2016 IEEE 24th Annual International Symposium on Field-Programmable Custom Computing Machines

SynADT: Dynamic Data Structures in High Level Synthesis

Zeping Xue
Department of Electrical and Electronic Engineering
Imperial College London
London, UK
zeping.xue10@imperial.ac.uk

David B. Thomas
Department of Electrical and Electronic Engineering
Imperial College London
London, UK
d.thomas1@imperial.ac.uk

***Abstract*—Abstract Data Types (ADTs) such as dictionaries and lists are essential for many embedded computing applications such as network stacks. However, in heterogeneous systems, code using ADTs can usually only run in CPUs, because components written in HLS do not support dynamic data structures. HLS tools cannot be used to synthesise dynamic data structures directly because the use of pointers is very restricted, such as not supporting pointers to pointers or pointer casting. Consequently, it is unclear what the API should look like and how to express dynamic data structures in HLS so that the tools can compile them. We propose SynADT, which consists of a methodology and a benchmark. The methodology provides classic data structures (linked lists, binary trees, hash tables and vectors) using relative-addresses instead of pointers in Vivado HLS. The benchmark can be used to evaluate the performance of data structures in HLS, ARM processors and soft processors such as MicroBlaze; CPUs can utilise either the default C memory allocator or a hardware memory allocator. We evaluate the data structures in a Zynq FPGA demonstrating scaling to approximately 10MB memory usage and 1M data items. With a workload that utilises 10MB memory, the HLS data structures operating at 150MHz are on average $1.35\times$ faster than MicroBlaze data structures operating at 150MHz with the default C allocator and $7.97\times$ slower than ARM processor data structures operating at 667MHz with the default C allocator.**

I. Introduction

Abstract Data Types (ADTs), such as dictionaries and stacks, are basic elements in many embedded computing applications. For example, ADTs are needed in network protocol stacks, such as dictionaries to map IP addresses to network flows, and lists to manage data buffers. Currently CPUs are often used for such applications, as the linked lists, binary trees and hash tables underlying the ADTs require runtime memory allocation using `malloc` and `free`, as well as complicated pointer arithmetic and "pointers to pointers".

It is desirable to migrate applications using ADTs from software to hardware, but while it is possible to implement complex data structures in RTL, the process is complex and requires the removal of data structure abstraction layers[1]. Recent developments in High Level Synthesis (HLS) tools, such as Xilinx Vivado HLS, allow developers to create hardware by porting C and C++ to hardware. However, current HLS tools do not support dynamic memory allocation, and also have severe restrictions on the types of pointer operations allowed, meaning software APIs and implementations of ADTs cannot be used. Recent work has looked at manually implementing restricted classes of data structures using simple allocators [2], and providing memory allocation as an IP core on a shared bus [3], but these do not offer the combination of simplicity, generality, and portability found in code using software ADTs.

In this paper, we first investigate the limitations imposed by HLS tools on the implementation and use of complex data structures. We then use this knowledge to design a simple and HLS-compliant API for accessing ADTs, as well as the concrete data structures used to implement them in HLS. A key goal is portability, both in terms of the same code running in software or hardware, but also in allowing data structures to be easily passed between software and hardware at runtime.

The contributions of this paper are:

- An analyis of the problems encountered when implementing complex data structures in HLS, as well as a set of techniques for resolving those problems.
- SynADT[1]: a open-source synthesisable library for linked lists, binary trees, hash tables and vectors which applies the techniques, enabling the creation and use of ADTs in HLS, as well as allowing data structures to be accessed from both software and hardware.
- BenchADT: a benchmark for evaluating both hardware and software data structures, including workload and test-harness, providing a portable and fair means of comparison between applications running on different platforms.
- An evaluation of SynADT in HLS, MicroBlaze, and ARM on a Zynq device, showing HLS dynamic data structures are on average $1.35\times$ faster than MicroBlaze for dynamic data structures containing 10M nodes.

The paper is organised as follows: Section II gives the background. Section III discusses the related work. Sections IV and V discuss the challenges to HLS data structures and our proposed solutions. Sections VI and VII show the benchmark design and evaluations of HLS data structures. Section VIII concludes the paper.

II. Background

ADTs such as stacks, queues and dictionaries are intended to expose functionality without requiring a specific implementation [4]. For example, C++ provides *std::map* as an

[1]The source code can be found at https://github.com/Hilx/SynADT

978-1-5090-2356-1/16 $31.00 © 2016 IEEE
DOI 10.1109/FCCM.2016.26

ordered dictionary ADT, but the underlying implementation could be an AVL tree, a Red-Black tree, or another kind of concrete data type. The user only needs to know the operations and asymptotic performance guarantees defined by the interface, and can compile their code on any compliant C++ implementation.

While there are many types of ADTs, we focus on the following core ADTs:

Vector: A dynamically sized array, which allows random access in $O(1)$ time, and appending in amortised O(1) time.

Stack: Provides a Last-In First-Out policy, requiring $O(1)$ time to push a new item into the sequence, or to pop the most recent item from the sequence.

