# High-Performance and Resource-Efficient Dynamic Memory Management in High-Level Synthesis

Qinggang Wang[†‡], Long Zheng[†], Zhaozeng An[†], Haoqin Huang[†], Haoran Zhu[†], Yu Huang[†‡], Pengcheng Yao[†‡], Xiaofei Liao[†], and Hai Jin[†]

[†]National Engineering Research Center for Big Data Technology and System/Service Computing Technology and System Lab/Cluster and Grid Computing Lab, Huazhong University of Science and Technology, Wuhan, China

[‡]Zhejiang Lab, Hangzhou, China

{qgwang,longzh,anzz,hqhuang,hr_zhu,yuh,pcyao,xfliao,hjin}@hust.edu.cn

## Abstract

With the merits of high productivity and ease of use, *high-level synthesis* (HLS) tools bring hope to fast FPGA-based architecture development. However, their usability and popularity are still limited due to lack of support for *dynamic memory management* (DMM). Though HLS-compatible DMM solutions have been proposed recently, nevertheless, based on our investigation, none of them can hit *high performance* (i.e., minimal memory (de-)allocation latency) and *resource efficiency* (i.e., managing arbitrarily sized memory with minimal FPGA resource consumption) with one stone, seriously limiting their practicality. In response, we propose HeroDMM, a high-performance and resource-efficient dynamic memory manager for HLS. Specifically, HeroDMM organizes the managed memory area with a novel *cartesian-like tree* (CT) structure, a key to resolving the dilemma between (de-)allocation latency and resource efficiency standing in front of prior efforts. With the CT structure, HeroDMM further devises a delicate memory management algorithm and specializes the hardware implementation for achieving ever-higher performance while ensuring resource efficiency. Results show that HeroDMM outperforms state-of-the-art HLS-compatible DMM solutions by 61.69%~99.99% in performance improvement and 23.79%~97.22% in resource consumption savings.

This work is supported by the National Key Research and Development Program of China under Grant No.2023YFB4503400. This work is also funded by China Postdoctoral Science Foundation (No.2023TQ0328, 2023M743256) and Zhejiang Provincial Natural Science Foundation of China (No.LQ24F020027). The corresponding author is Long Zheng.



*DAC'24, June 23–27, 2024, San Francisco, CA, USA*



ACM ISBN 979-8-4007-0601-1/24/06…$15.00
https://doi.org/10.1145/3649329.3655945

## 1 Introduction

*High-level synthesis* (HLS) tools, which offer FPGA design the potentials of high productivity and low barrier of entry [2], have been adopted in numerous application developments, including graph processing [1] and machine learning [9]. However, most existing commercial and research HLS tools, such as Vivado HLS [12], lack support for *dynamic memory management* (DMM), responsible for performing memory (de-)allocations dynamically at runtime [4, 6, 7, 14].

DMM is an indispensable functional component to optimize memory usage for real-world applications [9, 11]. The absence of DMM restricts HLS users to either perform manual *register transfer level* (RTL) code refactoring or employ static memory allocation. The former requires developers with specialized hardware knowledge to heavily implement DMM functionality, making it challenging for software engineers [7]. The latter necessitates the upfront identification of varying sizes for distinct allocation requests, which is prone to overestimating memory requirements, resulting in the underutilization of valuable memory resources [5]. This incentivizes research efforts towards HLS-compatible DMM.

In practice, latency-sensitive applications and resource-constrained FPGAs impose two requirements for the DMM within HLS. First, DMM expects ***high performance***. The average response latency of dynamic memory (de-)allocation requests is a key performance indicator. Long latency forces *processing elements* (PE) to be stalled to wait for the memory response, which may hinder overall application acceleration efficiency [8]. Thus, we expect to respond to dynamic (de-)allocation requests as fast as possible. Second, DMM should achieve ***resource efficiency*** in the sense that arbitrarily sized memory can be managed with minimal FPGA resource consumption. In particular, the LUT and BRAM resources of a single FPGA are scarce. Thus, DMM should consume as few resources as possible to manage larger on-/off-chip memory, which has increased from MB-scale to GB-scale [13].

