Hi-DMM: High-Performance Dynamic Memory Management in High-Level Synthesis

Tingyuan Liang, *Student Member, IEEE*, Jieru Zhao, *Student Member, IEEE*, Liang Feng, *Student Member, IEEE*, Sharad Sinha, *Member, IEEE*, and Wei Zhang, *Member, IEEE*

***Abstract*—High-level synthesis (HLS) of field programmable gate array (FPGA)-based accelerators has been proposed in order to simplify accelerator design process with respect to design time and complexity. However, modern HLS tools do not consider dynamic memory allocation constructs in high-level programming languages like C and limit themselves to static memory allocation. This paper proposes a dynamic memory allocation and management scheme, called Hi-DMM, for inclusion in commercial HLS design flows. Hi-DMM performs source-to-source transformation of user C code with dynamic memory constructs into C-source code with the dynamic memory allocator and management scheme developed in this paper. The transformed C-source code is amenable to synthesis by commercial tools like Vivado HLS. Relying on buddy tree-based allocation schemes and efficient hardware implementation of the allocators, Hi-DMM achieves 4× speed-up in both fine-grained and coarse-grained memory allocation compared to previous works. Experimental results obtained by including Hi-DMM with Vivado-HLS show that dynamic memory allocation of FPGA memory resources can be achieved at a much lower latency with minimal resource overhead, paving the way for synthesis of dynamic memory constructs in commercial HLS flows.**

***Index Terms*—Field programmable gate array (FPGA), high-level synthesis (HLS), memory management.**

I. Introduction

With the increase in complexity of designs implemented on devices like field programmable gate array (FPGA), the level of design abstraction used for description has been raised from register transfer level to high-level languages, in an effort to reduce design cost and design time. High-level synthesis (HLS) tools help designers to automatically transform the behavioral description specified in high-level languages (e.g., C/C++) into RTL-level design [1]. However, current HLS tools are unable to have comprehensive support for dynamic constructs [e.g., dynamic memory management (DMM)] in these languages. This limitation forces designers to perform code refactoring in case of legacy code and limit themselves to static memory allocation.

FPGAs come with abundant on-chip memory resource these days. However, the limited design expressibility because of limitations in support for DMM, prohibits a designer from creating designs that can maximize the utilization potential of the on-chip memory resources during run time and thus achieve higher performance. For solutions to DMM based on operating system for CPUs, efficient algorithms have been proposed by researchers, such as garbage collection [2] and slab allocation [3]. Unfortunately, these CPU-based solutions will lead to unnecessary resource overhead if transplanted on FPGA directly. Existing solutions [4]–[8] to FPGA DMM in previous works lead to high latency of memory allocation and significant area overheads. Thus, they significantly undermine the overall performance of FPGA-based accelerators. Additionally, the previous works have not sufficiently addressed the issue of fine-grained DMM, e.g., allocation of a dynamic tree node, and coarse-grained DMM, e.g., allocation of an array, which again means very limited flexibility.

To address these issues comprehensively, we propose an open-source tool, Hi-DMM, coupled with VivadoHLS, an HLS platform developed by Xilinx. Hi-DMM can automatically analyze the source code of HLS module and transform it, equipping the HLS-generated accelerator with high-performance DMM allocators, which are based on buddy tree allocation scheme and its proposed variants. The highlights of Hi-DMM are as follows.

1) It is a part of HLS methodology. The DMM components, including allocators and heap memories, are described in C and are synthesizable with commercial HLS tools like Vivado-HLS.
2) HLS accelerators can access Hi-DMM allocator via HLS handshake protocol. Most of the proposed DMM components are automatically configured for adaption to the characteristics of source code, e.g., memory allocation granularity and HLS directives.
3) It achieves high-performance memory allocation. The buddy-tree allocators search the allocable addresses by using bit-vector (BV) computation and maintain the information in parallel. Preallocation scheme, look-up table, and mini-heap are involved to minimize memory allocation latency.

Manuscript received April 3, 2018; revised June 8, 2018; accepted July 2, 2018. Date of current version October 18, 2018. This work was supported by the Hong Kong Research Grants Council General Research Funds under Grant 16245116. This article was presented in the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES) 2018 and appears as part of the ESWEEK-TCAD special issue. *(Corresponding author: Tingyuan Liang.)*

T. Liang, J. Zhao, L. Feng, and W. Zhang are with the Department of Electronic and Computer Engineering, Hong Kong University of Science and Technology, Hong Kong (e-mail: tliang@ust.hk; jzhaoao@ust.hk; lfengad@ust.hk; wei.zhang@ust.hk).

S. Sinha is with the Department of Computer Science and Engineering, Indian Institute of Technology Goa, Goa 403401, India (e-mail: sharad_sinha@ieee.org).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TCAD.2018.2857040

TABLE I
COMPARISON BETWEEN HI-DMM AND PREVIOUS DMM ALLOCATORS

Allocator	Hi-DMM	SysAlloc[8]	AMMU[9]	DMM-HLS[6]	DOMMU[5]	FLM[7]
Purpose & Features	HLS-Friendly, High Performance, Memory Utilization	Memory Utilization, Unbounded Management Capacity	Memory Utilization, High Performance	Memory Reuse among Multiple Accelerators	Memory Reuse	High Memory Utilization
Mechanism	Buddy System	Buddy System	Buddy System	Free-List	Fixed-size Allocation	Free-List+Buddy System
Clock Rate	100 MHz	150 MHz	9.6 MHz (256 MAUs)	Unknown	140 MHz (Highest)	175 MHz (Highest)
Latency	Tens of Cycles	Hundreds of Cycles	Single Cycle	20% of execution time	Unknown	Tens of Cycles

This paper is organized as follows. Section II discusses the related works. Section III presents motivation and overview of Hi-DMM. Section IV illustrates the compilation flow of Hi-DMM. Section V explains the mechanisms of Hi-DMM allocators. Section VI provides the evaluation results of Hi-DMM. Section VII concludes this paper.

II. RELATED WORK

Table I presents the comparison between the solution of Hi-DMM and related works. All the presented DMM allocators can perform memory reuse by de-allocating memory blocks for later allocation. Higher memory utilization indicates less inter- and intra-fragment. High-performance allocators have low latency of allocation.

By allocating fixed-size blocks, Dessouky *et al.* [5] proposed a hardware dynamic on-chip memory management unit (DOMMU) to manage distributed shared on-chip memory among several processing elements. The scalability of their work is limited by the resource utilization of crossbar-based interconnection and latency overhead due to arbitration. Additionally, the fixed-size allocation may result in serious memory fragmentation, memory resource waste, and difficulty for variable size of allocation.

For the purpose of higher utilization of memory resource, DMM based on free-list and first-fit algorithm has been studied by Diamantopoulos *et al.* [6]. They introduced the DMM-HLS framework that extends HLS with DMM for multiple accelerators case, maximizing the number of the accelerators through effectively sharing memory resources. However, since the first-fit algorithm has time complexity of $O(n)$, the latency of allocation increases if the granularity of allocation increases. Even for coarse-grained DMM, the latency of DMM takes up 20% of overall execution time on average.

To improve the performance of DMM, the implementation of buddy allocation algorithm on hardware has been proposed by Chang and Gehringer [4]. Based on this paper, Agun and Chang [9] presented a hardware active memory manager unit for software applications. The hardware allocator achieves an allocation latency of 1 clock cycle but increases the critical path dramatically, resulting in low clock rate. Moreover, the total overhead of latency and area of their design will increase exponentially when the depth of heap increases. Özer [7] proposed a DMM allocator, free-list manager (FLM), combining buddy tree and free-list to improve memory utilization. However, for a request of allocation, FLM will return a link list of noncontiguous memory blocks, making the allocated memory hard to be processed in parallel. Besides, an address translator for FLM costs the accelerator extra cycles to access BRAM, which originally can be accessed in one cycle. High access latency of BRAM significantly undermines the performance of accelerator. Xue and Thomas [8] proposed a framework called SysAlloc, which overcame the long critical path in [4] and achieved higher operating frequencies, by inserting registers between stages. SysAlloc can manage DDR-scale memories and enable clients to read and write memory over a shared AXI bus. However, the latency of memory allocation by SysAlloc reaches hundreds of cycles. Moreover, the shared AXI bus traffic overhead limits the scalability of the performance of SysAlloc and results in extra interconnection area.

Moreover, all these prior approaches leave the configuration of DMM components to designer and negate the effect of HLS directives that could be present in the design's description and have significant impact. In this paper, we will propose a high-performance low-overhead DMM scheme with the support of HLS directives.

Note that Hi-DMM can be operated at up to 220 MHz, at the cost of more cycles to handle requests. Through frequency optimization, we operate Hi-DMM at 100 MHz to perform lower total overhead, which is frequency times the number of latency cycles. Considering allocation latency, memory utilization, portability to HLS applications and single-cycle access of BRAM for accelerator, we choose SysAlloc as the baseline of Hi-DMM for later comparisons.