Queue: Provides an ordered sequence of items, requiring $O(1)$ time to push a new item into the sequence, or to pop the oldest item from the sequence.

Linear Dictionary: Maintains a mapping between keys and values, with insertion, deletion, and lookup taking $O(1)$ time.

Ordered Dictionary: A dictionary allowing items in the container to be enumerated in order of keys, with insertion, deletion, and lookup taking $O(\log n)$ time.

These ADTs need to be defined in terms of concrete underlying data types. While there are some exotic data structures, such as Skip-Lists or Splay-Trees, the needs of most embedded applications can be covered by a small number of concrete implementations:

Linked List (*Stack*,*Queue*): A linked list is formed of a sequence of nodes, where each node contains a value and a pointer to the next node. It can be used to implement stacks and queues, as nodes can easily be inserted or removed at the start and end of the list.

Vectors (*Vector*,*Stack*): Concrete vectors are simply dynamically allocated arrays, which grow in size when new data is added. In order to maintain amortised constant-time appends, the underlying array must be grown by a multiplicative factor when space runs out.

Unbalanced Binary Tree (*Ordered Dictionary*): A binary tree is constructed of nodes that can have at most two child nodes, where each node contains a key and a left and a right pointer. Binary trees have the invariant that everything under the left node is less than the key, and everything on the right is greater, allowing for efficient search. If the data is randomly distributed the height will be $O(\log n)$, but the binary tree will degenerate into a linked list for sequential data.

Balanced Binary Tree (*Ordered Dictionary*): Self-balancing trees ensure invariants on the heights of sub-trees in a binary tree, ensuring $O(\log n)$ operations even for unbalanced data. This provides guaranteed worst-case performance, but increases the cost per operation in the average case as the tree must be re-balanced during inserts and deletes. Two popular choices are the AVL tree and Red-Black tree, with the AVL tree requiring slightly more complicated operations in exchange for a better balanced tree.

Hash Table (*Dictionary*): Hash tables allow $O(1)$ lookup using a key, but do not allow the contained items to be enumerated in sorted order. This is achieved by forming the hash of each key, and using the hash to identify one potential bucket in an array of buckets. The ratio of nodes to buckets determines the load factor, which controls how much sequential searching is needed within each bucket. A low load factor ensures a large hash table with small buckets, but requires the size of buckets to be dynamically extended on overflow.

This work focuses on these concrete data structures, as they can be used to implement the basic ADTs such as dictionaries, queues and sets, while more advanced ADTs such as priority queues can be implemented using ordered sets.

III. Related Work

While ADTs are more commonly used in CPU-based embedded system applications, there is still some research regarding dynamic-data structures on FPGAs. Xu et al.[5] use a linked list in an FPGA as part of an image processing application to manage variable size data sets. Memory allocation was managed using an application-specific scheme, making it simpler and faster compared to a general memory allocator. As a result, their linked list has limited flexibility and scalability – the maximum list length is determined at design time, with the resource utilisation and critical path growing with list size. For example, when the number of nodes in the list grows from 512 to 1024, the amount of FPGA logic required increases from 1341 to 2486 and the maximum operating frequency drops from 145MHz to 129MHz.

Weisz et al.[6] proposed CoRAM++, a programming environment for FPGAs which provides optimised DRAM access over irregular data structures of limited size. Their work showed the potential for ADTs on FPGAs, demonstrating efficient list traversal and merging on FPGA, but the creation and manipulation of data structures was not supported. However, CoRAM++ has the potential to work *together* with the system proposed here, improving access time once the data structure has been created.

Winterstein et al.[7] designed a tree-based data structure for a K-Means application, using an application-specific memory allocation scheme. Their work demonstrated the improvement in memory efficiency of using dynamic memory management over static allocations. Their later work [2] attempted to reduce the overhead by analysing the application and converting dynamic memory usage into static memory allocations, so that they can be compiled as HLS code. However, it does not support the general case of an arbitrary data structure that cannot be statically analysed.

Diamantopoulos et al.[8] proposed a HLS dynamic memory management framework for multi-accelerator FPGA systems. By increasing the memory efficiency using the proposed HLS memory allocator, more parallel accelerators with private heaps can be added into the system with limited memory

resource and hence a higher speed-up can be achieved, but this did not specifically address the data structures and programming model needed to program the system.

All existing approaches for synthesising dynamic data structures require either dedicated implementation of a simple application-specific memory management scheme, or resolving the dynamic memory usage using static allocations. They also assume that data structures will be built and used in hardware only, and cannot be shared between software and hardware, nor can one data structure be placed inside another. In contrast, SynADT aims to provide dynamic data structures implemented in HLS that can also be shared between hardware and software. SynADT also tries to allow flexible composition of data structures so different data structures can be nested - for example, a linked list entry can store a vector.

IV. Challenges in mapping dynamic data structures to HLS

The lack of existing approaches and libraries for creating and accessing dynamic data structures in HLS is due to the restrictions that HLS tools place on source code. These restrictions vary between tools, but in order to provide a portable library we must first identify the intersection of these restrictions across common tools, then devise portable methods that can bypass the restrictions. The main tool considered here is Vivado HLS, as this has the tightest requirements when compared to languages such as Legup and Handel-C.

