator method as the mutator executions can modify the un-

Note, that a given M-SMU is a single instance processor method calls. Each mutator method is synthesized to create a degree of freedom to the designer for exploiting the parallelism across ac-

can be replicated (multi-lane). This provides a significant degree of freedom to the designer for exploiting the parallelism across accessor or mutator methods using two separate channels. Note, that some mutator methods do not return any values, in which case it will not be connected to the collector; instead the dispatcher will send a response message back to the algorithm unit.

The memory ports of each SMU are connected via a crossbar to a multi-port memory that stores the data members. A simple hardware arbiter controls which memory requests are served during each cycle between accessors. There is no need for arbitration between mutators in our scheme as only one will ever be active at any time. This arbitrated memory interface allows our SCU template to tolerate across different types of storage with varying number of memory ports. Although we focus on on-chip memories in this study, we note that with a decoupled memory interface it would not be difficult to extend our architectural template to utilize an external memory hierarchy, as shown in Figure 3.

It is important to note that only the complex methods of a data structure are decoupled in this fashion. Simple, constant-time methods such as \texttt{size} are synthesized in the conventional manner. This requires us to keep a copy of certain data in the algorithm unit, but the overhead of doing so is small. Similarly, some methods like \texttt{top} from the priority queue can be serviced in the dispatcher without needing to invoke a decoupled method unit.

3. Decoupled Data Structure Optimization

To address the challenges facing complex data structure synthesis, we propose to synthesize SCUs based on an architectural template. The design of the SCU is based on two key ideas. Firstly, we use compute-access decoupling to hide the latency of complex data structure methods from the algorithm and to allow modular optimization of the algorithm and the data structure. Secondly, we utilize hardware replication to exploit coarse-grain parallelism and possible conditional method calls.

3.1 SCU Architectural Template

The SCU architecture is shown in Figure 3, and composes of several distinct modules connected using latency-insensitive message-based interfaces. The dispatcher is responsible for receiving method calls from the algorithm in the form of a message that contains the opcode to indicate the requested method and the argument values based on the method type. The precise message format widths of each field is application specific. In some cases an algorithm uses only one method, in which case the opcode can be omitted entirely. The dispatcher is responsible for decoding messages and dispatching work to a specialized method unit (SMU) that executes the accessor or mutator methods using two separate channels. Note, we use separate channels for accessor and mutator methods as (1) it is possible to have multiple outstanding accessor calls executing in parallel, and (2) the message formats are usually quite different between the two method classes. There are two groups of SMUs: the mutator specialized method units (M-SMUs) that support mutator calls and the accessor specialized method units (A-SMUs) that support accessor calls. The dispatcher preserves dependencies between the mutator and accessor calls by ensuring that no other methods are launched while an M-SMU is active. However, the architectural template provides the freedom to a designer or compiler to relax this constraint if memory safety can be guaranteed during concurrent method executions.

Each accessor method in the data structure is synthesized to create an A-SMU. A-SMUs can be configured with a single instance of the synthesized method (single-lane) or the synthesized method can be replicated (multi-lane). This provides a significant degree of freedom to the designer for exploiting the parallelism across accessor method calls. Each mutator method is synthesized to create a single-lane M-SMU. Note, that a given M-SMU is a single instance of the mutator method as the mutator executions can modify the underlying data-structure and multiple mutator executions cannot be overlapped. The final module is the collector, which is responsible for collecting results from the A-SMUs and the M-SMUs and returning them to the algorithm in the original method call order. Note that some mutator methods do not return any values, in which case it will not be connected to the collector; instead the dispatcher will send a response message back to the algorithm unit.

The memory ports of each SMU are connected via a crossbar to a multi-port memory that stores the data members. A simple hardware arbiter controls which memory requests are served during each cycle between accessors. There is no need for arbitration between mutators in our scheme as only one will ever be active at any time. This arbitrated memory interface allows our SCU template to tolerant across different types of storage with varying number of memory ports. Although we focus on on-chip memories in this study, we note that with a decoupled memory interface it would not be difficult to extend our architectural template to utilize an external memory hierarchy, as shown in Figure 3.