III. FRAMEWORK OVERVIEW

For DMM schemes, the implementation of the allocators themselves can benefit from HLS methodology. The flow of allocation mainly consists of two steps: 1) searching available memory to satisfy the allocation request and 2) maintaining the allocation information. Both steps can be accelerated with the support of directives in HLS, e.g., loop pipeline and array partitioning.

Hi-DMM is designed to be high-efficient DMM framework based on HLS and for HLS. The overall framework of Hi-DMM consists of the compilation of a design with DMM and runtime mechanism of DMM. The compilation flow, shown in Fig. 1, which analyzes the characteristics of the source code of an accelerator, determines the configuration of DMM runtime mechanism, shown in Fig. 2.

The input of compilation flow is a C/C++ application with DMM APIs, as an example shown in Code 1. The output of the

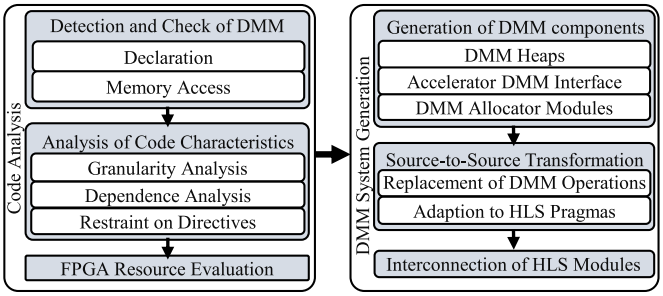


Fig. 1. Compilation flow of Hi-DMM.

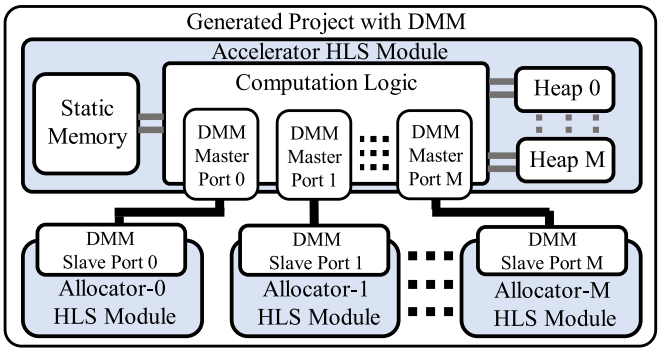


Fig. 2. Runtime DMM mechanism of Hi-DMM.

```
void accelerator(int input_data[1024],
        int output_data[1024], int n,int m){
#pragma HLS ARRAY_PARTITION variable=c cyclic \
                                factor=4 dim=1
  int *a = malloc(n*sizeof(int)); //(VGA)
  int *b = malloc(m*sizeof(int));
  int *c = malloc((1024)*sizeof(int));//(CCGA)
  for (i = 0; i < n; i++) {
    a[i] += b[j]; //interaction between pointers
    for (j = 4; j < 64; j++){
#pragma HLS unroll factor=4
#pragma HLS array_partition variable=c  \
                                factor=4  dim=1
      c[j] = input_data[j] + c[j-4];
    }
    free(c); // CCG allocation can also be freed
    node *root_node=malloc(sizeof(node));//(CFGA)
    root_node->left_son = malloc(sizeof(node));
    root_node->data = 1;//user-defined struct
                        // pointer
    free(old_node);
  }
  free(a);free(b);
}
```

Code 1. Example of HLS source code for DMM.

flow is a generated project where accelerator and DMM allocators, both based on HLS, interconnect and interact with each other. The compiler first checks DMM syntax, analyzes characteristics of DMM behavior involved and evaluates available FPGA resource. Next, according to the result of analysis, the compiler configures DMM components, generates related HLS modules and transforms DMM-related part in source code, as an example of transformed code shown in Code 2. At the end of DMM system generation, a Tcl script is used to connect HLS generated modules into a Vivado block diagram. The details of compiler are discussed in Section IV.

The foundation of Hi-DMM is based on optimized buddy tree-based allocators. The runtime mechanism of these allocators is also described in C, with parameters and operations tuned for high performance. To meet various requirements

```
//---------- Declaration of Hi-DMM Components ----------
int HTA0_heap[1024];
ap_uint<16> KWTA0_heap[4096];
#define NODE_LEFT_SON 0
#define NODE_DATA 2
void accelerator(int input_data[1024],
        int output_data[1024], int n,int m,
        volatile DMM_port *HTA0, //DMM Ports
        volatile DMM_port *KWTA0)
{
#pragma HLS interface ap_hs port=HTA0
#pragma HLS interface ap_hs port=KWTA0

//---------- Initialization of Hi-DMM Variables ----------
#pragma HLS ARRAY_PARTITION variable=HTA0_heap  \
        cyclic factor=4 dim=1// heap partition
  int offset_a,offset_b,offset_c, offset_root_node,
      offset_root_node_left_son, offset_old_node;

//---- Example of Coarse-Grained DMM Allocation ----
  offset_a = HTA0_malloc(n); // C malloc replaced by
  int *a=HTA0_heap+offset_a;
  offset_b = HTA0_malloc(m);  // Hi-DMM API
  int *b=HTA0_heap+offset_b;
  offset_c = HTA0_malloc(1024);
  int *c=HTA0_heap+offset_c;
  for (i = 0; i < n; i++)
  {
    a[i] += b[j];

//------ Example of Hi-DMM Loop Transformation ------
    int *loop0_c = HTA0_heap+((offset_c +3)/4)*4;
    int separate_loop0 = (loop0_c - c) % 4;

    for (j = 0; j < separate_loop0; j++) // first
        c[j] = input_data[j] + c[j-4];   // loop


    for (j = 0; j < 64; j++)
    { // second loop
#pragma HLS unroll factor=4
      if ( j<64-separate_loop0)
      {
        loop0_c[j] = input_data[j+separate_loop0] +
            loop0_c[j-4];
      }
    }
    HTA0_free(offset_c); // de-allocation
                         // based on offset

//---- Example of Hi-DMM Fine-Grained Allocation ----
    int offset_root_node = KWTA0_malloc();
    int *root_node =  KWTA0_heap+offset_root_node;
    int offset_root_node_left_son = KWTA0_malloc();
    int *root_node_left_son = KWTA0_heap +
        offset_root_node_left_son;

//-- Example of Operation on User-Define C-Struct --
    root_node[NODE_LEFT_SON] =
        offset_root_node_left_son;
    root_node[NODE_DATA] =  1;

//---------- Example of Hi-DMM De-Allocation ----------
    KWTA0_free(offset_old_node);
  }
  HTA0_free(offset_a);HTA0_free(offset_b);
}
```

Code 2. Example of output code of Hi-DMM.

of DMM, multiple allocators, with different parameters and variations in allocation algorithms, could be integrated into the generated project. An accelerator is connected directly to allocators using HLS handshake protocol so that the former can access the latter with lower latency and at a less cost of area than the AXI4 protocol. Note that each allocator can handle request sent from an accelerator independently, to support concurrent memory allocation. If multiple accelerators can be grouped into a top module, they can share an allocator. Detailed mechanisms are discussed in Section V.

IV. Hi-DMM Compilation Flow

The Hi-DMM compiler is composed of an analyzer of source code, and a generator for FPGA projects with DMM. The analyzer checks the DMM behaviors and computes the statistics, required by the project generator to transform the source code and configure DMM components.

A. DMM Analyzer

In the view of source code, granularity of memory allocation and co-operations among dynamically allocated pointers are more critical to the performance of DMM. As for HLS on FPGA, compatibility with directives in HLS and available resource on the target FPGA need to be evaluated to ensure the DMM solution suitable for the specified hardware.

1) Granularity of Memory Allocation: According to the size of memory requested, each function call of DMM will be classified into three different types, constant-coarse-grained allocation (CCGA), constant-fine-grained allocation (CFGA), and variable-grained allocation (VGA). The parameters of CCGA are known during compilation and the size of requested memory is large, compared to CFGA. CFGA is similar to CCGA but the size of CFGA is relatively small, e.g., an element in a list. As for VGA, the parameters of allocation are unknown during compilation. As shown in Code 1, allocated array pointers *a*[] and *b*[] are based on VGA while *c*[] is a CCGA-based pointer. Many data structures, e.g., tree and list, involve frequent CFGA. As an example, root_node in Code 1 is allocated via CFGA.