A. Challenges due to HLS restrictions

An initial challenge for dynamically sized data structures is the lack of dynamic memory allocation in HLS tools. Currently no HLS tools support `malloc` or `free`, so it is up to the user to implement simple versions or fake the calls. One issue is the need for a dynamic memory allocation algorithm, along with the data structures used to track the allocated and free blocks. These algorithms could be implemented as functions, and the data structures as global variables:

```
// Challenge: who initialises this?
struct malloc_state_t _malloc_state;

// Challenge: Points to which memory space?
void *malloc()
{
    // Challenge: how to get a pointer to DDR?
    void *managedMem= ...;
    // ...
    return newPtr;
}
```

However, this presents a number of challenges:

- Initialisation: how is the management data initialised? If software and hardware share an allocator, who sets it up? What about when there is no software?
- Location: how does a piece of hardware connect with the memory being managed? There is no equivalent to `sbrk` to hand out segments of heap memory, and different HLS tools have different methods for creating pointers to external RAMS.
- Memory Space: HLS tools may support a single address space (Legup), require each pointer to be bound to a distinct address space (VHLS), or perform points-to analysis to allow a pointer to identify potential targets at compile-time (Handel-C). What kind of pointer should malloc return?

System-wide allocators such as SysAlloc[3] and HLS malloc[8] allow some of these problems to be solved, as they place the algorithm and state outside the HLS and CPU domains. However, there is still the potential to have multiple address spaces, and multiple instances of distinct allocators.

Other challenges arise when trying to describe general-purpose data structures. In software this is simply a case of declaring structures with embedded pointers:

```
struct binary_node
{
  int key;      // Challenges:
  void *value; // what is a generic payload?
  binary_node *left; // pointer to pointer?
  binary_node *right;
};
```

However, we again encounter fundamental restrictions:

- Type Erasure: generic payloads cannot be supported using `void` pointers, as such pointers are not supported in all tools – for example, it is not possible to cast away (erase) types in Vivado HLS. This makes it difficult to nest or embed arbitrary data structures within another type of data structure.
- Pointer-to-Pointer: Some HLS tools do not allow structures to contain embedded pointers.

A further challenge is the restriction on recursion in HLS. For example, when rebalancing a tree or deleting a tree, the function needs to be called recursively:

```
void DeleteTree(pointer_type *root){
  if(root != NULL){        // Challenge:
    DeleteTree(root->left); // recursion
    DeleteTree(root->right);
    free(root);
  }
}
```

Recursion can be rewritten to use stacks, but the problem arises of where to put the stack - if it cannot be given a guaranteed bounded size at compile-time, it must be placed in global memory.

A final challenge when interoperating between software and hardware is that embedded pointers may not mean the same thing. A pointer written with address 0xC000 in hardware may resolve to a different memory location when read from software, as the same memory may be mapped to two different locations.

B. Objectives

Set against these restrictions, we can identify the properties that a good library for ADT in HLS should have:

- Pluggable memory management: allocation can be handled locally, or using a shared system-wide allocator.
- Shared data structures: data structures can be allocated in one domain, such as a CPU, then passed to another domain such as HLS or a soft CPU for processing and deallocation.
- Structure composability: data structures should be able to be composed flexibly so that one data structure can be placed inside another.
- Portable API: the same code can be compiled and synthesised in multiple CPU architectures and using different HLS tools.
- Implementation Abstraction: while the data structures should be shared between software and hardware, the API should not be bound to the particular code which manipulates and accesses a data structure.
- SynADT is not aiming for high performance so it needs to be optimised after the initial version.

C. Principles of SynADT

Given the challenges and objectives, we now define a number of general principles for creating HLS-compatible ADTs and dynamic data structures.

In order to deal with the need for multiple possible memory allocators for multiple address spaces, as well as solving the initialisation problem of getting an initial pointer, a general principle is *explicit allocators*. This means that memory allocation functions, and by extension any ADTs which use memory allocation, must receive the allocator as a parameter:

```
sys_ptr_t HW_Malloc( //pAlloc identifies the
  sys_alloc_t pAlloc, // allocator , as well as
  int nBytes          // the memory space type
);
```

Making the allocator explicit means that HLS tools which require a root pointer from which to form other pointers will always have a root in the form of the allocator. The `sys_alloc_t` type can be as simple as a pointer (e.g. for raw SysAlloc), or could contain a pair of the pointer to an allocator and a pointer to the allocated state.