It is important to note that only the complex methods of a data structure are decoupled in this fashion. Simple, constant-time methods such as \texttt{size} are synthesized in the conventional manner. This requires us to keep a copy of certain data in the algorithm unit, but the overhead of doing so is small. Similarly, some methods like \texttt{top} from the priority queue can be serviced in the dispatcher without needing to invoke a decoupled method unit.

3.2 Overlapped Execution with Mutator Methods

Figure 4(a) shows code for the inner loop of Dijkstra’s Algorithm; the code attempts to update the distance to each neighbor \( v \) of a node \( u \). If successful, it pushes \( v \) onto a min priority queue keyed by the distance. Figure 4(b) shows the baseline execution without decoupling. The algorithm must stall each time it calls \texttt{push}, and the method unit is idle while the algorithm runs. Overall application performance is low due to poor hardware utilization.

Using decoupling, we can take advantage of the fact that \texttt{push} is a method which does not return any values. After receiving a \texttt{push} request, the dispatcher immediately returns a response to the algorithm at the same time it activates the appropriate M-SMU. The algorithm can then proceed to the next iteration, which is executed concurrently with the \texttt{push} call from the previous iteration as seen in Figure 4(c). This greatly reduces stalling on both hardware modules. Figure 4(c) also illustrates how our approach exploits
conditional method calls in the loop. An iteration of Dijkstra only
calls push if it successfully relaxes the best-known distance, and
in some cases we can process multiple algorithm iterations in the
period it takes to execute a single call to push.

Decoupling also facilitates the pipelining of the algorithm loop.
Typically, for current HLS tools to effectively pipeline an outer
loop, the inner loops must be unrolled [17], which cannot be done
for the variable-bound while loop in the push method. After de-
coupling, the method call is replaced with simple operations to read
and write the latency-insensitive interface. Figure 4(d) shows the
execution of the program after pipelining. The SMU is saturated
and its throughput now dictates the performance of the application.

of a method request, the result is written to the appropriate entry
in the ROB and that entry is marked valid. Different lanes execute
independently and can finish in any order. Results are only returned
from the head of the ROB when it is valid. This ensures results are
produced by the SCU in the original call order.

Figure 5(c) shows the execution of KeySearch on three lanes.
The algorithm is pipelined, though the long latency of the method in
iteration k=2 causes some stalling. Iterations that reuse lanes (e.g.,
k=0 and k=3) improve the hardware utilization. We also see that
the k=3 call finishes before k=2, in which case its return value is
stored in the ROB until its predecessors have also finished and their
values returned to the algorithm. With sufficient lanes, the latency
of the method calls can be hidden from the algorithm and the total
throughput increases significantly.

3.4 Dynamic Memory Allocation

Our decoupled architecture also enables the elegant integration of
a dynamic memory allocator which allows data structures to call
malloc and free to alter their storage size at runtime. Such self-
sizing structures carry considerable benefits in ease of use, as their
sizes do not need to be statically known. Multiple applications or
data structures drawing from the same pool of memory will also
tend to be more memory efficient in the average case. However, the
need for malloc and free to interact with a free list (or some other
global record of free segments) makes them behave as complex
mutator methods. The SCU approach is helpful in this case because
the optimizations which apply to mutators also benefit any data
structure which makes use of dynamic memory allocation.