2) Co-Operations Among Dynamically Allocated Pointers: The co-operation between a pair of dynamically allocated pointers refers to the operations involving both of them. These co-operations can be detected via Clang, which is the frond-end of LLVM [10]. The extent of co-operation between two pointers is evaluated by the number of direct co-operations involving both of them, in the source code. The output of such procedure is a fully connected graph of pointer variables, where the weight of an edge represents the extent of dependency between the two vertices (pointers) joined by that edge. As shown in Code 1, the allocated array pointer *a*[] co-operates with *b*[] frequently while the allocated array pointer *c*[] is not involved in the direct calculation with other pointers.

3) Compatibility With Directives in HLS: HLS directives are crucial for HLS designs since they guide the tool in synthesizing the source code into FPGA-friendly RTL design with higher performance (higher parallelism and lower area utilization). Since the dynamically allocated memory (DAM) could be involved in the code sections that have directives, the analyzer will check whether the DAM can be compatible with the directives, thus ensuring successful synthesis. Additionally, the check on directives also guides the DMM project generator to configure the parameters of heaps and allocators. As shown in Code 1, if the heap where *c*[] is allocated is not partitioned or there is no transformation of loop structure, the loop unrolling will not result in high parallelism in computation as expected.

4) Available Resource on Target FPGA: The available resource on target FPGA constrains the implementation of DMM. The analyzer will estimate how much resource, especially BRAMs, will be involved in the implementation of accelerators excluding the DMM. A report is generated to the generator indicating how much resource could be left for DMM, including heaps and allocators.

B. DMM Project Generator

The automatic generator is based on libraries and templates. According to the statistics collected by the analyzer, the generator maps the DMM calls in the source code to heaps, configures the parameters of heaps, selects proper DMM algorithms for each allocator and finally generates the project integrated with DMM.

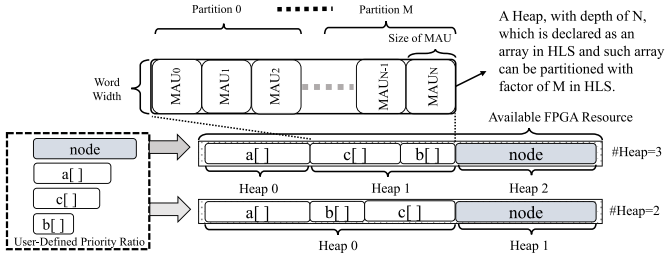


Fig. 3. Example: configuration of heaps with Hi-DMM.

1) Generation of DMM Components: Heaps will be instantiated as arrays in HLS source code and accelerator requests memory allocation in the heaps from allocator via DMM interface. The heaps are generated with the following parameters: 1) the heap depth [the number of minimum allocable units (MAUs) in the heap]; 2) the size of a single MAU; 3) the heap word width (the number of bits of every single word of heap); 4) the number of partitions of heap; and 5) the data type of elements, e.g., float. An example is shown in Fig. 3.

The generator needs designers to set some flexible and critical parameters. There can be multiple heaps for the application with different heap word width. More heaps will enable more parallel accesses but require more resources. Hence, given a range of possible heap numbers, the designer can set the heap number according to the desired performance-area tradeoff. The designer can also set the size of MAU in a heap which is the most basic block for allocation. Note that larger MAUs can shrink the depth of heap and improve the performance of allocation but undermine the utilization of memory due to fragmentation. The designer can also provide a priority table which indicates which call of allocation in source code could be more intensive than others, according to prior experience. This table can be used to determine the corresponding heap depth. Otherwise, the generator will set the same depth for all the heaps, which might lead to resource contention for hot-spot allocations. Moreover, heaps can be partitioned according to the designer-configured directives in the source code. An example is illustrated in Section IV-B2 for detailed heap partition.

The other parameters of heaps will be handled by the generator automatically as described next. First, the heap word width and the type of elements are determined by the allocation function calls in HLS source code. Then, before evaluating the proper depth for each heap, the assignment of allocation request to heaps should be determined. An example of such a procedure is shown in Fig. 3. Allocation requests with different word widths and data types will be assigned to separate heaps while the requests, whose two characteristics are the same, may share a heap. However, to allocate pointers in the same heap implies VivadoHLS that there could be unknown data dependencies, resulting in conservative schedule during synthesis [11]. Moreover, the concurrent accesses to a heap

TABLE II
COMPARISON OF HI-DMM ALLOCATORS

Allocator	FBTA	PATA	HTA	KWTA
Machanism	OR-gate tree		group tree	AND-gate tree
Allocation Latency	7 - 10 cycles	3 or 9-12 cycles	7-8 or 12-19 cycles	1 or 9 cycles
Maintenance Latency	7-18 cycles	24-26 cycles	7-24 cycles	3-4 cycles
Management Capability	<=512 MAUs		<=64k MAUs	>2048 MAUs
Allocation Type	variable size			fixed size
Allocation Granularity	coarse		coarse / fine	fine

are limited by the number of heap ports. Therefore, in order to increase parallelism, the generator will do its best to assign the pointers to different heaps. Those pointers involved in cooperations frequently have higher priority to be assigned to different heaps, while the pointers less related to each other, will be assigned to a shared heap. The assignment of pointers to heaps is done via a greedy algorithm based on the graph provided by the analyzer.

Karger's algorithm [12] is applied on this graph to find the weighted max-cut and separates the graph into two subgraphs by the max-cut. For the subsequent iterations, Karger's algorithm finds the max-cuts in the subgraphs and removes the maximum one from the corresponding subgraph, thus increasing the number of subgraphs. The greedy procedure will cease when the number of subgraphs is equal to the specified number of heaps. The pointers in a subgraph will be assigned into the same heap. As the example shown in Fig. 3, the generator allocates the array pointers *a*[] and *b*[] in different heaps when the number of heaps is enough, because *a*[] and *b*[] have stronger relation. Moreover, for most applications, CFGA requests usually occur more frequently than the allocation requests with other granularities. In order to improve the performance of the CFGA management, the requests of CFGA will be assigned to the special heaps which are distinct from those for the allocation requests of other granularities. After assignment, based on the priority table, the priority of allocation call will be accumulated to the heaps and those heaps with higher priority can be distributed with more BRAM resource.

Finally, according to the assignment of allocations, the selected allocator as described in the next paragraph, parameters set by designer, and total available resources, the maximum allowed depth for each heap can be calculated.

In the implementation of Hi-DMM, the DMM allocators are also HLS modules separate from the accelerator. To avoid conflicts of data and control, each allocator accounts for a single heap. Hi-DMM provides a library of allocators, including fast buddy tree allocator (FBTA), preallocation tree allocator (PATA), hybrid tree allocator (HTA), and *K*-way tree allocator (KWTA) which are compared in Table II. The detailed implementation of allocators' mechanism will be discussed in Section V. Each of these allocators has unique characteristics so that they are suitable for different requirements of applications, e.g., various granularities of allocations, interval of allocations, and the number of MAUs (i.e., the depth of heap).

DMM project generator will select one of these allocators for each heap and configure it with the parameters from analyzer. For the selection of allocators, Hi-DMM has a table that contains the details of FPGA resource consumption, performance and management capability of the allocators. Based on this table, Hi-DMM takes the following factors into consideration: allocation granularity, FPGA resource constraints, the management capability of allocators, the performance and resource requirement of allocators themselves, etc.

For example, in Code 1, the allocations of tree nodes are types of CFGA and KWTA achieves lower latency for these frequent allocations and requires less area for the allocator itself. However, since KWTA is an allocator for fixed-size memory blocks, when handling allocation request with variable size, it will lead to high memory fragmentation for the heap. Therefore, for allocation of *a*[] and *b*[] which requires VGA, FBTA, PATA, and HTA are better candidates. If the number of MAUs is large, Hi-DMM will choose HTA for these dynamically allocated arrays. Note CCGA, e.g., *c*[] can share heap with VGA.

For the communication between allocators and accelerator, I/O ports in the top module of accelerator, connected to allocators, will be instantiated according to the number of allocators. Based on HLS handshake protocol, these ports are implemented with associated valid and acknowledge signals to provide a two-way handshake.

2) Source-to-Source Transformation: Source code transformation makes the HLS accelerator adapt to DMM components and guarantees high DMM performance. Code 2 is an example of the output of transformation.

The purpose of the replacement of function calls is to assign allocation function calls to specified allocators, which has been determined by the process of heap generation. The original function *malloc((size_t)* and *free(void *)* will be replaced by the particular API functions in the Hi-DMM library, e.g., *HTA0_malloc(int size, DMM_port* port)* and *HTA0_free(int addr, DMM_port* port)*.

Moreover, since Hi-DMM supports the dynamic allocation of user-defined struct ("struct" in C), the accesses of the components of these structures are also transformed into expressions based on pointers and offsets.