ADT API functions must also take an allocator so that they know how to allocate or de-allocate memory associated with a given data structure:

```
tree_ptr_t Tree_Insert ( // pAlloc identifies
  sys_alloc_t pAlloc,    // both RAM and allocator
  tree_ptr_t pTree,      // Abstract data type for
  int key, int value     //                  list
);
```

The type *tree_ptr_t* is a classic opaque data type, hiding the details of implementation. However, instead of hiding an opaque pointer type, for HLS it is used to hide the fact that there is no pointer. Instead the data types are expressed in terms of relative offsets:

```
typedef int tree_ptr_t ;
```

```
tree_ptr_t Tree_GetLeft(
  sys_alloc_t pAlloc,
  tree_ptr_t pTree
){
  return Sys_GetPointer(pAlloc)+pTree+1;
}
```

This solves three problems in one go. First, we no longer have pointers to pointers, as there are only relative offsets. These relative offsets can always be locally turned into pointers by making use of the supplied allocator. Second, the data structures are now location independent, so the same data structure can be accessed via two pointers with different addresses, but accesses to data structures will still resolve to the correct location. Third, if a data structure is represented by an integer, and data structures can contain integers, then it becomes possible to embed a "pointer" to any data structure within another data structure (as long as they share an allocator):

```
// Add something to a list
myStack=Stack_Push(pAlloc, myStack, v);
// Store the stack in a map
myMap=Dictionary_Insert(pAlloc,myMap, key,myStack);
// ...
// Retrieve stack from the map
myStack=Dictionary_Lookup(pAlloc,myMap,key);
```

So now the problem of type erasure is removed - there are no pointers needed, as only the API provides access to the data in data structures.

The final problem is recursion requiring an unbounded stack, and here we can exploit the approach already described. Any ADT manipulating function receives an allocator argument, which allows it to allocate nodes of a different type. So a user-facing function for manipulating unbalanced trees can internally allocate and destroy lists as necessary. This will typically only be needed in the worst case, but allows us to escape the limits of fixed-size local stacks.

V. SynADT

The previous section outlined some challenges and objectives, as well as portable methods for solving and meeting the objectives. SynADT is a concrete implementation of these ideas, designed to be synthesisable in the majority of HLS tools, as well as being useable in software.

A. Memory Allocation

As outlined in the objectives, the memory allocation method is pluggable: in HLS, the default allocator supported is the SysAlloc framework[3]. But this is an open-source IP core which is allocated a global address. Knowing the allocator address allows clients to request access via a bus-based mutex, then to perform allocation and free requests. Both software and hardware clients can share the same allocator, allowing the construction of data structures which can be shared. SysAlloc returns addresses relative to its own address. This fits with the model proposed earlier: given a pointer to SysAlloc, it is possible to both allocate memory, and to turn an offset back into a pointer.

Hardware allocators have some overhead for locking and client arbitration, so when working exclusively within software it is likely to be more efficient to use the native C-library allocator. SynADT allows the memory allocator to be swapped, as shown for the ARM and Microblaze implementations in the results. Performance will be higher, but this loses the ability to create data structures in software and modify or destroy them from hardware. However, hardware clients can still access software allocated data structures, as long as they do not try to perform operations which allocate or release nodes.

B. Binary Trees

SynADT supports all five basic data structures identified in Section II, including both unbalanced and balanced (AVL) trees. The reason for supporting both is that unbalanced trees need less logic and fewer memory operations per update, as they do not need to rotate nodes for rebalancing. However, during deletion it is necessary to recurse, and because the tree could be severely unbalanced this might require a large stack. In exchange for low overhead updates, the balanced tree must use a list based dynamic stack, but this is hidden behind the user visible API.

In comparison, the balanced tree requires more memory accesses per insert, but can guarantee a balanced tree. Because the tree is balanced, and the total number of nodes cannot exceed 2^{32} (due to memory constraints), the balanced tree can use a fixed size stack. The maximum depth is 32, allowing an efficient BRAM based stack.

C. Hash Table

There are many ways of implementing hash tables, including open addressing and seperate chaining. In this work, we realise a chained hash table, where each bucket is implemented with a contained linked list. The hash function used is as follows: $h = k \mod (n/bucketSize)$ where the bucket size used in this work is 16. The locations of the heads of lists are stored in an array and the key will then be looked up in the hth list.

D. Vectors

Vectors are implemented as arrays, which contain information about both their current size and their allocated size. When new data is appended to a vector, if the new size exceeds the allocated size then the vector must be grown to double the size. In C this would be performed using `realloc`, but SysAlloc does not support such a function. Instead a new block of memory will be allocated, the data will be copied over, and the old block discarded.

E. Discussion

SynADT FPGA implemented data structures are composable, as different operations can be realised on different platforms such as FPGA and software processors. For data structures, although the contained values are integers, the data itself can have variable sizes as this can be decided when the node is created.

The memory management functions can allow developers to choose different allocators to use for their project. Furthermore, multiple memory allocators can be used in parallel for special data structure manipulations to increase the efficiency since the modifications can be done easily within two functions (*Sys_Malloc()*, *Sys_Free()*). Allocators can even be private or global to further enhance the performance which is desirable in embedded systems.

VI. BenchADT

To explore what the performance penalties and advantages of different platforms are, a portable benchmark, BenchADT, can be applied to provide comparisons of hardware and software data structures. For now we only apply this benchmark to SynADT within the Zynq environment, but it is designed to be applied to other ADT implementations in other platforms, allowing meaningful comparisons against other approaches.