There has emerged a series of efforts on HLS-compatible DMM [3–6, 8, 13, 14], delivering substantial gains in either (de-)allocation performance or resource efficiency. Nevertheless, none of these prior efforts can hit high performance and resource efficiency with one stone. In view of this, we conduct a set of experiments to evaluate the representative

DMM solutions in terms of performance (measured by the response latency) and resource efficiency (measured by LUTs and BRAMs utilization). The results (presented in §2.2) reveal that existing DMM solutions for HLS might encounter the dilemma of simultaneously achieving low (de-)allocation latency and mitigating the managed memory scaling-incurred resource consumption expansion. Thus, this paper focuses on developing a "one-stone-two-birds" solution, in which performance and resource efficiency for HLS-compatible DMM can be enhanced simultaneously.

In this paper, we present HeroDMM, a high-performance and resource-efficient dynamic memory manager for HLS. First, HeroDMM introduces the novel *Cartesian-like Tree* (CT) structure to track the managed memory usage, and devises an elegant memory management algorithm to turn (de-)allocation request handling into search operations on the CT. On this basis, HeroDMM further specializes the CT-friendly hardware implementation, which not only enables low-latency search to significantly improve (de-)allocation performance but even suppresses resource utilization expansion caused by managed memory scaling for resource efficiency. Finally, HeroDMM presents high-level interfaces that can be synthesized with Vivado HLS to facilitate users.

This paper presents the following main contributions:

- We investigate performance and resource efficiency issues arising in prior HLS-compatible DMM solutions.
- We present HeroDMM, a practicable HLS-compatible dynamic memory manager, to break the dilemma between (de-)allocation latency and resource efficiency, which is unique to the best of our knowledge.
- We evaluate HeroDMM against three state-of-the-art HLS-compatible DMM solutions, achieving significant performance improvement (61.69%∼99.99%) and resource consumption savings (23.79%∼97.22% ).

# 2 Background and Motivation

## 2.1 Background

***Dynamic Memory Management* (DMM).** DMM is responsible for allocating a free area or freeing an occupied area in a dynamic fashion so that the same memory resources can be shared and reused at runtime. The managed memory area used for serving these dynamic requests is referred to as the *heap*. A heap comprises *minimum allocable units* (MAUs), whose states (free or not) are usually recorded in the form of a linked list [5], bitmap [3], or buddy tree [8]. Some contiguous free MAUs form a free memory block. An allocation request is associated with the requested memory size, and its response provides the starting address of the memory block that satisfies the requested size. In the case of a deallocation, the starting address of the to-be-released memory block must be given.

***High-Level Synthesis* (HLS).** To simplify the prototyping process of an FPGA-based accelerator with respect to development time and complexity, HLS tools [12] are proposed to take algorithmic descriptions written in C/C++ as input and automatically produce RTL specifications. However, the well-known DMM functions `malloc()` and `free()` are not supported by existing HLS tools [2, 8]. In HLS, developers usually pre-allocate a certain amount of memory through a static memory allocation approach to meet all runtime memory requirements of an HLS kernel. Nonetheless, dynamic memory requirement is unclear in advance for numerous applications, leading to an overestimation of memory needs or kernel termination due to insufficient memory [7]. While it is possible to count an accurate pre-allocation size by pre-running the task, this inevitably incurs unnecessary overhead. Thus, a major challenge in HLS is to enable effective and synthesizable `malloc()` and `free()` functions.

**HLS-compatible DMM Efforts.** Existing efforts generally fall into three series. First, the bitmap-based DMM series represented by DMM-HLS [3] features a *bitmap-based structure* to enable a *linear search* with the time complexity of O($N$). Second, the buddy tree-based DMM series represented by SysAlloc [13] inherits the advantage of *binary search* from the *buddy tree structure*, reducing the time complexity to O(log($N$)) yet consuming more memory resources for caching the tree structure. Hi-DMM [8] further turns binary search into bitwise operations more suited for FPGAs, improving performance at the expense of LUT resources. Third, the linked list-based DMM series represented by gnumem [5] tracks free blocks with a *linked list structure* to save BRAMs, suffering from high latency caused by the *linear search*.

## 2.2 Motivation

**Limitation Analysis.** To investigate both (de-)allocation performance and resource efficiency of the existing three series of HLS-compatible DMM solutions, we take the three contemporary efforts, namely DMM-HLS [3], gnumem [5], and Hi-DMM [8], as representatives to evaluate their average (de-)allocation latency and utilization of LUTs and BRAMs. Figure 1 depicts the results by benchmarking random memory requests [6] on heap sizes ranging from 256K MAUs to 8M MAUs. More experiment details can be found in §4.1.