It is important to ensure that the directives, involving dynamically allocated pointers, are supported during the HLS but VivadoHLS cannot support the DAM partition working with loop unrolling. For static arrays, array partitions can be mapped to corresponding iterations of loop and if the loop is unrolled, the accelerator can benefit from the high bandwidth [13]. However, the partitions of DAM cannot map to the loop iterations because DAM have uncertain offset in the heap which means the starting address of the array in the heap is unknown. To overcome this issue, Hi-DMM first partitions the heap where the partitioned dynamic array is allocated, and then separates the original loop into two loops. The first loop will process a few of elements and the second loop always starts processing from the element in the partition zero (e.g., *P*0 in Fig. 4) so that VivavdoHLS can know which partition the following element

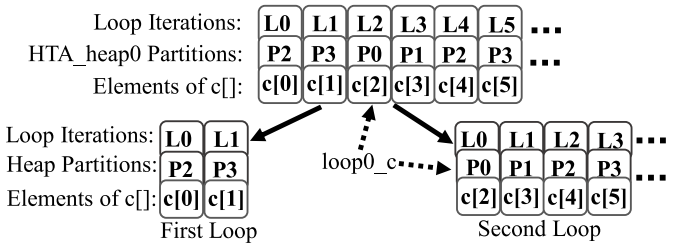


Fig. 4. Array partitioning and loop unrolling with Hi-DMM (partition factor = unroll factor = 4, offset of c[] = 2).

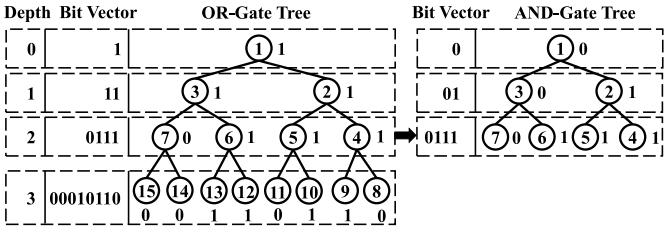


Fig. 5. OR-gate tree and AND-gate tree based on eight MAUs.

is in and map partitions to the iterations of the second loop.

An example of loop transformation is shown in Fig. 4 and Code 2, where the partition factor of *HTA0_heap* is 4 and the offset of *c*[] in *HTA0_heap* is 2. First, according to the directive of cyclic loop partition, the heap will be partitioned equally into four partitions. The array elements in each of the partition are shown as in Fig. 4. We derive the equation and make the resultant pointer *loop0_c* point to *c*[2], which is the element located in the partition *P*0 of *HTA_heap0*. *separate_loop0* is the array index of *c*[2] which is 2. Therefore, the second loop after transformation will start with accessing the partition 0 of *HTA0_heap* certainly. As a result, VivadoHLS can map the elements in *loop0_c* to the iterations of the second loop so that loop unrolling will be successful.

After the generator finishes, a Tcl script is used to interconnect the accelerator and the allocators, handle ports and wires automatically, and finally generate a complete project with DMM.

V. Hi-DMM Allocators

This section discusses the implementation of Hi-DMM allocators.

A. Conventional Buddy Tree Allocator

As shown in Fig. 5, buddy tree allocator splits the entire space of heap repetitively in half, to find an available memory block best fitting the size of request. Based on the buddy tree, the number of MAUs managed by the allocator is a power of 2, and the resultant two blocks from a split are buddies, which can be merged into a double-size block when both are available for larger requests.

1) Representation of Buddy Tree: Proposed by Chang and Gehringer [4] and adopted by Xue and Thomas [8], OR-gate BVs and AND-gate BVs are used to maintain the memory usage of the heap and one BV accounts for the information of a particular depth of the buddy tree. Supposed that the allocator is designed to manage 2^D MAUs. Accordingly, the buddy tree will have a depth of D, as the tree shown in Fig. 5, which has a depth D of 3 and can manage 8 MAUs. Each node located in the layer at the depth of D_L represents a memory block with size of 2^{D-D_L} MAUs and a 2^{D_L}-bit vector BV_{D_L} of the layer can be used to record the status of these nodes, in which the ith node is mapped to the bit $\mathrm{BV}_{D_L}(i)$. A bit in the OR-gate BV of the layer, $\mathrm{BV}_{D_L}^{\mathrm{OR}}(i)$, indicates the ith memory block in this layer is empty when it is 0, otherwise, the value of the bit will be 1. As for a bit in the AND-gate BV of the layer, $\mathrm{BV}_{D_L}^{\mathrm{AND}}(i)$, it will be 1 only if the corresponding memory block is full. Based on this, the allocator will handle the (de-)allocation requests via two stages, (de-)allocation stage and maintenance stage.

For conventional buddy tree, each layer will be first described with a BV, $\mathrm{BV}_{D_L}^{\mathrm{OR}}$ based on OR-gate from leaves to root. The leaves of buddy tree represent the availability of MAUs and an entire OR-gate tree can be built from the leaves to the root according to the following equation:

$$\mathrm{BV}_{D_L}^{\mathrm{OR}}(i) = \mathrm{BV}_{D_L+1}^{\mathrm{OR}}(2i) \cup \mathrm{BV}_{D_L+1}^{\mathrm{OR}}(2i+1). \quad (1)$$

The OR-gate tree can indicate the availability of memory blocks in the D_Lth layer with $\mathrm{BV}_{D_L}^{\mathrm{OR}}$, where the first zero bit represents an available block for the request with size ranging from 2^{D-D_L} to $2^{D-D_L-1}+1$.

In order to find the first zero bit, the specified OR-gate BV will be fed into an AND-gate tree as leaves. For example, in Fig. 5, the second layer of OR-gate tree will be the input of AND-gate tree, when searching a 2-MAUs block for allocation. Only when both children nodes are allocated, the node can be marked as full. Therefore, AND-gate buddy tree can be constructed according to the following equation:

$$\mathrm{BV}_{D_L}^{\mathrm{AND}}(i) = \mathrm{BV}_{D_L+1}^{\mathrm{AND}}(2i) \cap \mathrm{BV}_{D_L+1}^{\mathrm{AND}}(2i+1). \quad (2)$$

The AND-gate tree can propagate availability information upward toward the root in such a way that the location of the first zero in $\mathrm{BV}_{D_L}^{\mathrm{OR}}$ can be found by a nonbacktracking search from the root to the target leaf by tracking the child node with a value of 0, with a time complexity of $O(\log(N))$.

2) Allocation Stage: Suppose the size of the incoming request ranges from 2^{D-D_L} to $2^{D-D_L-1}+1$. The allocation stage begins with the search based on AND-gate tree. Then, if the first zero bit is found at the ith lowest position of the bit-vector $\mathrm{BV}_{D_L}^{\mathrm{OR}}$, the resultant memory address will be $2^{D-D_L}\cdot(i-1)$. For example, if searching for a 2-MAUs block, the allocator will go through nodes 0, 3, and 7 of AND-gate tree. Then, because node 7 is the fourth node at the depth of 2, the allocated address should be 6. After allocation, the allocated bits in the corresponding BVs will be flipped.

3) Deallocation Stage: Since the information of previous allocation has already been recorded during allocation stage, the action of freeing a memory block flips the corresponding bit in the OR-gate BV and takes much less time than allocating one.

4) Maintenance Stage: Marking downward and marking upward in this stage maintain the allocation information for latter requests. The procedure of marking downward starts

from the layer of OR-gate buddy tree, which is located in allocation stage according to the request size, to the bottom layer. During marking downward after allocation, for each layer, the bits covered by the allocated block will be set to 1. In contrast, those bits will be set to be 0 after freeing. For instance, if the node 7 of OR-gate tree in Fig. 5 is allocated, nodes 14 and 15, will be marked with 1.

To mark upward, all the upper layers need to be updated, by setting their BVs according to (1) and (2).

As noticed, the search of allocable memory block and the maintenance of the buddy tree are the major workload of buddy tree allocators and both of these procedures are the performance bottlenecks for the allocator.

B. Proposed Hi-DMM High-Performance Allocator

To get rid of long critical path of [4] and high latency cycles of [8], Hi-DMM makes full use of the techniques of HLS. As presented in Section IV-B, we propose four variants of conventional buddy-tree-based allocators: 1) FBTA; 2) PATA; 3) HTA; and 4) KWTA. These allocators can be grouped for one design, thus providing designers with the flexibility to adapt DMM to various behaviors in one application. All these allocators are described in C and implemented in HLS. The detailed implementation of these allocators will be presented in this section.

