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.

DAC '16, June 05-09, 2016, Austin, TX, USA
2016 ACM. ISBN 978-1-4503-4236-0/16/60$15.0
DOI: http://dx.doi.org/10.1145/2897937.2898030

```
1  void bintree::insert (key, value) {
2    NodeType curr = root;
3    // data dep. variable-bound loop
4    while (curr && !curr.insert_at(key)) {
5      curr = curr.choose_child(key);
6    }                                              (a)
7
8    // modify the tree structure
9    insert_node(curr, key, value);
10   rebalance_tree();
11 }
```

```
1  NodeType bintree::find (key) {
2    NodeType curr = root;
3    // data dep. variable-bound loop
4    while (curr && curr->key != key) {
5      curr = curr.choose_child(key);               (b)
6    }
7
8    return curr;
9  }
```

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 method 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-

```
1  void
2  priority_queue::heap_up (i) {
3    while (i != 0) {
4      // j is i's parent node
5      j = (i-1) / 2;
6
7      if (heap[i] < heap[j])
8        swap(heap[i], heap[j]);
9      else break;
10
11     i = j;
12 } }
```

(a) heap_up

```
1  NodeType
2  hash_table::find (k) {
3    i = hash_func(k);
4    NodeType n = hashtbl[i];
5
6    while (n && n->key!=k) {
7      n = n->next;
8    }
9
10   return n;
11 }
```

(b) hash_find

Figure 2: **Complex data structure method examples** – (a) `heap_up` is used to restore the heap condition in a heap-based priority queue after a `push`. (b) `hash_find` looks up the hash bucket for a key and searches through the collision chain to find the associated value.

rithm. This not only enables traditional HLS optimizations to be applied to each individual part, but also allows us to exploit parallelism between method calls. In this paper, we propose a novel HLS methodology and an associated architectural template to create *specialized container units* (SCUs), which implement such decoupled data structures efficiently in HLS by enforcing a clean separation between computation and data access. Our proposed approach is partly inspired by the C++ standard template library (STL) [10], which utilizes the concept of containers to represent common data structures.

There are several previous efforts at designing specialized hardware for a limited subset of complex data structures (e.g., priority queue [7], random decision tree [11]) using both HLS and RTL flows. However, the focus of this paper is not to create the most optimized implementation of any data structure; our technique is orthogonal to such efforts. Instead, we propose a technique which can potentially improve the performance of any program using a complex software data structure. In essence we are trading off optimality for generality and ease of use. In this sense, our work represents a first step towards enabling efficient complex data structure synthesis for HLS, and to expand the scope of HLS beyond highly regular, algorithm-dominated applications. Our primary technical contributions are as follows:

- We investigate the synthesis problem for a set of STL-inspired data structures containing complex methods that are poorly optimized by conventional HLS techniques.
- We propose a novel HLS architectural template, in which complex data structure methods are decoupled using a latency-insensitive interface to facilitate pipelining and parallelization.
- We experimentally demonstrate how our techniques lead to significant performance improvements on a variety of classic data structures and algorithms.

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.

2.1 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. Figure 2 contains examples of complex methods belonging to two well-known data structures: `heap` and `hash_table`.

Figure 2(a) shows `heap_up`, which plays an important role in the `push` method of a heap-based priority queue by iteratively swapping the newly pushed value up towards the top of the heap if necessary. The variable-bound main loop as well as the early exit condition (which complicates unrolling and pipelining) marks this as a complex method. Figure 2(b) shows `hash_find`, which searches for a key in a hash table utilizing separate chaining. This method is complex because it hashes the key to an array index (causing unpredictable memory access), then traverses a variable-length collision chain (requiring the use of a variable-bound loop).

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.