Overall, we can see that the existing HLS-compatible DMM solutions might encounter the dilemma of achieving both high performance and resource efficiency simultaneously. Owing to fast binary search on the buddy tree and FPGA-friendly bitwise arithmetic operations, Hi-DMM outperforms DMM-HLS and gnumem by 359.75× and 27.21× in terms of (de-)allocation latency, on average. Nevertheless, since Hi-DMM naturally entails more BRAM resources for caching the auxiliary tree structure and LUT resources for performing bitwise operations, its resource efficiency can be affected more considerably. In contrast, due to tracking only free memory blocks instead of all MAUs, gnumem greatly alleviates the BRAM consumption than Hi-DMM, with an average reduction of 15.33×. In particular, since DMM-HLS leverages LUTs to implement a distributed RAM structure for storing the number of positions of the memory objects, its LUT usage increases dramatically as the number of MAUs

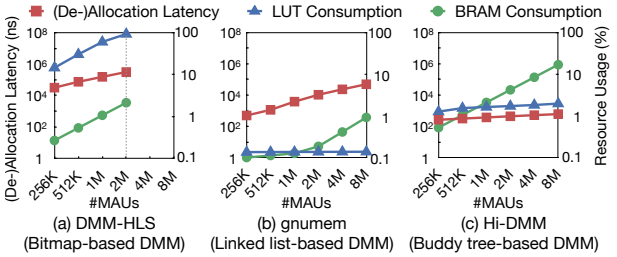



**Figure 1.** Comparing DMM-HLS [3], gnumem [5], and Hi-DMM [8] in terms of performance and resource efficiency

increases, reaching 92.89% when the number of MAUs is 2M. In summary, we have the following observation:

**Observation:** *Buddy tree-based DMM schemes deliver high performance yet entail more resource consumption, while linked list-based DMM schemes hold the advantage in resource efficiency yet suffer from high (de-)allocation latency.*

Thus, a seemingly feasible intuition for realizing high-performance and resource-efficient DMM is to combine the respective strengths of buddy tree-based and linked list-based DMM schemes. Specifically, inspired by linked list-based DMM schemes, we suppress the heap scaling-incurred resource utilization expansion by tracking free memory blocks instead of all MAUs. Moreover, we learn from buddy tree-based DMM schemes to achieve better performance in two ways: 1) we organize the free memory blocks with a tree structure so as to leverage the binary search to reduce the time complexity; 2) we specialize FPGA-friendly hardware implementation to enable the low-latency search.

**Challenges.** Materializing the above idea presents several challenges. First, each free memory block contains two critical parameters, namely starting address and memory size. It is difficult to link together free memory blocks in a tree, ordered by both address and size, to simultaneously facilitate the address search for deallocation and the desired size search for allocation. Second, during memory (de-)allocation, dynamically modifying these two parameters incurs non-negligible maintenance overheads, which must be carefully controlled. Finally, caching all tree structure on-chip can greatly alleviate the latency increase caused by random accesses yet entail more BRAM resources. Balancing between latency reduction and BRAM savings remains challenging.

# 3 HeroDMM Architecture

## 3.1 Cartesian-like Tree Structure

To combine the respective strengths of buddy tree-based and linked list-based DMM schemes, HeroDMM organizes free memory blocks into a novel *cartesian-like tree* (CT) [10]. Figure 2(a) depicts an example heap with a capacity of 160 bytes, where each colored (uncolored) square indicates a free (allocated) MAU of size 4 bytes. There are nine free memory blocks, each consisting of contiguous MAUs with the same color. Figure 2(b) displays the CT for tracking heap usage with each node in the tree mapping to a unique free block. Each node has a pair of attributes $< addr, size >$

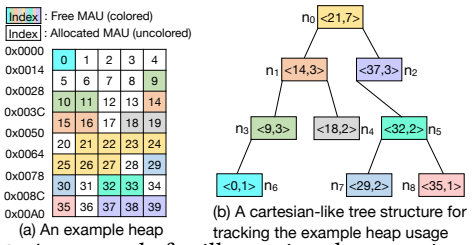



**Figure 2.** An example for illustrating the cartesian-like tree

(e.g., $< 21, 7 >$), where *size* is the number of MAUs in the represented free block, and *addr* indicates the index of the first MAU. Each node $n_i$ satisfies two rules: 1) the *addr* of $n_i$ is larger than that of any node in the left subtree of $n_i$ and less than that of any node in the right subtree of $n_i$; 2) the *size* of $n_i$ is no less than that of any node in $n_i$'s subtrees.