1) Fast Buddy Tree Allocator: To overcome the limitations of previous works, some HLS techniques are applied to the representation of buddy tree and the mechanism of allocation and maintenance.

a) Buddy tree without AND-gate tree: As mentioned in Section V-A, previous buddy tree allocators can get trapped in the search of the first zero bit in the OR-gate BV. In [4], the AND-gate and the OR-gate trees are described as combinational logic resulting in long critical path. The work in [8] inserted registers to break the critical path but led to hundreds of steps to perform the search. Without AND-gate tree in [8], FBTA extracts the first zero bit via direct bitwise operations on the OR-gate BV [14], which is illustrated in the next.

b) Allocation stage based on bitwise and arithmetic operators: By inverting the bit-vector $\mathrm{BV}_{D_L}^{\mathrm{OR}}$ into BV', the first-zero search problem is turned into finding the lowest set bit of BV'. By subtracting 1 from BV', we can clear the lowest set bit of BV' but all the other one bits in BV' remain set. Thus, $[\mathrm{BV}'\&\mathrm{inverse}(\mathrm{BV}'-1)]$ consists of only the lowest set bit of BV'. These operations can be grouped into the following equation:

$$\text{Let } \mathrm{BV}' = \mathrm{inverse}\left(\mathrm{BV}_{D_L}^{\mathrm{OR}}\right) \tag{3}$$

$$\mathrm{MASK}_{\mathrm{first\_zero}} = \mathrm{BV}'\&\mathrm{inverse}(\mathrm{BV}'-1). \tag{4}$$

$\mathrm{MASK}_{\mathrm{first\ zero}}$ in (4) will indicate the lowest zero with 1 while the other bits will be 0. For example,

$$\text{if } \mathrm{BV}_{D_L}^{\mathrm{OR}} = b'00111011$$
$$\mathrm{MASK}_{\mathrm{first\_zero}} = b'00000100.$$

$\mathrm{MASK}_{\mathrm{first\ zero}}$ can be translated into the actual location number of the first zero bit in the bit-vector $\mathrm{BV}_{D_L}^{\mathrm{OR}}$, with an MUX-based base 2 logarithm calculator. In this way, the latency of searching the first zero bit decreases significantly from the time complexity of $O(\log N)$. For instance, to find the first zero bit in a 64-bit vector, the latency of allocation stage based on FBTA is only four cycles at 100 MHz. By grouping these 64-bit vectors and searching concurrently among them, FBTA can manage hundreds of MAUs much quicker. Note that the location of layer and the communication between accelerator and allocator cost 4 cycles so that the overall latency of allocation stage will be 8 cycles.

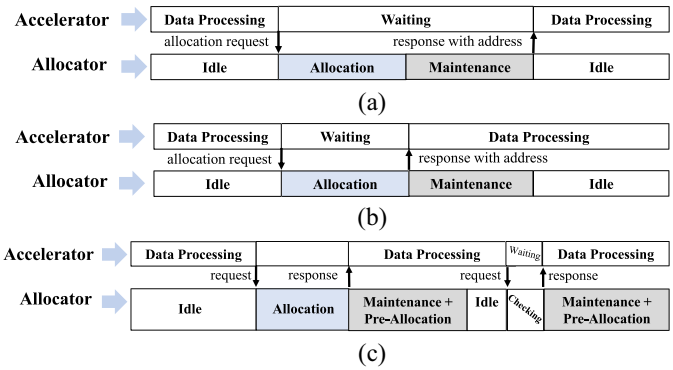


Fig. 6. Parallelism between allocator and accelerator. (a) SysAlloc: accelerator needs to wait for the maintenance of allocator. (b) Hi-DMM: accelerator operations overlap with the maintenance of allocator. (c) Hi-DMM: preallocation scheme.

Moreover, in previous works, the accelerator needed to wait for the allocator to finish not only allocation but also maintenance, as shown in Fig. 6(a). However, since the maintenance stage of allocator does not rely on any information from the accelerator, overlapping the maintenance of allocator with the subsequent actions of the accelerator can hide the latency of maintenance stage from the accelerator, as shown in Fig. 6(b).

c) Maintenance stage based on parallelism: In the implementation of [8], marking upward is done after marking downward. However, these two procedures actually process different objects and there is no dependence between them. Thus, in FBTA, they are executed concurrently. The work in [8] inserted registers between stages manually which may not be the optimal solution and can be replaced by HLS optimized solution. In Hi-DMM, the process of maintenance is pipelined by using HLS directive thus achieving higher hardware parallelism, which could be hard to implemented at RTL level by manual analysis. With these improvements, as an instance, to maintain a buddy tree with depth of 7, it only costs FBTA 7 cycles on average at 100 MHz.

2) Preallocation Tree Allocator: Although FBTA improves the performance of allocation to a large extent compared to previous works, the latency of allocation stage can still be a bottleneck when there are frequent requests for DMM allocation.

a) Allocation locality: By analyzing applications with DMM, we notice that most requests of DMM allocation present allocation locality, in both the temporal aspect and spatial aspect. Temporal locality of allocation means that when one request of allocation occurs, it is usually followed by multiple allocation requests. Spatial locality of allocation

indicates that when one request of allocation occurs, the following requests are usually in the same type of allocation and they are likely to request for similar size of memory. These characteristics of allocation locality are general results from the initialization of accelerator and DMM allocation in loops.

b) Preallocation scheme: Based on this characteristic of applications, preallocation scheme is proposed to find the proper address for the same type allocation with similar size before the arrival of next request so that the latency of searching address can be hidden for those requests.

The scheme of preallocation is shown in Fig. 6(c). Suppose that the allocator has just finished an allocation of memory block, size of which ranges from 2^{D-D_L} to $2^{D-D_L-1}+1$. After the maintenance, allocator will try to preallocate a memory block at the same layer of buddy tree but information of which will not be marked in the buddy tree. Such preallocation can also be hidden from the requesting accelerator. When a new request comes, the allocator will first check whether the request size is in the range from 2^{D-D_L} to $2^{D-D_L-1}+1$. If the size of request hits the memory block preallocated, the address for the allocation will be provided immediately without waiting for address searching. Thus, the latency of allocation stage will be just the time for checking the size and the overhead of communication, which is 3 cycles in Hi-DMM. Then, the maintenance will mark the preallocated memory block in the buddy tree. Otherwise, if the size of request is out of the range, the preallocation will be abandoned and the allocator will start the search of the first-zero bit.

3) Hybrid Tree Allocator: Although FBTA and PATA provide high-performance solution to DMM, when the number of MAUs is raised to more than 512, their consumption of area will be increased notably because of the large width of BVs array in HLS and high parallel computation between them. Therefore, FBTA and PATA perform quite well for coarse-grained allocation, such as the allocation of arrays, for which designers can define MAU with large size, for example, 1 KB, so that there would not be too many MAUs. However, the solution based on large MAU is not suitable for fine-grained allocation because it will lead to serious memory fragmentation issues resulting in wasted memory. To overcome this limitation, HTA, provided by Hi-DMM, adopts the basic hierarchical idea based on *group tree* from SysAlloc to store the information of buddy tree, which can save a lot of area. However, HTA does not rely on the search based on AND-gate tree so that it takes less than 20 cycles to accomplish the allocation request for thousands of MAUs, instead of suffering the long allocation latency (hundreds of cycles) of SysAlloc.

a) Structure of hybrid tree: The hybrid tree consists of two different kinds of layers, trunk layers and branch layers, as shown in Fig. 7. The depths of trunk and branch parts of a hybrid tree are D_T and D_B, respectively. Based on the hybrid tree structure, HTA can manage $2^{D_T+D_B}$ MAUs. The trunk part of hybrid tree is described in the same way as FBTA but leaves in the trunk part, e.g., the nodes 4–7 in Fig. 7, are mapped to a corresponding *group tree* in branch layer, indicating whether a large memory block is available. As for the branch layers, *group trees* are used to manage fine-grained

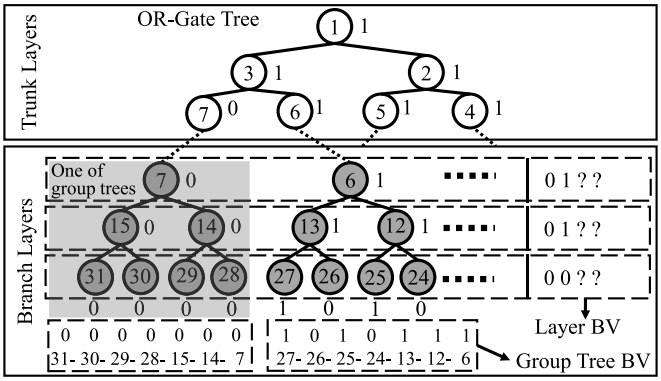


Fig. 7. HTA: trunk layers and branch layers.

allocation and each of their leaves is mapped to an MAU. To locate an available memory node with small size efficiently, these *group trees* are described by two different kinds of BVs, layer BV and *group trees* BV. Each branch layer has its own layer BV, and each bit of layer BV accounts for the status of a specific layer of a *group tree*, indicating whether the *group tree* has available nodes in the specified branch layer. For example, in Fig. 7, both of the first two layer BVs (01 . . .) indicate that the first *group tree* has available node in the layers while the second *group tree* does not. In contrast, the third layer BV (00 . . .) indicate that both two *group trees* have available nodes in the third layer. Each *group tree* in the branch part can be described within a *group tree* BV, where each bit indicates the status of a node of the *group tree*. Therefore, if the tree in trunk layer has 2^{D_T} leaves, there will be 2^{D_T} *group trees* in branch part and the width of each layer BV will be 2^{D_T}. Similarly, the width of *group tree* BV should be $2^{(D_B+1)}-1$.