Latency Measurements: The recommended approach to measure latency is to use a set of timers implemented in FPGA logic and packaged as an IP core. Both software and hardware data structures can start, stop and reset timers at any time by performing memory-mapped writes.

Data Structure Scalability: By scaling the size of data structures, we can explore how the performance can be influenced by unpredictable factors such as the use of caches in different platforms and data structures. This work is designed to support data structure sizes up to DDR scale, so the target size is 10MB memory usage (2^{20} entries).

Performance Breakdown: Benchmarking the performance per-operation breakdown allows pre-implementation analysis. For example, an application might need to to create a tree of size a in one phase of a program, then perform b look-ups in another phase. Using the performance breakdown benchmark, the latency of such applications can be estimated with the two phases located in software, hardware or a combination of the two. In order to obtain the performance breakdown information, latencies of each phases of the experiment run are measured. Table I shows the required operations chosen in each phase for each data structures. The typical phases are creation, manipulation and deletion.

Benchmark Stimulus: To study how input values can impact the tree and hash table data structures performances, we define two workloads: uniform random values and sequential values. The xorshift[9] Random Number Generator (RNG) is used in both software and hardware experiments to generate random values. As the generator generates 32-bit outputs, a bit-mask is used to reduce the output range of the random number generator according to the current data structure scale.

The range of random numbers in each phase is given in Table I. In certain phases, half the nodes in the structure are processed. The range of random numbers varies according to the phase to provide inputs that are already (or not) in the structure depending on the phase definition.

VII. Evaluation

In this section, we will evaluate SynADT data structures implemented in HLS and compare them against software in Zynq SoC using BenchADT.

Data structures are implemented using Vivado HLS (v14.4) and packaged using Xilinx Vivado Design Suite (v14.4) into

TABLE I: Benchmark design: performance breakdown (2^n nodes or entries are processed in each experiment run; i is the iteration index; *RNG r* is the range of the random numbers

		Phase 1			Phase 2			Phase 3				Phase 4
Data Structures	**Input Case**	**Operation**	**Total Iterations**	**Input per Iteration**	**Operation**	**Total Iterations**	**Input per Iteration**	**Operation**	**Total Iterations**	**Input per Iteration**		**Operation**
Linked List	N/A	Insert	2^{n-1}	N/A	Reverse	N/A	N/A	Delete	N/A	N/A		N/A
Vectors	N/A	Insert	2^{n-1}	N/A	Sequential Access	2^{n-1}	access ith item	Random Access	2^{n-1}	access kth item, k = RNG 2^n		N/A
Unbalanced Tree, AVL Tree, Hash Table	Random	Insert	2^{n-1}	RNG r = 2^{n-1}	Check, Insert	2^{n-1}	RNG r = 2^n	Update old value with new value	2^{n-1}	old: RNG r = 2^{n-1}	new: RNG r = 2^n	Delete
	Sequential			$(i+1) \times 2^4$			RNG r = 2^{n+2}			old: RNG r = 2^{n+2}	new: RNG r = 2^{n+3}	

TABLE II: For data structures with different memory usages: Latency of the data structure relative to the ARM core with C malloc (Absolute latency in us)

(a) 128KB memory usage (2^{13} entries)

Platform	Linked List	Unbalanced Tree	AVL Tree	Hash Table	Vector
HLS	9.65 (5.22)	29.25 (237.29)	2.92 (62.19)	3.80 (11.35)	4.44 (2.22)
ARM C malloc	1.00 (0.54)	1.00 (8.11)	1.00 (21.30)	1.00 (2.99)	1.00(0.50)
ARM SysAlloc	9.49 (5.13)	1.46 (11.89)	1.13 (23.96)	2.30 (6.85)	1.00 (0.50)
uBlaze C malloc	17.88 (9.66)	9.28 (75.24)	8.16 (173.91)	11.74 (35.04)	11.28 (5.64)
uBlaze SysAlloc	15.99 (8.64)	8.19 (66.41)	7.76 (165.30)	11.75 (35.08)	11.27 (5.64)

(b) 10MB memory usage (2^{20} entries)

Platform	Linked List	Unbalanced Tree	AVL Tree	Hash Table	Vector
HLS	8.88 (5.22)	155.58 (2461.17)	2.57 (97.21)	2.52 (11.80)	3.58 (2.28)
ARM C malloc	1.00 (0.59)	1.00 (15.82)	1.00 (37.75)	1.00 (4.68)	1.00 (0.64)
ARM SysAlloc	18.70 (10.98)	45.77 (724.07)	48.54 (1832.60)	4.13 (19.34)	1.00 (0.64)
uBlaze C malloc	17.59 (10.33)	8.00 (126.61)	7.67 (289.73)	10.68 (50.01)	12.60 (8.01)
uBlaze SysAlloc	14.99 (8.80)	7.19 (113.73)	7.46 (281.54)	10.20 (47.75)	12.60 (8.01)

IP cores. SysAlloc is managing 64MB DDR with base block size of 4 bytes. The target platform is Zynq-7000 SoC and the ARM processor is operating at 667MHz with 32KB L1 instruction and data caches and 512KB L2 cache. The soft processor MicroBlaze (FPGA) with highest performance configuration is operating at 150MHz with 64KB data and instruction caches. HLS data structures, SysAlloc and timer IPs are in FPGA operating at 150MHz. XC7Z045 FPGA is used for HLS AVL tree IP because AVL tree implementation requires much more FPGA logic; an XC7Z020 FPGA is used for all other data structures. DDR memory in both platforms operate at same speed.