Consider the node $n_0$ in Figure 2(b) as an example. The *addr* of $n_0$ (i.e., 21) is larger than that of its left child node $n_1$ (i.e., 14) and less than that of its right child node $n_2$ (i.e., 37). Moreover, the *size* of $n_0$ (i.e., 7) is greater than that of its two child nodes (i.e., 3 and 3).

## 3.2 CT-based (De-)Allocation Algorithm

Whether handling a memory allocation or deallocation request, HeroDMM relies on two phases: *search* and *update*.

**Allocation.** For allocation, the *search* phase aims to locate a node in the CT that can accommodate the requested allocation size. Subsequently, the *update* phase first modifies *size* and *addr* of the found node to signify that the requested memory area has been occupied, and then moves this node downwards in the CT until its modified *size* is greater than or equal to that of any node in its subtrees.

Algorithm 1 shows the procedure of CT-based memory allocation. In the *search* phase, the root node of the CT is set as an initial $n$ (Line 1). If the *size* of $n$ is larger than the requested allocation capacity $c$ (Line 2), its child node with a smaller *size* that is larger than $c$ will be selected as a new $n$ (Lines 3-6). Then, a new round of search starts and repeats until $n$ has no child nodes that can accommodate the requested $c$ (Line 7). Eventually, the *addr* of the current $n$ will be returned (Lines 8 and 17).

In the *update* phase, the *addr* and *size* of $n$ are first modified to signify that certain MAUs have been occupied (Line 8). Then, the node $n$ is moved down in the CT until its *size* is no less than that of any node in its subtrees. Specifically, the child node (i.e., $l$ and $r$ in Line 10) of $n$ with a larger size is first chosen (Lines 11 and 14), and the parent node of $n$ is denoted by $p$ (Lines 10). Subsequently, when the *addr* of $p$ is greater (less) than that of $n$, the chosen node $l$ or $r$ becomes $p$'s new left (right) child node (Lines 12 and 15). Meanwhile, the right (left) child node of $l$ ($r$) becomes $n$'s new left (right) child node, and $n$ is turned into $l$'s new right child node ($r$'s new left child node) (Lines 13 and 16).

*Allocation Example.* Let us consider the CT shown in Figure 2(b) when an allocation request for 3 MAUs is received. As the initial $n$ (i.e., $n_0$) has a larger size than 3, we designate

**Algorithm 1:** CT-Based Memory Allocation

**Input:** (1) *root* as the root node of the cartesian-like tree, and (2) *c* for representing the memory size requested for allocation
**Output:** *address* – the starting address of the memory block that satisfies the request.

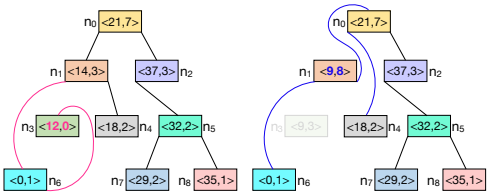
```
1  n ← root /* Search Phase */
2  while c <= n.size do
3      if c ≤ Left (n).size & c ≤ Right (n).size then
4          n ← The child node of n with a smaller size
5      else if c ≤ Left (n).size || c ≤ Right (n).size then
6          n ← The child node of n with a larger size
7      else break;
8  address = n.addr    n.addr += c;    n.size -= c /* Update Phase */
9  while Left (n).size > n.size || Right (n).size > n.size do
10     l ← Left(n);    r ← Right(n);    p ← Parent (n)
11     if l.size > r.size then
12         (p.addr > n.addr) ? Left(p) : Right(p) ← l
13         Left (n) ← Right (l);    Right (l) ← n;
14     else
15         (p.addr > n.addr) ? Left (p) : Right (p) ← r
16         Right (n) ← Left (r);    Left (r) ← n;
17 return address
```

its child node $n_1$ with a smaller *size* that is larger than 3 as a new $n$. The above process is repeated until we find $n_3$, which has no child node that can accommodate the requested 3 MAUs. Then, as shown in Figure 3(a), the *addr* of $n_3$ is updated to 12 (i.e., 9+3) and its *size* becomes 0 (i.e., 3-3), which is smaller than the *size* of $n_6$ (i.e., the original left child node of $n_3$ in Figure 2(b)). Thus, we move $n_3$ down to turn it into a new right child node of $n_6$, and $n_6$ becomes the new left child node of $n_1$ (i.e., the original parent node of $n_3$ in Figure 2(b)).

**Deallocation.** For deallocation, the search phase tries to find nodes that represented free memory blocks are adjacent to the memory area to be released. Then, the *update* phase coalesces these adjacent free memory blocks into a larger one, and moves the node with an increased *size* up in the CT until its *size* is no larger than the *size* of its parent node.