b) Allocation stage of HTA: The requests of allocation are classified into coarse-grained and fine-grained according to their sizes. A request, size of which is greater than 2^{D_B-1}, is coarse-grained. The allocation for this request will only involve the trunk part of hybrid tree. The process of coarse-grained allocation will be the same as FBTA. In contrast, fine-grained allocation will involve *group trees* in branch part of hybrid tree. According to the request size, HTA locates the target layer for allocation in branch part. Based on the corresponding layer BV, HTA can find which *group tree* has space for the request by searching the first zero bit in the layer BV. Finally, techniques of FBTA will be applied on the located *group tree* and the address of allocable block will be obtained. Since HTA conducts the search of first zero bit twice for fine-grained allocation, the latency of allocation stage will be about two times more than FBTA.

c) Maintenance stage of HTA: The maintenance for trunk part and the *group tree* BV of branch part can be implemented according to the definition of OR-gate tree. Note that the maintenance after allocation of a node in the buddy tree is known during compilation. Therefore, Hi-DMM maintains the *group tree* BV by applying an OR operation with a mask BV to it. These mask BVs can be grouped into a look-up table so that the maintenance of *group tree* BVs requires lower latency. For example, if node 14 is allocated, then nodes 7, 28, and 29

should be marked as 1 and the corresponding mask BV is 0011011. As for the layer BVs, when the request is coarse-grained, all of them are only involved in marking downward and the operation applied on the BV of leaves in trunk layer can be passed to these layer BV directly. However, when the request is fine-grained, each layer BV for branch layer needs to be handled according to the status of the *group tree*. Since only the bits for the target *group tree* in layer BVs are involved in maintenance, the computation overhead is low. Moreover, since the layer BVs are independent of each other, HTA can update them concurrently.

4) K-Way Tree Allocator: The allocation for the components in dynamic data structures, e.g., tree, are fine-grained and frequent. Therefore, even FBTA, PATA, and HTA can be the performance bottleneck for these accelerators. However, most requests for these allocations are usually C struct with constant size. Considering this characteristic, Hi-DMM provides designers with high-performance fixed-size allocator based on *K*-way tree which can handle more than 2048 fixed-size blocks.

a) Structure of K-way tree allocator: The structure of KWTA is a *K*-way AND-gate tree with depth of 2 instead of the binary OR-gate tree for other allocators in this paper. An example of a 8-way tree is shown in Fig. 8. The first layer is summary layer and the second one is leaf layer. Each node marked with zero in summary layer has available child (children) and each node marked with zero in leaf layer represents an available mini-heap, a small memory space with constant size, which can hold one or more fixed-size blocks. Each mini-heap is maintained by two registers, *Allocated* and *Freed*. *Allocated* records how many fixed-size blocks have been allocated in the mini-heap, no matter whether they have been freed or not, while *Freed* records how many fixed-size blocks have been freed from the mini-heap.

b) Allocation stage of KWTA: By applying the search of the first zero bit twice on the BVs of *K*-way AND-gate tree, an available mini-heap will be located. According to the offset of the leaf $\text{Offset}_{\text{leaf}}$, the size of mini-heap $\text{Size}_{\text{mini-heap}}$ and the register *Allocated*, the allocated address will be $[\text{Offset}_{\text{leaf}} \cdot \text{Size}_{\text{mini-heap}} + \text{Allocated}]$. Note that the allocation in a mini-heap is one-way, which means the fixed-size blocks will be allocated from the head of mini-heap to the tail, according to the increasing value of the register *Allocated*, even if there are some freed space at the head. In this way, the latency of allocation stage will be decreased. Moreover, the mini-heap will be recorded if the value of its register *Allocated* is smaller than the size of mini-heap, so that the next request will not need any search. The allocation of a new mini-heap will cost 9 cycles based on 64-way tree. However, if the allocation can be handled by the recorded mini-heap, the latency will be just 1 cycle. As the size of mini-heap increases, the average latency of allocation stage will decrease significantly, because each allocated heap can be recorded for several subsequent requests. For example, when the size of mini-heap is 16, the average latency of allocation stage is 1.5 cycles.

c) Maintenance stage of KWTA: After each allocation in the mini-heap, the value of register *Allocated* will be increased by 1. If the value of register *Allocated* is equal to the size of mini-heap and the value of register *Freed* is smaller than that of register *Allocated*, the corresponding leaf in *K*-way tree will be marked with 1, indicating that it is unavailable. In contrast, after each de-allocation, the value of register *Freed* will be increased by 1 and if the value of register *Freed* is equal to that of register *Allocated*, the corresponding leaf in *K*-way tree will be marked with 0 and both registers will be reset to 0. The summary layer will be updated according to the definition of AND-gate tree. In other situations, the value of the leaf in *K*-way tree will not be changed.

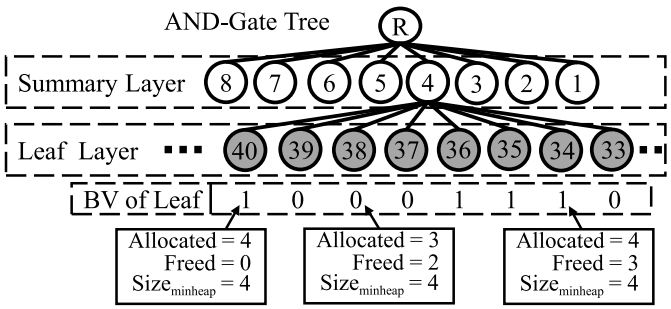


Fig. 8. KWTA: summary layer and leaf layer.

As an example presented in Fig. 8, the BV value of node 34 is 1, indicating the mini-heap is unavailable, because the value of register *Allocated* is 4, equal to $\text{Size}_{\text{mini-heap}}$. Since 3 blocks have been freed in the mini-heap, one more de-allocation in this mini-heap can make it available again and reset the value to 0.

VI. Evaluation of Hi-DMM

We conduct the evaluation of Hi-DMM using Zynq-7000 SoC XC7Z0-20 with FPGA fabric operating at 100 MHz. All of the four Hi-DMM allocators, implemented with VivadoHLS (v17.2) are packaged as intellectual property cores using Vivado (v17.2). As mentioned in Section IV-B, the HLS accelerator requests the allocation of memory from these allocators via HLS handshake protocol. Each accelerator can be integrated with more than one allocators to adapt to various allocations. All the data of buddy tree are stored in BRAM.

A. Performance and Cost of Hi-DMM Allocators

1) Management Capability and Utilization of Resource: For each type of allocator, the utilizations of LUT and BRAM are presented in Fig. 9(a) and (b). FBTA and PATA have the same utilization of BRAM as they use the same method to store buddy tree. To handle the same number of MAUs, PATA will cost 36% area more than FBTA on average, due to PATA's extra logic for preallocation scheme. Noticeably, the costs of area, for FBTA and PATA, increase significantly when the number of MAUs is more than 256 though they are low-latency allocators. Their utilization of LUT and BRAM exceed 19% for those situations due to the large width of BVs, which might be not acceptable for some applications. That is the major reason why we propose HTA, which uses less resources but still takes much lower latency compared to SysAlloc, by searching address based on bitwise-operation and maintaining