A. Comparisons of Platforms

The platform comparisons for data structures with 128KB and 10MB memory usages are shown in Tables II; 10MB is a DDR-scale memory usage and 128KB is a BRAM-scale memory usage that can fit into a small Cyclone III 3C16 FPGA chip. HLS data structures are slower than those in ARM core with default C malloc, and are faster than those in MicroBlazes. The exception is in the case of very large unbalanced tree, as the search path is unbounded and there is no cache to improve the memory access efficiency in HLS.

TABLE III: Alternative platforms' speed-ups over reference platforms calculated using geometric means of latencies

(a) 126KB memory usage (2^{13} entries)

(each entry = Ref. Lat./Alter. Lat.)		Reference				
		HLS	ARM C malloc	ARM SysAlloc	uBlaze C malloc	uBlaze SysAlloc
Alternative	HLS	1.00	0.15	0.30	1.67	1.57
	ARM C malloc	6.74	1.00	2.05	11.24	10.61
	ARM SysAlloc	3.29	0.49	1.00	5.49	5.18
	uBlaze C malloc	0.60	0.09	0.18	1.00	0.94
	uBlaze SysAlloc	0.64	0.09	0.19	1.06	1.00

(b) 10MB memory usage (2^{20} entries)

(each entry = Ref. Lat./Alter. Lat.)		Reference				
		HLS	ARM C malloc	ARM SysAlloc	uBlaze C malloc	uBlaze SysAlloc
Alternative	HLS	1.00	0.13	1.40	1.35	1.26
	ARM C malloc	7.97	1.00	11.14	10.78	10.06
	ARM SysAlloc	0.72	0.09	1.00	0.97	0.90
	uBlaze C malloc	0.74	0.09	1.03	1.00	0.93
	uBlaze SysAlloc	0.79	0.10	1.11	1.07	1.00

Table III shows the platform-to-platform comparisons. In both size cases, HLS is faster than MicroBlaze. In 10MB memory usage case, MicroBlaze outperforms ARM core with SysAlloc because MicroBlaze cache flush time is shorter as its cache is much smaller. Fig.1 compares the performance of different platforms. Microblaze data structures with C malloc and SysAlloc have very close performance and HLS is faster than MicroBlaze; however, all HLS data structures utilise more FPGA logic than MicroBlaze, as shown in Table IV. On average, HLS is $1.67\times$ and $1.35\times$ faster than MicroBlaze with C malloc in both 128KB and 10MB memory usage cases; $6.74\times$ and $7.97\times$ slower than ARM core with C malloc in both cases. For linked lists, AVL trees, hash tables and vectors, the speed of HLS relative to the ARM core with C malloc do not change much because the numbers of accesses to the data structures are bounded; whereas for unbalanced trees, the latency of the larger tree in HLS is much larger than in ARM core with C malloc because the search path can be very long.

B. Linked List

The average latency per node for linked lists is shown in Fig.2a. The linked list on ARM processor with default malloc is faster because the ARM processor is operating at a much higher frequency and also it has a 512KB L2 Cache. The

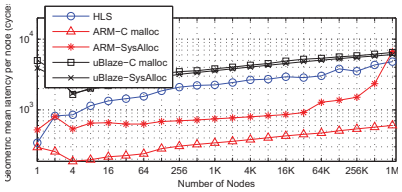


Fig. 1: Geometric means of latencies of data structures in each platform.

TABLE IV: Resource Utilisation (respect to XC7Z020 FPGA)

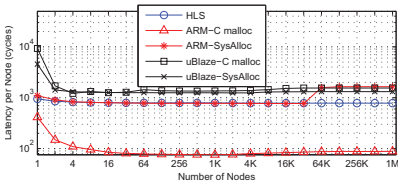
Entity	LUTs		Registers		BRAM Tile	
MicroBlaze	1774	(3.33%)	1597	(1.50%)	36	(25.71%)
Linked List	2815	(5.29%)	4331	(4.07%)	0	(0%)
Unbalanced Tree	15148	(28.47%)	25523	(23.99%)	0	(0%)
AVL Tree	53261	(100.11%)	81166	(76.28%)	1	(0.71%)
Hash Table	9136	(17.17%)	11006	(10.34%)	0	(0%)
Vector	3793	(7.13%)	4192	(3.94)	0	(0%)

linked list on the ARM with SysAlloc is slower than on ARM using default C allocator, and has similar latency to the HLS implementation. This is because the SysAlloc has much higher latency than the C-allocator as it needs to access DDR multiple times when allocating memory and there is no cache. When the list size changes from 32K nodes to 64K nodes long, there is a larger increase in the latency per node of the ARM processor linked list with SysAlloc compared to other cases. This is because, although each node actually needs $2 \times 4B$ memory, SysAlloc allocated $3B$ memory so that it could use the first 4-byte word to record the memory block size for memory de-allocation use. Hence, the actual memory usage of 64K nodes linked list is more than 512KB which is larger than the L2 cache and SysAlloc is not cache-aware. Consequently, the cache flush causes a larger increase in latency.