Algorithm 2 presents the procedure for our CT-based memory deallocation. When a deallocation request associated with *address* and *c* arrives, the *search* phase begins. First, we initialize *pre* and *suc* to represent the predecessor and successor memory blocks adjacent to the memory region to be released, respectively (Line 1). Second, the initial *root* (i.e., the root of the CT) is checked to see whether the memory block it represents is adjacent to the to-be-released memory region. If so, we have found *pre* or *suc* (Lines 3 and 4). When *address* is less (greater) than the *addr* of *root*, the left (right) child node of *root* is then selected as a new *root* (Line 5). Subsequently, a new round of search starts and repeats until both *pre* and *suc* are found or *root* is null (Line 2).

In the *update* phase, whenever both *pre* and *suc* are found (Line 6), the one with a smaller (larger) *size* between them is chosen as a to-be-deleted node $n_d$ (a to-be-upward node $n_{up}$) (Lines 7 and 9). Then, we remove $n_d$ from the CT by redirecting the pointer to $n_d$ to its unique child node (Line 8). Meanwhile, the *addr* and *size* of $n_{up}$ are updated to signify




(a) The CT snapshot after the memory request **Malloc**(3 MAUs) is completed (b) The CT snapshot after the memory request **Free**(12, 2) is completed

**Figure 3.** Evolution of HeroDMM's CT from the initial CT depicted in Figure 2(b) to the state after the memory request illustrated is completed

that three adjacent free memory blocks have been coalesced into a larger one (Line 10). If only one between *pre* and *suc* has been found, the found one will be chosen as $n_{up}$ (Lines 11-13). If none of them has been found, we create a new node $n_{up}$ to represent the newly released memory region, and set $n_{up}$ as a leaf node of the CT (Lines 14-16). Finally, we move $n_{up}$ up in the CT, level by level, until the *size* of $n_{up}$ is no larger than that of the parent node of $n_{up}$ (Lines 17-21).

*Deallocation Example.* Consider the heap snapshot shown in Figure 2(a), where the thirteenth and fourteenth MAUs (i.e., *address* = 12 and $c = 2$ ) need to be freed. As the to-be-released memory area is not adjacent to the memory block represented by the initial *root* (i.e., $n_0$) in Figure 2(b) and the *addr* (i.e., 21) of $n_0$ is larger than *address* (i.e., 12), the left child node $n_1$ of $n_0$ is chosen as a new *root*. The above process repeats until both *pre* (i.e., $n_3$) and *suc* (i.e., $n_1$) have been found. As shown in Figure 3(b), since $n_3$ is selected as a to-be-deleted node, its unique child node $n_6$ becomes a new left child of $n_1$. Meanwhile, *addr* and *size* of $n_1$ are updated to 9 and 8, respectively. In this context, the parent node (i.e., $n_0$) of $n_1$ has a smaller *size* than $n_1$. Thus, we move $n_1$ up to turn it into the new root of the CT. Subsequently, the right child node (i.e., $n_4$) of $n_1$ becomes $n_0$'s left child node, and $n_0$ is turned into the new right child node of $n_1$.

### 3.3 Hardware Implementation

Figure 4 depicts the architecture of HeroDMM.

**Allocator.** Initially, the *Set_N* module sets the root node of the CT as $n$ and determines whether the *size* of $n$ is larger than the requested allocation capacity $c$. If not, $n$ is sent to the *update* phase and the *search* phase is completed (④). If so, the *Get_Left* and *Get_Right* modules fetch the left and right child nodes of $n$, respectively (①). Then, HeroDMM determines whether there is a child node of $n$ whose *size* is larger than or equal to $c$ (②). If so, the child node with a smaller *size* that is sufficient to meet the requested allocation capacity is selected as a new $n$ (③), and a new round of search begins.