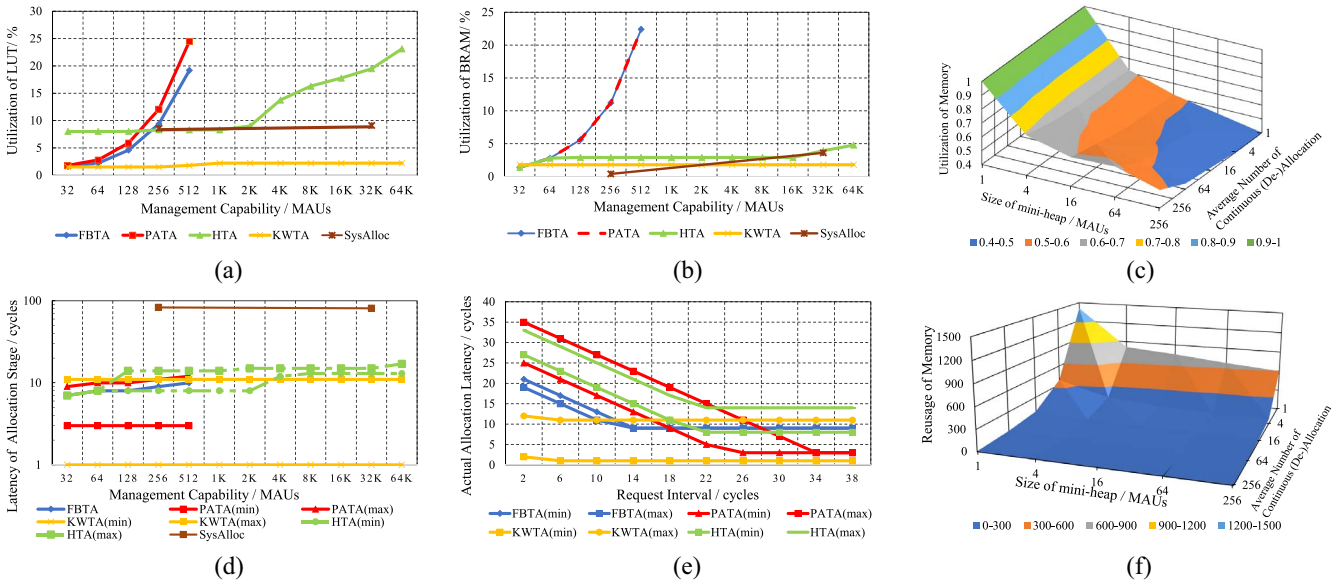


Fig. 9. Evaluation of Hi-DMM allocators compared with SysAlloc. (a) LUT utilization. (b) BRAM utilization. (c) KTWA: utilization of memory. (d) Latency of allocation stage. (e) Actual allocation latency. (f) KTWA: reusage of memory.

the buddy tree with look-up table of mask. For heaps with a large number of MAUs, it is more reasonable to use HTA than FBTA and PATA. As indicated in Fig. 9(a) and (b), KWTA can handle more than 2048 MAUs at very low cost of area. Most of those applications, requiring large management capability of the allocator, usually use DMM to deal with dynamic data structures, e.g., tree and graph. For example, SysAlloc tested its management capability of 32 K MAUs based on an application with dynamic list. The allocations for these data structures, e.g., a node in tree, are fine-grained and fixed-size. Therefore, using allocator based on complex buddy tree for these applications can be a waste of area and, instead, KWTA can perform efficiently in these applications. Even compared to previous fixed-size allocator DOMMU [5], KTWA costs less area while performs lower allocation latency. Moreover, DOMMU increases the BRAM access latency due to address translator but KTWA does not have this limitation.

2) Management Capability and Allocation Latency: For single allocator of the different types, the latencies of allocation stage are shown in Fig. 9(d). Note that for comparison, keeping the same total overhead, we scale the latencies of SysAlloc according to the clock rate from 150 to 100 MHz. As presented, the average latencies of SysAlloc is 83 for 256 MAUs and 81 cycles for 32 K MAUs. For each kind of allocators, the latency of allocation stage of a request only depends on the management capability of allocator and the granularity of request size, since they determine the overhead of searching allocable addresses. We implemented a generator of requests with random sizes. To handle a certain number of MAUs, FBTA has a constant latency of allocation stage while other Hi-DMM allocators have variable latencies of allocation stage. In the case of HTA, searching a node in the trunk part for coarse-grain allocation, the allocation stage takes 7–8 cycles, while for fine-grained allocation which involves the branch part, it takes 12–19 cycles. For PATA, if the request of allocation hits the preallocated block, the latency will be 3 cycles. In case of a miss, the latency is 9–12 cycles. Similarly, when the request to KWTA can be handled by the recorded mini-heap, the latency of allocation stage is 1 cycle instead of 9 cycles to allocate a new mini-heap.

However, the actual allocation latency is not simply equal to the latency of allocation stage which can be treated as a lower bound. In some cases, a request may come to an allocator when the allocator has not finished the maintenance of a previous allocation. As result, the latency of maintenance cannot be completely hidden from requesting accelerator and the requesting accelerator needs to wait for the completion of maintenance. Therefore, the actual allocation latency is also affected by the latencies of maintenance stage, shown in Table II, and the intervals between the response of last request and the following request. By generating requests with random sizes and at different intervals, the actual allocation latencies for different allocators managing 256 MAUs are shown in Fig. 9(e). With the interval increasing, the actual latencies of allocators will decrease until the interval covers the maintenance of allocators. Though PATA has lower allocation latency bound than FBTA, the actual allocation latency of PATA can be greater than that of FBTA due to higher maintenance latency of PATA. Therefore, PATA is suitable to those applications involving similar requests with relatively large intervals.

3) Memory Utilization and Reuse: We also evaluate the memory (BRAM for Xilinx FPGAs) utilization of the proposed allocators.

For FBTA, PATA, and HTA which are based on binary buddy tree, unless requested sizes is in the set of provided sizes, a power of 2, the allocator will allocate more memory than requested. Such a waste of memory is intrafragmentation. The extent of intrafragmentation depends on the set of provided sizes and the distribution of request sizes. According to the calculation in [15], for buddy system, asymptotic value

of intrafragmentation ranges from 25% to 33% of memory, based on a uniform distribution of request size.

However, KTWA is an allocator for fixed-size blocks based on mini-heaps and one-way allocation scheme and has different issue of memory utilization. The experiments for KWTA is conducted with a generator of random requests. For each step of the test, the generator first determines allocation and deallocation with equal possibilities and then generates a number of the same requests continuously. The number of continuous requests for each step is based on a uniform distribution with an average. We assume each application can be represented by a corresponding average. We collect the statistics of utilization and reusage of memory when the first failure of allocation occurs due to all the mini-heaps marked unavailable. The reusage of memory is defined as the ratio of the number of allocation requests to the management capability of KWTA. The related results, based on a KWTA managing 4096 MAUs, are presented in Fig. 9(c) and (f). As demonstrated in the figure, the utilization of memory, slightly impacted by the average number of continuous requests, will decrease as the size of mini-heaps increase, because a freed block will be available after all the blocks in a mini-heaps are freed and a larger mini-heap may take a longer period to be cleared. However, designers may be more concerned about the reusage of memory since it means how many allocation requests can be handled by KTWA before overflow occurs. As presented in the figure, the reusage of memory will decrease if the size of mini-heaps increases, but the average latency of KTWA allocation will be lower with larger mini-heaps, as mentioned in Section V-B4. Moreover, KTWA can achieve high memory reusage ($\geq$50) for those applications with relatively small average number ($\leq$16) of continuous requests, even with the size of mini-heap greater than 4. Therefore, designers need to set an expected reusage of memory and an estimated average number of continuous requests of the application and Hi-DMM will choose the largest size for mini-heaps that meets the requirements to reduce the average latency.

Hi-DMM will assign allocators to heaps according to the curves in Fig. 9 to improve allocation performance for a target FPGA.

B. Impact of Configurations on Application

As mentioned in Section IV, it is necessary for Hi-DMM compiler to optimize the configuration of DMM components under designers' guide for the sake of the performance of accelerators. In this section, the experiments are used to explore how the configuration impacts the overall performance of allocator instead of memory utilization, which is analyzed in Section VI-A. The most significant factors in configurations are the assignment of pointers to heaps, the adaption to HLS directives and the selection of allocator. For each of them, we use a corresponding application, involving common DMM and HLS behaviors, to explain its impact. In the experiments, the applications are the common processes in HLS designs, e.g., matrix multiplication, reduction operation, and interaction of dynamic data structures. The three applications are listed as below and the results are given in Table III, with the overall latencies of the execution of accelerators. We explore the impacts of pointer assignments and loop transformation based on application A and B, respectively. Moreover, for KWTA and HTA, which have similar management capability, we compare their performance based on application C.

TABLE III
IMPACT OF HI-DMM CONFIGURATIONS ON APPLICATION

App	Configuration	Latency / cycles
A	FBTA: Assign all pointers to one heap	133033
	FBTA: Distributes pointers properly	125033
B	FBTA: No transformation of loop	842
	FBTA: Separate the unrolled loop	412-812
C	HTA without pre-allocation scheme	16840
	HTA with pre-allocation scheme	16563
	KWTA	15890

A: Calculation with multiple matrices does the matrix computation $\boldsymbol{ABCD}+\boldsymbol{EF}$. It can show the effect of the assignment of pointers to heaps. The number of heaps is 2. Four dynamically allocated arrays, $\boldsymbol{W}, \boldsymbol{X}, \boldsymbol{Y}$, and $\boldsymbol{Z}$, will be involved to store intermediate results. $\boldsymbol{W}, \boldsymbol{X}$, and $\boldsymbol{Y}$ will co-operate to do a matrix multiplication and then $\boldsymbol{W}$ and $\boldsymbol{Z}$ will be processed for matrix addition. Hi-DMM reduces 6.01% of runtime by assigning $\boldsymbol{W}$ and $\boldsymbol{X}$ to one heap, and $\boldsymbol{Y}$ and $\boldsymbol{Z}$ to the other one.