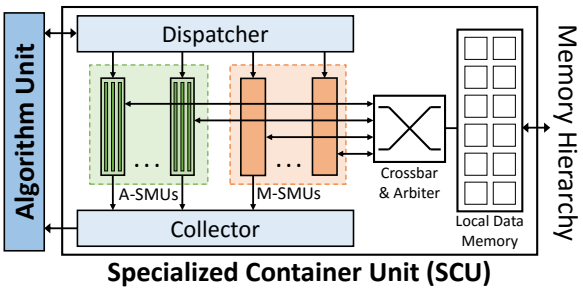

Figure 3: **SCU architectural template** – Complex methods are synthesized into specialized method units (SMUs) managed by a *Dispatcher* and *Collector*. SMUs interface with memory through a crossbar and arbiter. Mutator SMUs (M-SMUs) connect to the collector if they return a value. Accessor SMUs (A-SMUs) can be further optimized to support multi-lane out-of-order execution.

```
1  s = u.begin_neighbors();
2  e = u.end_neighbors();
3
4  // algorithm loop
5  for (v = s; v < e; ++v) {
6  #pragma pipeline
7    alt = dist[u] + edge[u][v];
8
9    if (dist[v] > alt) {
10     dist[v] = alt;
11
12     // Priority Queue
13     // push method
14     Q.push(v, dist[v]);
15   }
16 }
```

(a) Dijkstra's algorithm inner loop

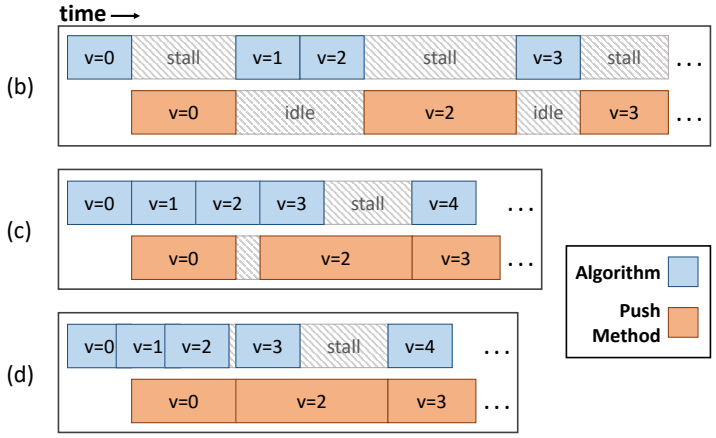

Figure 4: **Execution of a program containing a complex mutator method –** (a) Code for the inner loop of Dijkstra's algorithm, which uses `push` into a priority queue. Not every loop iteration calls `push`; (b) Baseline execution with no optimization, both algorithm and method modules are poorly utilized; (c) Decoupled execution with overlapping eliminates unnecessary stalling in the algorithm; (d) Pipelining the algorithm (indicated via overlapped blocks) further improves utilization, throughput is now limited by the method.

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 be portable 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 `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 `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 `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 `push` is a method which does not return any values. After receiving a `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 `push` call from the previous iteration as seen in see Figure 4(c). This greatly reduces stalling on both hardware modules. Figure 4(c) also illustrates how our approach exploits

```
1  void KeySearch (keys, vals) {
2    // algorithm loop
3    for (k : keys) {
4      // Hash Table find method
5      // returns matching bucket node
6      HashTableNode n;
7      n = H.find(k);
8
9      // check if key was found
10     if (n != NULL) {
11       vals[i] = n.value;
12     }
13   }
14 }
```

(a) KeySearch kernel

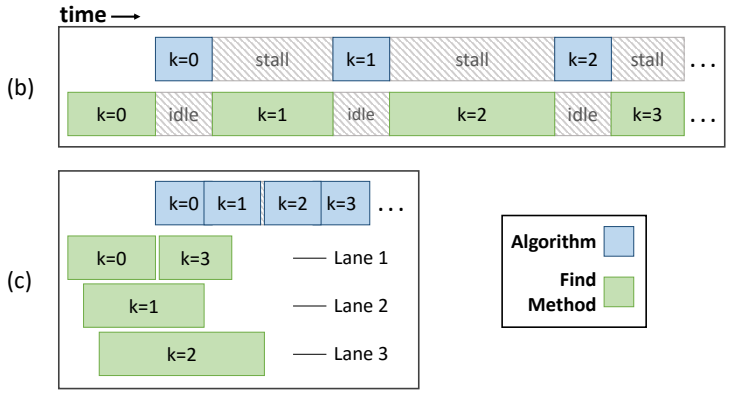

Figure 5: **Execution of a kernel containing a complex accessor method –** (a) Code for the KeySearch kernel, which searches a hash table for a list of keys using the find method and contains a variable-bound while loop; (b) Baseline execution with no optimization, the algorithm cannot be pipelined;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

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 nested loop in the push method. After decoupling, 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.