4. Implementation

The synthesis of an SCU can be automated using code transforma-
tions combined with parameterized hardware generators. The first
step in SCU synthesis is to identify which methods are to be decou-
pled as well as whether each method is an accessor or mutator. This
is easily done using a compiler pass to detect variable-bound loops
and the presence of memory writes. The request and response mes-
sage formats can then be inferred from each method’s arguments
and return type. After this a source-to-source transform strips the
code from the methods to be decoupled and replaces them with in-
structions to send and receive SCU messages. The method bodies
are then used to create standalone functions which can be pushed
through the HLS flow to generate hardware modules. This allows a
data structure to be converted to a SCU with only minor program-
mer effort and no change to the algorithm code.

```c
1 void KeySearch (keys, vals) {
2    // algorithm loop
3    for (k : keys) {
4        // returns matching bucket node
5        HashTableNode n;
6        n = H.find(k);
7        // check if key was found
8        if (n ! = NULL) {
9            vals[i] = n.value;
10        }
11    }
12 }
13 }
```

(a) KeySearch kernel

Figure 5: Execution of a kernel containing a complex accessor method – (a) Code for the KeySearch kernel, which searches a hash table
for a list of keys using the find method and contains a variable-bound while loop; (b) Baseline execution with no optimization, the algorithm
cannot be pipelined; (c) Decoupled execution on parallel lanes and pipelining the algorithm greatly improves method throughput.

![Figure 6: Multi-lane SMU example](image-url)

- The find method in a hash table can be parallelized using a multi-lane A-SMU. The host SCU
must contain a Request Queue and Reorder Buffer to ensure correct
operation. The SCU’s local memory is not shown here.

3.3 Parallel Execution of Accessor Methods

The previous example showcases optimizations which can be made
to a mutator method, but accessors receive the further benefit of
parallel and out-of-order execution across multiple lanes. Figure
5(a) shows the KeySearch kernel, which iteratively finds keys
inside a hash table. As before, the find method cannot be pipelined
and the baseline execution is often stalled (see Figure 5(b)).

Our approach instead generates an SCU for the hash table as
shown in Figure 6, which includes a multi-lane accessor SMU. In
this architecture, each incoming method call is given an ID and
placed into a request queue (RQ) in the dispatcher to wait for an
available lane. Each call added to the queue also reserves an entry
at the tail of the reorder buffer (ROB). On each cycle, the head
request of the RQ checks each SMU lane in a round-robin fashion,
dispatching to the lane if it is idle. When a lane completes execution

3.4 Dynamic Memory Allocation

Our decoupled architecture also enables the elegant integration of
dynamic memory allocators which allows data structures to call
malloc and free to alter their storage size at runtime. Such self-
sizing structures carry considerable benefits in ease of use, as their
sizes do not need to be statically known. Multiple applications or
data structures drawing from the same pool of memory will also
tend to be more memory efficient in the average case. However, the
need for malloc and free to interact with a free list (or some other
global record of free segments) makes them behave as complex
mutator methods. The SCU approach is helpful in this case because
the optimizations which apply to mutators also benefit any data
structure which makes use of dynamic memory allocation.

4. Implementation

The synthesis of an SCU can be automated using code transforma-
tions combined with parameterized hardware generators. The first
step in SCU synthesis is to identify which methods are to be decou-
pled as well as whether each method is an accessor or mutator. This
is easily done using a compiler pass to detect variable-bound loops
and the presence of memory writes. The request and response mes-
sage formats can then be inferred from each method’s arguments
and return type. After this a source-to-source transform strips the
code from the methods to be decoupled and replaces them with in-
structions to send and receive SCU messages. The method bodies
are then used to create standalone functions which can be pushed
through the HLS flow to generate hardware modules. This allows a
data structure to be converted to a SCU with only minor program-
mer effort and no change to the algorithm code.
The next step is to determine the number of parallel lanes to synthesize for each accessor method. The optimal number depends on the throughput of the algorithm loop, the average latency of a single method call, as well as the memory throughput required by each lane. Creating more lanes is not useful if the algorithm cannot fill them with calls or if memory bandwidth is saturated. Currently, we require the designer to explore this design space and indicate the desired number of lanes using pragmas. With the above information the dispatcher, collector, and memory interface (including crossbar and arbiter) can all be generated from parameterized templates.