Whenever the *update* phase receives $n$, the *Upd_Params* module first modifies the *size* and *addr* of $n$ to signify that certain memory areas have been occupied and then awakens *Get_Left* and *Get_Right* to fetch child nodes (❶). Subsequently, HeroDMM determines whether there is a child node of $n$ whose *size* is larger than that of $n$ (❷). If not, the *update* phase is completed (❺). If so, the *Get_Parent* module fetches the parent node (i.e., $p$) of $n$ and triggers *Get_Left* and

**Algorithm 2:** CT-Based Memory Deallocation

**Input:** (1) *root* as the root node of the cartesian-like tree, and (2) *address* and *c* for representing the starting address and the memory size requested for deallocation, respectively

```
1  pre ← null;   suc ← null /* Search Phase */
2  while root != null & (pre == null||suc == null) do
3      if root.addr + root.size == address then pre ← root ;
4      if address + c == root.addr then suc ← root;
5      root ← address< root.addr ?Left(root):Right(root)
6  if pre!=null & suc != null then
7      n_d ← pre.size<suc.size ? pre : suc/* Update Phase */
8      The pointer to n_d is redirected to the only child node of n_d
9      n_up ← pre.size>suc.size ? pre : suc
10     n_up.addr = pre.addr;   n_up.size = pre.size + suc.size + c
11 else if pre!=null then  pre.size+=c;   n_up ← pre ;
12 else if suc!=null then
13     suc.addr=address;   suc.size += c;   n_up ← suc
14 else
15     n_up ← CreateNewNode (address, c)
16     Parent (root).AddChildNode (n_up)
17 p ← Parent(n_up)
18 while p.size < n_up.size do
19     Parent(p).addr > n_up.addr ? Left (Parent(p)) : Right
         (Parent(p)) ← n_up
20     Right(p) ← Left(n_up);   Left(n_up) ← p;
21     p ← Parent(n_up)
```

*Get_Right* to obtain the $p$'s child nodes, where the left (right) child node is sent to the *Redirect* module if the *addr* of $p$ is greater (less) than that of $n$ (③). Meanwhile, the child node of $n$ with a larger size is chosen and becomes both the new child node of $p$ and the new parent node of $n$ (④). In this way, $n$ is moved down one level in the CT, and the process is repeated until the termination condition (i.e., ❺) is triggered.

**Deallocator.** When a deallocation request associated with the starting *address* and memory size $c$ is received, the *Init* module sets the root node of the CT as an initial *root*. Then, HeroDMM determines whether the free memory block represented by *root* adjoins the memory region to be released (① and ②). If so, *root* is recorded as *pre* or *suc* (③). If not, the left (right) child node of the current *root* is selected as a new *root* when the *addr* of the current *root* is greater (less) than *address* (④). HeroDMM repeats the above procedure until both *pre* and *suc* have been found or *root* is empty (⑤).

Once both *pre* and *suc* have been found, HeroDMM coalesces the free memory blocks represented by them and the to-be-released memory area into a larger free memory block. Specifically, the one with a smaller *size* between *pre* and *suc* is deleted from the CT by redirecting the pointer to it to its unique child node (❶), and another is tagged as the to-be-upward node $n_{up}$ (❷). If only one between *pre* and *suc* has been found, the found one is marked as $n_{up}$ (❸). When none of them has been found, the *Crt_Node* module is triggered to create a new node $n_{up}$ associated with *address* and $c$, which is then set as a leaf node of the CT by the *Get_Par* and *Add_Child* modules (④). Finally, the *Move_Node_Upwards* module moves the node $n_{up}$ up in the CT, level by level, until

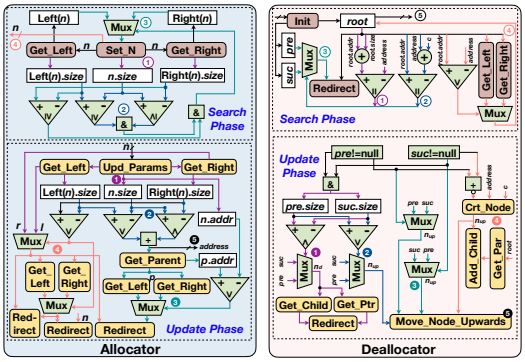



**Figure 4.** The HeroDMM architecture

the *size* of $n_{up}$ is no larger than that of the current parent node of $n_{up}$ (❺). Note that since an upward movement of $n_{up}$ can be formulated into a downward movement of its parent node that has been discussed in our allocator, the implementation details of *Move_Node_Upwards* are omitted.

**Programming Interfaces.** We have implemented and encapsulated HeroDMM as a library, which provides three HLS-compatible programming interfaces to facilitate the users to explore the library. In detail, `HeroDMM_Init` is used to generate the cartesian-like tree structure based on the given heap. `HeroDMM_Malloc` and `HeroDMM_Free` are used to replace traditional `malloc` and `free` functions, respectively.