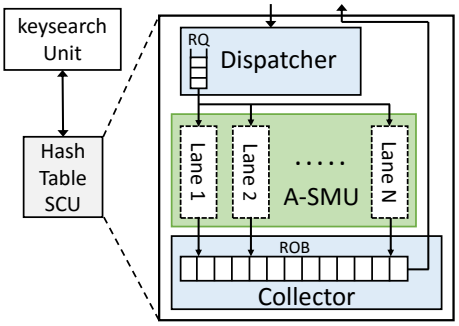

Figure 6: **Multi-lane SMU example** – The find method in a hash table can be parallelized using a multi-lane A-SMU. The host SCU must contain a Request Queue and Reorder Buffer to ensure correct operation. The SCU's local memory is not shown here.

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 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 transformations combined with parameterized hardware generators. The first step in SCU synthesis is to identify which methods are to be decoupled 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 message 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 instructions 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 programmer effort and no change to the algorithm code.

Table 1: **Latency and resource usage comparison for each benchmark —** Target clock period is `5ns`. `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.

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

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. For the decoupled designs we synthesized the algorithm and decoupled data structure methods separately, composed them in a simple RTL wrapper, then pushed the combined module through the implementation flow. All clock period and area numbers were obtained post place and route, and all latency numbers were obtained from cycle-accurate RTL simulation. The simulation testbenches were written leveraging a Python-based hardware modeling framework called PyMTL [8], which simplified the creation of our testing and evaluation environment.

Table 1 shows the benchmark algorithms chosen for evaluation as well as the data structures they use. `Dijkstra` solves the single-source shortest path problem with Dijkstra's algorithm using a priority queue implemented as an array-based binary heap. `DigitrecKnn` uses the K-nearest neighbors algorithm to classify digits, leveraging the same priority queue design to keep track of the best matching candidates. `DFS` performs a depth-first search of a graph using a dynamically-sized stack (invoking a dynamic memory allocator using implicit free lists). These three algorithms make heavy use of `push` and `pop`, both of which are synthesized as decoupled mutator SMUs to take advantage of overlapped execution. The `empty` method is also invoked, but it is not complex and is inlined into the algorithm. The next two benchmarks are read-heavy kernels which make many calls to a complex accessor method/ Both use a two-lane A-SMU architecture with a 32-entry ROB. Dual-port block RAMs (BRAMs) were used to implement the data memory. `ImgSeg` segments an image by classifying each pixel into one of multiple predefined classes. The data structure used is a random decision tree with 11 levels stored in an array. `KeySearch` was shown earlier, and finds a list of input keys inside a 2000 entry hash table stored in an array.

In `Dijkstra` and `DFS`, the algorithm is mostly responsible for loading and storing graph data, which is small compared to the latency of the data structure method. However, both benchmarks exhibit conditional method calls: in `Dijkstra`, a node is only pushed to the priority queue if its distance was updated; in `DFS`, adding a node to the stack does not always require the memory allocator. This property, along with pipelining of the algorithm, allows for significant execution overlap achieving good speedup. 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 hash table). 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 the SCU in the 2-lane hash table design. The dispatcher, collector, and memory interface incurs significant overhead.

However, we also have results showing that this area overhead stays relatively constant even when the number of lanes is increased. Table 7 shows the scalability of our approach up to 4 lanes. FF and LUT usage grow 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 2: **Area breakdown of the KeySearch SCU (2 lanes)**.

KeySearch-SCU	FF	LUT	BRAM
Dispatcher	19%	26%	0
Collector	25%	25%	2
SMU #1	14%	7%	0
SMU #2	14%	10%	0
Mem Interface	28%	32%	12

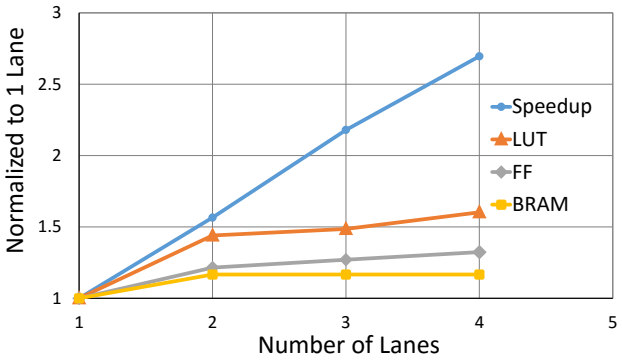

Figure 7: **Latency and resource usage of KeySearch-SCU with increased number of lanes.**

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 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

[1] G. Bloom, G. Parmer, B. Narahari, and R. Simha. Shared Hardware Data Structures for Hard Real-Time Systems. *Int'l Conf. on Embedded Software*, Oct 2012.