## 5. Experimental Results

To evaluate our techniques, we used a state-of-the-art commercial HLS tool and Vivado to implement the synthesized RTL, targeting the Xilinx Virtex-7 FPGA. The baseline designs were simply pushed through this flow from C++ code to the final implementation. In contrast DigitrecKnn sees much less improvement due to the fact that the workload is unbalanced towards the SCU, causing reduced benefit for execution overlapping. In two of the designs the FF usage is lower in the SCU design, which can be attributed to more aggressive operation chaining caused by pipelining. "ImgSeg and KeySearch are both kernels which do most of their work inside the method call. Performance improvement comes from parallel execution of the accessor method. In both cases although there is speedup, there is also non-trivial area overhead (1.93x LUTs and 1.84x FFs for the decision tree, 1.43x LUTs and 1.22x FFs for the decision tree). This is due to modules necessary for managing parallel lanes in the SCU, which become more complex going from a 1-lane to 2-lane design. Table 2 gives an area breakdown of each lane. Table 2 shows the scalability of our approach up to 4 lanes. FF and LUT usage grows very slowly, and BRAM usage stays constant after 2 lanes as the increase from 1 to 2 is used for the ROB. Meanwhile, performance improves linearly as more lanes are added despite the limited memory bandwidth of 2 read ports in our design. This is possible because each lane does not fully saturate a single port’s memory bandwidth, a key property leveraged by our arbitrated memory interface.

### Table 1: Latency and resource usage comparison for each benchmark — Target clock period is 5 ns. Methods = which methods were specialized in the SCU; LAT = latency; CP = achieved clock period; SLICE = # of slices; LUT = # of look-up tables; FF = # of flip-flops; BRAM = # of block rams; DSP = # of DSPs; Speedup = improvement in latency relative to baseline.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Data Structure</th>
<th>Methods</th>
<th>LAT</th>
<th>CP</th>
<th>LUT</th>
<th>FF</th>
<th>BRAM</th>
<th>DSP</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dijkstra-Base</td>
<td>Priority Queue</td>
<td>push, pop</td>
<td>4006</td>
<td>4.36</td>
<td>640</td>
<td>884</td>
<td>2</td>
<td>4</td>
<td>1.82x</td>
</tr>
<tr>
<td>Dijkstra-SCU</td>
<td></td>
<td>empty</td>
<td>2207</td>
<td>4.38</td>
<td>750</td>
<td>850</td>
<td>3</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>DigitrecKnn-Base</td>
<td>Priority Queue</td>
<td>push, pop</td>
<td>2732</td>
<td>4.66</td>
<td>467</td>
<td>635</td>
<td>2</td>
<td>0</td>
<td>1.17x</td>
</tr>
<tr>
<td>DigitrecKnn-SCU</td>
<td></td>
<td>empty</td>
<td>2331</td>
<td>4.70</td>
<td>583</td>
<td>586</td>
<td>2</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>DFS-Base</td>
<td>Dynamically-</td>
<td>push, pop</td>
<td>2851</td>
<td>4.12</td>
<td>1157</td>
<td>1798</td>
<td>2</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>DFS-SCU</td>
<td>Sized Stack</td>
<td>empty</td>
<td>1615</td>
<td>4.63</td>
<td>1263</td>
<td>1873</td>
<td>2</td>
<td>3</td>
<td>1.77x</td>
</tr>
<tr>
<td>ImgSeg-Base</td>
<td>Random</td>
<td>classify</td>
<td>17007</td>
<td>4.69</td>
<td>226</td>
<td>181</td>
<td>5</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>ImgSeg-SCU</td>
<td>Decision Tree</td>
<td></td>
<td>8506</td>
<td>4.95</td>
<td>437</td>
<td>333</td>
<td>7</td>
<td>0</td>
<td>2.00x</td>
</tr>
<tr>
<td>KeySearch-Base</td>
<td>Hash Table</td>
<td>find</td>
<td>21281</td>
<td>4.05</td>
<td>4533</td>
<td>3462</td>
<td>12</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>KeySearch-SCU</td>
<td></td>
<td></td>
<td>13592</td>
<td>4.49</td>
<td>6527</td>
<td>4207</td>
<td>14</td>
<td>0</td>
<td>1.57x</td>
</tr>
</tbody>
</table>