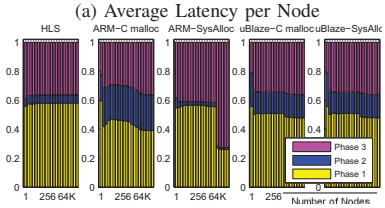
Fig.2b shows the performance break down of the linked list. For the HLS case, the proportion of time taken by each phase does not vary significantly as the linked list size increases. Phase 1 (creating list) costs approximately 55% of the execution time whilst phase 3 (deleting list) uses 38%. The creation takes time because they require SysAlloc to allocate and free memory and it is faster for SysAlloc to free memory. For the ARM processor processing a linked list on top of SysAlloc, the phase 3 latency proportion increases more significantly as size increases due to the cache flush. In comparison, when the ARM processor manages lists on the default allocator, the proportions vary smoothly, because the allocator in C is cache aware.

C. Vectors

Average latency per node for vectors is shown in Fig.3a. The ARM processor vectors have similar performance for both the SysAlloc and default allocator when the vectors are big. This is because only phase 1 makes use of the memory allocator, and the rest of phases are memory accesses of a range of already allocated memory. In terms of performance breakdown, which is shown in Fig.3b, HLS vectors spend most of the time creating the data structure, and the sequential access phase takes approximately the same portion of time; in comparison, the random access takes a larger portion of time when the vector is larger, because the memory access addresses becomes

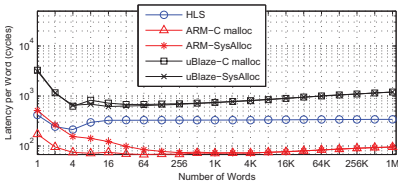


(a) Average Latency per Node

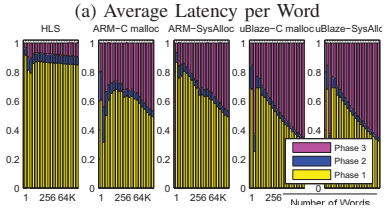


(b) Performance Breakdown

Fig. 2: Linked List Performance



(a) Average Latency per Word



(b) Performance Breakdown

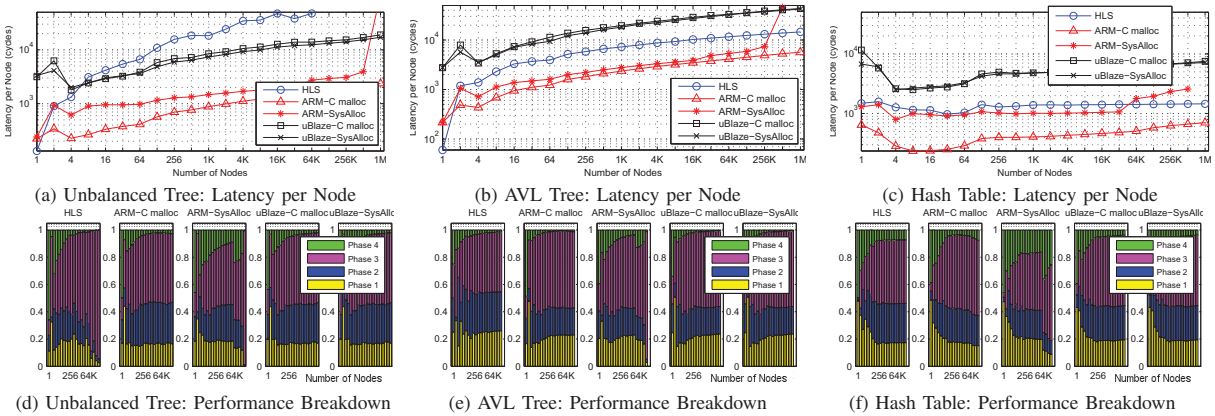
Fig. 3: Vectors Performance

more random. In software case, the portion of time taken by phase 1 is lower compared to HLS case because re-allocating is faster when a cache is used. The random access phase portions of time scale worse than the HLS case due to the need of cache flush when the maximum vector size is big.

D. Random/Sequential Input Values for Trees and Hash Tables

The performance of trees and hash table with random input values are shown in Fig.4. In terms of latency, the HLS unbalanced tree is the slowest compared to other data structures as it has unbounded search paths; it is also the slowest compared to unbalanced trees in other platforms because software processors have caches which make the memory accesses during the search much faster.