B: Sum reduction with unrolled loop and partitioned arrays reduces a 2-D array to a dynamically allocated 1-D array with both loop unroll factor and array partition factor being 4. It demonstrates the improvement by adapting DMM to HLS directives. Hi-DMM separates the inner loop of 2-D reduction into two loops according to Section IV so that tools like Vivado HLS can optimize the performance of the unrolled loop. The resultant runtime of application is a range, 412–812 cycles, since it depends on the exact allocated address of corresponding dynamic array, which leads to different latency of the first loop after loop separation.

C: Shortest path faster algorithm is a queue-based improvement of the Bellman–Ford algorithm which computes single-source shortest paths in a weighted directed graph. Hi-DMM will automatically choose KWTA for the dynamic queue, since it can perform allocation with average latency of 1.5 cycles for the dynamic data structure. For comparison, we repeat the experiment but use HTA to handle the queue, which leads to average allocation latency of 12.0 cycles. According to Table III, KWTA can reduce the runtime of application by 5.64%. When KWTA is applied, the allocation latency only takes up 0.84% of runtime but it increases to 6.34% when HTA is applied. If HTA is replaced by SysAlloc allocator, the allocation latency will be more than 40% of runtime. Note that preallocation scheme can help HTA reduce the average allocation latency by 25.5%.

VII. CONCLUSION

This paper presents Hi-DMM, a DMM platform for HLS, covering the analysis and transformation of the HLS source code and providing designers with flexible and high-performance allocators based on optimized buddy tree algorithm. For future work, the management capability of HTA will be extended to more than 128 K MAUs and the impact

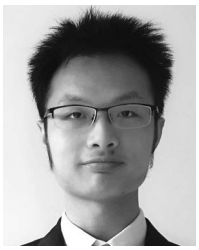
of DMM behaviors should be further analyzed based on HLS to improve the efficiency of accelerator. The source code of Hi-DMM and of the applications used in Section VI-B can be found at http://ece.ust.hk/~eeweiz/tools.html.

ACKNOWLEDGMENT

The authors would like to thank the anonymous reviewers for their valuable comments to improve the quality of this paper.

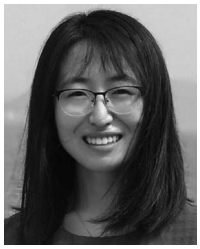
REFERENCES

[1] J. Cong *et al.*, "High-level synthesis for FPGAs: From prototyping to deployment," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 30, no. 4, pp. 473–491, Apr. 2011.
[2] R. Jones and R. Lins, *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*, vol. 208. Chichester, U.K.: Wiley, 1996.
[3] J. Bonwick *et al.*, "The slab allocator: An object-caching kernel memory allocator," in *Proc. USENIX Summer*, vol. 16. Boston, MA, USA, 1994, p. 6.
[4] J. M. Chang and E. F. Gehringer, "A high performance memory allocator for object-oriented systems," *IEEE Trans. Comput.*, vol. 45, no. 3, pp. 357–366, Mar. 1996.
[5] G. Dessouky *et al.*, "Adaptive dynamic on-chip memory management for FPGA-based reconfigurable architectures," in *Proc. FPL*, Munich, Germany, Sep. 2014, pp. 1–8.
[6] D. Diamantopoulos, S. Xydis, K. Siozios, and D. Soudris, "Dynamic memory management in Vivado-HLS for scalable many-accelerator architectures," in *Applied Reconfigurable Computing*. Cham, Switzerland: Springer, 2015, pp. 117–128.
[7] C. Özer, "A dynamic memory manager for FPGA applications," Ph.D. dissertation, Elect. Electron. Eng., Middle East Tech. Univ., Ankara, Turkey, 2014.
[8] Z. Xue and D. B. Thomas, "SysAlloc: A hardware manager for dynamic memory allocation in heterogeneous systems," in *Proc. FPL*, London, U.K., Sep. 2015, pp. 1–7.
[9] S. K. Agun and M. Chang, "Reconfigurable fast memory management system design for application specific processors," in *Proc. IEEE VLSI*, Tampa, FL, USA, 2003, pp. 92–97.
[10] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *Proc. CGO*, San Jose, CA, USA, 2004, p. 75.
[11] G. Corre, E. Senn, P. Bornel, N. Julien, and E. Martin, "Memory accesses management during high level synthesis," in *Proc. IEEE CODES+ISSS*, Stockholm, Sweden, 2004, pp. 42–47.
[12] D. R. Karger, "Global min-cuts in RNC, and other ramifications of a simple min-cut algorithm," in *Proc. SODA*, vol. 93. Austin, TX, USA, 1993, pp. 21–30.
[13] A. Cilardo and L. Gallo, "Interplay of loop unrolling and multidimensional memory partitioning in HLS," in *Proc. DATE*, Grenoble, France, 2015, pp. 163–168.
[14] GNU. (2018). *C Library*. [Online]. Available: http://ftp.gnu.org/gnu/glibc/glibc-2.27.tar.xz
[15] J. L. Peterson *et al.*, "Buddy systems," *Commun. ACM*, vol. 20, no. 6, pp. 421–431, 1977.



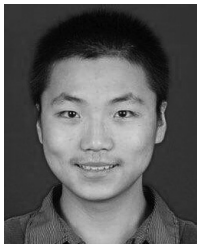
Tingyuan Liang (S'18) received the B.S. degree in electrical and information engineering from Zhejiang University, Hangzhou, China, in 2017. He is currently pursuing the M.Phil. degree with the Department of Electronic and Computer Engineering, Hong Kong University of Science and Technology, Hong Kong.

His current research interests include high level synthesis, computer architecture, and reconfigurable computing.



Jieru Zhao (S'18) received the B.S. degree in microelectronics from Nanjing University, Nanjing, China, in 2015. She is currently pursuing the Ph.D. degree with the Department of Electronic and Computer Engineering, Hong Kong University of Science and Technology, Hong Kong.

Her current research interests include high level synthesis, reconfigurable computing, and electronic design automation.



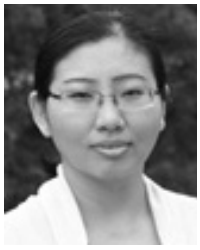
Liang Feng (S'16) received the B.S. degree in microelectronics from Nanjing University, Nanjing, China, in 2014. He is currently pursuing the Ph.D. degree with the Department of Electronic and Computer Engineering, Hong Kong University of Science and Technology, Hong Kong.

His current research interests include reconfigurable computing, multicore system, and electronic design automation.

Sharad Sinha (S'03–M'14) received the B.Tech. degree in electronics and communication engineering from the Cochin University of Science and Technology, Kochi, India, in 2007.

He was a Research Scientist with Nanyang Technological University, Singapore. He is an Assistant Professor with the Department of Computer Science and Engineering, Indian Institute of Technology Goa, Goa, India. From 2007 to 2009, he was a Design Engineer with Processor Systems (India) Pvt. Ltd., Bengaluru, India. His current research and teaching interests include computer architecture, embedded systems, and reconfigurable computing.

Mr. Sinha was a recipient of the Best Paper Award at ICCAD 2017 and the Best Speaker Award from IEEE CASS Society, Singapore Chapter, in 2013 for his Ph.D. research on High Level Synthesis. He serves as an Associate Editor for IEEE Potentials and a Senior Editor for ACM Ubiquity.



Wei Zhang (M'05) received the B.S. and M.S. degrees from the Harbin Institute of Technology, Harbin, China, in 1999 and 2001, respectively, and the Ph.D. degree from Princeton University, Princeton, NJ, USA, in 2009.

She is currently an Associate Professor with the Department of Electronic and Computer Engineering, Hong Kong University of Science and Technology, Hong Kong, where she established the Reconfigurable System Laboratory. She was an Assistant Professor with the School of Computer Engineering, Nanyang Technological University, Singapore, from 2010 to 2013. She has authored over 80 technical papers in referred international journals and conferences, and three book chapters. Her current research interests include reconfigurable system, power and thermal management, embedded system security, and emerging technologies.

Dr. Zhang currently serves as an Associate Editor for the IEEE TRANSACTIONS ON VERY LARGE SCALE INTEGRATION SYSTEMS (VLSI), the IEEE TRANSACTIONS ON COMPUTER-AIDED DESIGN OF INTEGRATED CIRCUITS AND SYSTEMS, the *ACM Transactions on Embedded Computing Systems*, and the *ACM Journal on Emerging Technologies in Computing Systems*. She serves as the General Chair of ISVLSI 2018 and also serves on many technical program committees, including DAC, ICCAD, ASPDAC, CASES, and FPL.