### Table 2: Area breakdown of the KeySearch SCU (2 lanes).

<table>
<thead>
<tr>
<th>KeySearch-SCU</th>
<th>FF</th>
<th>LUT</th>
<th>BRAM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dispatcher</td>
<td>19%</td>
<td>26%</td>
<td>0</td>
</tr>
<tr>
<td>Collector</td>
<td>25%</td>
<td>25%</td>
<td>2</td>
</tr>
<tr>
<td>SMU #1</td>
<td>14%</td>
<td>7%</td>
<td>0</td>
</tr>
<tr>
<td>SMU #2</td>
<td>14%</td>
<td>10%</td>
<td>0</td>
</tr>
<tr>
<td>Mem Interface</td>
<td>28%</td>
<td>32%</td>
<td>12</td>
</tr>
</tbody>
</table>
6. Related Work

Several recent studies have investigated the benefits of specialized data structure accelerators to eliminate memory accesses and improve the performance of data and instruction caches [16]. Hardware implementations of some data structures have been reported including linked lists [18], queues [1, 7], trees [11, 2], and other pointer-based structures [12, 15]. Our work focuses on synthesizing container units for a broader range of complex data structures, instead of crafting the most efficient implementation of a specific structure. Nevertheless, many of the above techniques (e.g., customized prefetching) are complementary to our work.

Loew et al. also studied data structure co-processing in the form of decoupling and offloading complex methods [9]. Their implementation relies on CPU multithreading, as opposed to using specialized hardware. In several cases the authors observed a slowdown due to inter-thread communication latency.

Cheng and Wawrzynek proposed generating pipelines with decoupled stages for memory operations to improve memory throughput and tolerate access latency [4]. However, their work targets raw memory loads and stores to improve pipeline throughput, while we focus on improving the synthesis of complex data structure methods using decoupling and managed parallel lanes.

The implementation of a dynamic memory allocator builds on similar systems such as DMM-HLS [6]. Our contribution is to show how memory management functions can be integrated with our approach and optimized as part of data structure methods.

Cattaneo et al. used compute-access decoupling and polyhedral analysis to optimize loop tile sizes for memory reuse [3]. CoRAM++ is a recent work on data structure-specific memory interface modules for transferring data to FPGA from DRAM [14], which focuses mostly on optimizing traversals on multi-dimensional arrays and linked lists from off-chip memory. In comparison to both works, we study more complex data structures and methods and use on-chip memory management.

The idea of using a decoupled multi-lane accelerator for parallelism extraction bears similarity to ElasticFlow [13]. However, ElasticFlow targets irregular loop nests, while our techniques target coarse-grained parallelism across independent data structure method calls. We also demonstrate speedup for mutator methods due to decoupling even without the multi-lane architecture.

7. Conclusions and Future Work

We propose a novel methodology and architectural template for the HLS of complex data structures. By decoupling complex methods from the algorithm, our approach is capable of exploiting coarse-grained parallelism between the algorithm and method call via overlapped execution. For accessor methods, we can further improve performance via parallel and out-of-order execution across multiple lanes. We evaluate our approach by implementing four different complex data structures and obtain promising speedups. Future work includes integrating SCUs with a memory hierarchy for accessing off-chip memories and using intelligent static analysis to identify opportunities to parallelize mutator methods.

Acknowledgements

This work was supported in part by NSF Awards #1149464, #1337240, #1453378, #1512937, a DARPA Young Faculty Award, and donations from Intel Corporation and Xilinx, Inc.

References