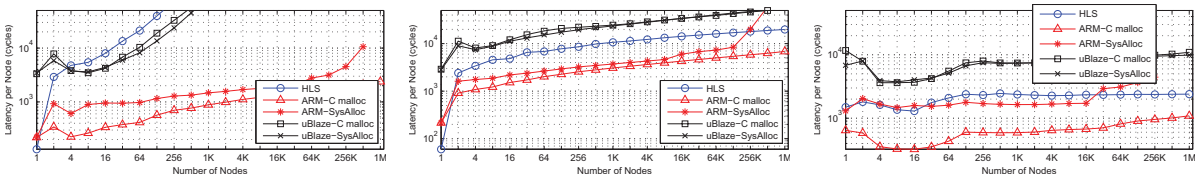
For all three data structures, phase 3 (updates) takes the most portion of time. This is because up to two look-ups in the fully-generated data structures are done in each iteration in the phase. There is also a significant increase in latencies of ARM core data structures with SysAlloc for very large sizes because of the big increase in the number of cache flushes. This is because of the randomness of memory accesses addresses become higher when the data structures are very large as SysAlloc is not cache-aware.



(a) Unbalanced Tree: Latency per Node (b) AVL Tree: Latency per Node (c) Hash Table: Latency per Node

(d) Unbalanced Tree: Performance Breakdown (e) AVL Tree: Performance Breakdown (f) Hash Table: Performance Breakdown

Fig. 4: With random value inputs, the latency per node and the performance breakdown for trees and hash tables



(a) Unbalanced Tree: Latency per Node (b) AVL Tree: Latency per Node (c) Hash Table: Latency per Node

Fig. 5: With sequential value inputs, the latency per node for trees and hash tables

The use of sequential inputs creates a pathological case for trees. The latency per node performance for trees and hash table are shown in Fig.5. As expected, the latencies of unbalanced trees in HLS and MicroBlaze scales badly with size because of the $O(n)$ search paths. Whereas for AVL trees, because of the extra self-balancing process, although the latencies for smaller sizes are higher, the performance scales better. For hash tables, the pathological case does not have a visible impact on performance.

VIII. Conclusion and Future Work

This paper presents SynADT, a methodology and design for providing shareable and composable dynamic data structures in HLS; and BenchADT, a benchmark that can be applied to evaluate both hardware and software data structures. Challenges to developing HLS data data structures such as memory management and pointer manipulations are discovered and resolved using explicit passing of allocators to API functions, and pointer-free relative addressing within data structures. The HLS data structures are evaluated by comparing against data structures in ARM core and MicroBlaze with C malloc and a hardware allocator. HLS data structures with sizes of 128KB and 10MB achieves 1.67× and 1.35× speed-up over MicroBlaze using default malloc; HLS is on average 6.74× and 7.97× slower than an ARM running at 677MHz with 512KB L2 cache with C malloc. For future work, hardware-specific optimisations such as burst mode memory access and bit-packing can be used to improve the performance of HLS dynamic data structures. In terms of heterogeneous dynamic data structures, applications can first be modelled using SynADT C in software then analysed using BenchADT to decide how to partition operations to be implemented in different platforms to better use the heterogeneous platforms.

References

[1] J. Naous, D. Erickson, G. A. Covington, G. Appenzeller, and N. McKeown, "Implementing an openflow switch on the NetFPGA platform," in *Proceedings of the 4th ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, ser. ANCS '08. ACM, 2008, pp. 1–9.

[2] F. Winterstein, S. Bayliss, and G. Constantinides, "High-Level Synthesis of Dynamic Data Structures : A Case Study Using Vivado HLS," *2013 International Conference on Field-Programmable Technology (FPT)*, pp. 362–365, 2013.

[3] Z. Xue and D. B. Thomas, "SysAlloc : A Hardware Manager for Dynamic Memory Allocation in Heterogeneous Systems," *2015 25rd International Conference on Field Programmable Logic and Applications (FPL)*, 2015.

[4] S. McConnell, *Code Complete, Second Edition*. Redmond, WA, USA: Microsoft Press, 2004.

[5] 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," *Proceedings - 1st International Congress on Image and Signal Processing, CISP 2008*, vol. 4, pp. 171–175, 2008.

[6] G. Weisz and J. C. Hoe, "CoRAM ++ : Supporting Data-Structure-Specific Memory Interfaces for FPGA Computing," *2015 25rd International Conference on Field Programmable Logic and Applications (FPL)*, 2015.

[7] F. Winterstein, S. Bayliss, and G. Constantinides, "FPGA-BASED K-MEANS CLUSTERING USING TREE-BASED DATA STRUCTURES," *2013 23rd International Conference on Field Programmable Logic and Applications (FPL)*, 2013.

[8] D. Diamantopoulos, S. Xydis, K. Siozios, and D. Soudris, "Dynamic Memory Management in Vivado-HLS for Scalable Many-Accelerator Architectures," *11th International Symposium, Applied Reconfigurable Computing 2015*, pp. 41–52.

[9] G. Marsaglia, "Xorshift RNGs," *Journal Of Statistical Software*, no. 14, pp. 1–6.