# 4 Evaluation

## 4.1 Experimental Setup

**HeroDMM Settings.** We implement HeroDMM on a Xilinx Alveo U250 FPGA accelerator card, which provides 11.81MB on-chip BRAMs, 1.68M LUTs, 3.37M Registers, and four 16GB DDR4. Resource utilization and clock rate are obtained using Xilinx Vitis 2020.2, with HeroDMM operating at 250MHz.

**Baselines.** We compare HeroDMM with three representative HLS-compatible DMM solutions, DMM-HLS [3], gnumem [5], and Hi-DMM [8]. In order to achieve a fair and meaningful comparison, we use the same parameter configurations as described in their papers and evaluate them on the same FPGA platform as HeroDMM.

**Benchmarks.** To investigate both (de-)allocation performance and resource efficiency, we use the publicly available benchmark suite *dmbenchhls* [5], which can provide a series of random memory (de-)allocation requests with varying allocation sizes. In the tests, the size of a single MAU is configured as 32B, and the number of MAUs within the heap ranges from 256K to 8M. Besides, we also evaluate the ratio of fragmentation-induced *memory allocation failures* (MAFs).

## 4.2 Overall Results

**(De-)Allocation Performance.** Figure 5 shows the (de-)allocation performance of HeroDMM against DMM-HLS, gnumem, and Hi-DMM with a varying number of MAUs.

*HeroDMM vs. DMM-HLS.* Overall, HeroDMM outperforms DMM-HLS by lowering (de-)allocation latency by 99.76% ∼ 99.99% (99.93% on average). The reason is simple. DMM-HLS faces the time complexity of O($N$) caused by the linear search,

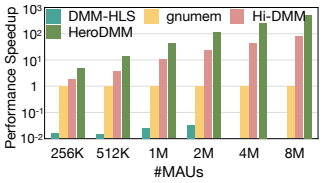



**Figure 5.** Performance comparison of HeroDMM with DMM-HLS [3], gnumem [5], and Hi-DMM [8]

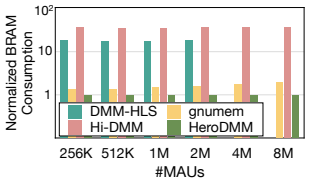



**Figure 6.** BRAMs usage comparison of HeroDMM with DMM-HLS [3], gnumem [5], and Hi-DMM [8]

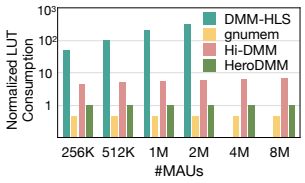



**Figure 7.** LUTs usage comparison of HeroDMM with DMM-HLS [3], gnumem [5], and Hi-DMM [8]

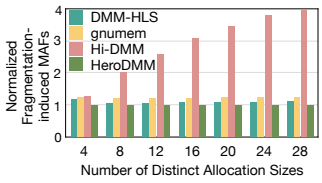



**Figure 8.** The fragmentation-induced MAF rates across varying numbers of distinct allocation sizes

whereas HeroDMM inherits the advantage of fast binary search from buddy tree-based DMM schemes, reducing (de-)allocation latency significantly.

*HeroDMM vs. gnumem.* While gnumem only needs to track free memory blocks rather than all MAUs, its latency is still limited by the inefficient linear search. With the support of our CT structure, HeroDMM reverses this situation, achieving substantial latency reduction. Thus, HeroDMM's performance surpasses that of gnumem by 6.65×~508.56×.

*HeroDMM vs. Hi-DMM.* As disclosed in Figure 1(c), Hi-DMM suffers from a linear growth in hardware resource requirements for its bitwise computation logic with respect to the number of MAUs, leading to a significant reduction in operational frequency. In contrast, HeroDMM achieves a high operating frequency of 250MHz, owing to its simplified hardware implementation. Consequently, HeroDMM outperforms Hi-DMM by 2.61×~6.38× (4.85× on average).

**Resource Consumption.** Figure 6 counts the resource utilization of BRAMs. Compared to DMM-HLS, gnumem, and Hi-DMM, HeroDMM diminishes BRAM usage by 17.78×, 1.57×, and 35.56×, on average. Benefits of HeroDMM stem from tracking free memory blocks instead of all MAUs, as explained in §2.2. In particular, while gnumem also tracks only free memory blocks, it employs a doubly linked list to organize free blocks, leading to more BRAMs consumption.

Figure 7 shows the LUT resource consumption. Compared with DMM-HLS and Hi-DMM, HeroDMM achieves LUT savings of 169.86× and 5.73× on average, owing to its simplified hardware implementation. In particular, HeroDMM consumes more LUTs compared to gnumen, with up to 2.11× higher LUT costs for its node movement within the CT.