[2] A. Carbon, Y. Lhuillier, and H.-P. Charles. Hardware Acceleration of Red-Black Tree Management and Application to Just-In-Time Compilation. *Journal of Signal Processing Systems*, Oct 2014.

[3] R. Cattaneo, G. Pallotta, D. Sciuto, and M. D. Santambrogio. Explicitly Isolating Data and Computation in High Level Synthesis: the Role of Polyhedral Framework. *Intl. Conf. on ReConFigurable Computing and FPGAs (ReConFig)*, Dec 2015.

[4] S. Cheng and J. Wawrzynek. Architectural Synthesis of Computational Pipelines with Decoupled Memory Access. *Int'l Conf. on Field Programmable Technology (FPT)*, Dec 2014.

[5] J. Cong, B. Liu, S. Neuendorffer, J. Noguera, K. Vissers, and Z. Zhang. High-Level Synthesis for FPGAs: From Prototyping to Deployment. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, Apr 2011.

[6] D. Diamantopoulos, S. Xydis, K. Siozios, and D. Soudris. Mitigating Memory-induced Dark Silicon in Many-Accelerator Architectures. *Computer Architecture Letters (CAL)*, Mar 2015.

[7] M. Huang, K. Lim, and J. Cong. A Scalable, High-Performance Customized Priority Queue. *Int'l Conf. on Field Programmable Logic and Applications (FPL)*, Sep 2014.

[8] D. Lockhart, G. Zibrat, and C. Batten. PyMTL: A Unified Framework for Vertically Integrated Computer Architecture Research. *Int'l Symp. on Microarchitecture (MICRO)*, Dec 2014.

[9] J. Loew, J. Elwell, D. Ponomarev, and P. H. Madden. A Co-Processor Approach for Accelerating Data-Structure Intensive Algorithms. *Int'l Conf. on Computer Design (ICCD)*, Oct 2010.

[10] D. R. Musser, G. J. Derge, and A. Saini. *STL Tutorial and Reference Guide: C++ Programming with the Standard Template Library*. Addison-Wesley Professional, 2009.

[11] J. Oberg, K. Eguro, R. Bittner, and A. Forin. Random Decision Tree Body Part Recognition Using FPGAS. *Int'l Conf. on Field Programmable Logic and Applications (FPL)*, Aug 2012.

[12] J. Park and P. C. Diniz. Data Reorganization and Prefetching of Pointer-Based Data Structures. *IEEE Design and Test of Computers*, Aug 2011.

[13] M. Tan, G. Liu, R. Zhao, S. Dai, and Z. Zhang. ElasticFlow: A Complexity-Effective Approach for Pipelining Irregular Loop Nests. *Int'l Conf. on Computer-Aided Design (ICCAD)*, Nov 2015.

[14] G. Weisz and J. C. Hoe. CoRAM++: Supporting Data-Structure-Specific Memory Interfaces for FPGA Computing. *Int'l Conf. on Field Programmable Logic and Applications (FPL)*, Sep 2015.

[15] F. Winterstein, S. Bayliss, and G. A. Constantinides. High-Level Synthesis of Dynamic Data Structures: A Case Study Using Vivado HLS. *Int'l Conf. on Field Programmable Technology (FPT)*, Dec 2013.

[16] L. Wu, M. Kim, and S. A. Edwards. Cache Impacts of Datatype Acceleration. *Computer Architecture Letters*, Jun 2012.

[17] Xilinx Inc. *Vivado Design Suite User Guide: High-Level Synthesis*, Jul 2012.

[18] J. Xu, Y. Dou, J. Song, Y. Zhang, and F. Xia. Design and Synthesis of a High-Speed Hardware Linked-List for Digital Image Processing. *Congress on Image and Signal Processing (CISP)*, May 2008.