### 4.3 Memory Fragmentation Analysis

Figure 8 characterizes the fragmentation-induced MAFs rates of HeroDMM against DMM-HLS, gnumem, and Hi-DMM across varying numbers of distinct allocation sizes. All results are normalized to HeroDMM. Compared to DMM-HLS, gnumem, and Hi-DMM, HeroDMM reduces the number of MAFs by 1.09×, 1.23×, and 2.89× on average, respectively. The reasons are twofold. First, our CT-based allocation algorithm follows the best-fit policy, which tries to find the smallest memory block that satisfies the allocation request, to minimize the internal fragmentation. Second, our CT-based deallocation algorithm can coalesce contiguous free memory blocks into a larger one, mitigating memory fragmentation.

## 5 Conclusion

This paper presents HeroDMM, a high-performance and resource-efficient dynamic memory manager, designed for HLS-based FPGAs. HeroDMM introduces a *cartesian-like tree* (CT) structure to record the managed memory area usage for suppressing the resource expansion caused by the managed memory scaling. With the CT structure, a delicate memory management algorithm and specialized hardware implementation are employed for improving performance. Results demonstrates that HeroDMM outperforms existing HLS-compatible DMM solutions significantly.

## References

[1] X. Chen, Y. Chen, F. Cheng, H. Tan, B. He, and W. Wong. 2022. ReGraph: Scaling Graph Processing on HBM-enabled FPGAs with Heterogeneous Pipelines. In *Proceedings of MICRO*. 1342–1358.

[2] J. Cong, J. Lau, G. Liu, S. Neuendorffer, P. Pan, K. Vissers, and Z. Zhang. 2022. FPGA HLS Today: Successes, Challenges, and Opportunities. *ACM TRETS* (2022), 51:1–51:42.

[3] D. Diamantopoulos, S. Xydis, K. Siozios, and D. Soudris. 2015. Dynamic Memory Management in Vivado-HLS for Scalable Many-Accelerator Architectures. In *Proceedings of ARC*. 117–128.

[4] N. Giamblanco and J. Anderson. 2019. ASAP: Automatic Sizing and Partitioning for Dynamic Memory Heaps in High-Level Synthesis. In *Proceedings of FPT*. 275–278.

[5] N. Giamblanco and J. Anderson. 2019. A Dynamic Memory Allocation Library for High-Level Synthesis. In *Proceedings of FPL*. 314–320.

[6] A. Kokkinis, D. Diamantopoulos, and K. Siozios. 2022. Dynamic Heap Management in High-Level Synthesis for Many-Accelerator Architectures. In *Proceedings of FPL*. 287–293.

[7] J. Lau, A. Sivaraman, Q. Zhang, M. Gulzar, J. Cong, and M. Kim. 2020. HeteroRefactor: refactoring for heterogeneous computing with FPGA. In *Proceedings of ICSE*. 493–505.

[8] T. Liang, J. Zhao, L. Feng, S. Sinha, and W. Zhang. 2018. Hi-DMM: High-Performance Dynamic Memory Management in High-Level Synthesis. *IEEE TCAD* (2018), 2555–2566.

[9] M. Maas, U. Beaugnon, A. Chauhan, and B. Ilbeyi. 2023. TelaMalloc: Efficient On-Chip Memory Allocation for Production Machine Learning Accelerators. In *Proceedings of ASPLOS*. 123–137.

[10] J. Vuillemin. 1980. A Unifying Look at Data Structures. *CACM* 23, 4 (1980), 229–239.

[11] Q. Wang, L. Zheng, Y. Huang, P. Yao, C. Gui, X. Liao, H. Jin, W. Jiang, and F. Mao. 2021. GraSU: A Fast Graph Update Library for FPGA-based Dynamic Graph Processing. In *Proceedings of FPGA*. 149–159.

[12] Xilinx. 2021. *Vivado Design Suite User Guide High-Level Synthesis*. https://docs.amd.com/v/u/en-US/ug902-vivado-high-level-synthesis.

[13] Z. Xue and D. Thomas. 2015. SysAlloc: A hardware manager for dynamic memory allocation in heterogeneous systems. In *Proceedings of FPL*. 1–7.

[14] Z. Xue and D. Thomas. 2016. SynADT: Dynamic Data Structures in High Level Synthesis. In *Proceedings of FCCM*. 64–71.