Received 13 May 2023, accepted 6 June 2023, date of publication 27 June 2023, date of current version 7 July 2023.

*Digital Object Identifier 10.1109/ACCESS.2023.3290100*

RESEARCH ARTICLE

# High-Performance Memory Allocation on FPGA With Reduced Internal Fragmentation

**MOHAMAD MEHDI SADEGHI[1], SOMAYEH TIMARCHI[2], AND MAHMOOD FAZLALI[2]**
[1]Faculty of Electrical Engineering, Shahid Beheshti University, Tehran 1983969411, Iran
[2]Cybersecurity and Computing Systems Research Group, School of Physics, Engineering and Computer Science, University of Hertfordshire, AL10 9AB Hertfordshire, U.K.

Corresponding author: Somayeh Timarchi (s.timarchi@herts.ac.uk)

**ABSTRACT** In this paper, we present two distinct hardware dynamic memory allocation schemes that are based on the binary buddy system algorithm. Our aim is to mitigate internal fragmentation without impacting the area and performance of the system. The first scheme introduces a parallel design for calculating the addresses of free blocks, which results in a decrease in allocation latency while maintaining acceptable resource utilization. This scheme is particularly well-suited for managing a limited number of minimum allocable units (MAU). On the other hand, the second allocator can handle a large number of MAUs due to its innovative searching mechanism. This allocator exhibits lower resource consumption and operates with an acceptable allocation latency. Furthermore, to decrease internal fragmentation, we develop a novel update mechanism for allocating information data structures in both methods. By employing these two allocator schemes, we can improve the efficiency and resource management of dynamic memory allocation for hardware systems. Experimental results demonstrate that the first and second proposed schemes achieve a minimum allocation speed-up of $\times 2$ and $\times 1.8$ compared to their counterparts. At the same time, they achieve a reduction of at least 78% and %88 in resource utilization, respectively. The results show that the total fragmentation is reduced by at least 14% due to the lower internal fragmentation.

**INDEX TERMS** Dynamic memory allocator, field programmable gate array (FPGA), high-performance, internal fragmentation.

## I. INTRODUCTION

Field programmable gate array (FPGA) devices outperform CPUs and GPUs in terms of speed and power consumption [1], [2], [3]. To support this in the last decade, high-level synthesis (HLS) tools have been developed to raise the abstraction level and ease the design of complex systems with a performance similar to hand-written register transfer level (RTL) codes [4]. HLS tries to convert input hardware specification to an efficient Data path [5], [6], [7]. However, neither today's commercial HLS tools nor hardware description languages (HDL) support the dynamic memory management (DMM) features. This constraint forces hardware designers to statically allocate on-chip memory blocks (BRAM), which is often a limiting factor in designing complex embedded systems [8]. Although modern FPGAs are built with a considerable amount of BRAMs distributed throughout the chip, Diamantopoulos et al. [9] revealed that dynamically (de)allocating memory (from)to units that have varying memory usage patterns through their run-time can alleviate memory-induced dark silicon.

For decades, software designers have utilized DMM to design memory-efficient software. Therefore, they developed many DMM algorithms to manage dynamic memory [10]. Two widely used allocation mechanisms are sequential fit [10] and segregated free list [11]. The former allocator sequentially searches the memory and is generally deployed in software. This type of allocator has a high allocation latency. The latter allocator is faster since it employs an array of free lists, each with free blocks of particular sizes. Some types of segregated free lists, like the buddy system [12], virtually split and merge the memory for efficient usage. The buddy system has a fast allocation response at the cost of high internal fragmentation. The most straightforward type

The associate editor coordinating the review of this manuscript and approving it for publication was Ilaria De Munari.

of buddy system is the binary buddy system (BBS) [12]. However, the BBS algorithm has more internal fragmentation compared to more complicated types of buddy systems like Fibonacci buddies [13] and weighted buddies [14].

Some research focuses on developing dynamic memory allocator (DyMA) algorithms for FPGA platforms [15], [16], [17], [18], [19], [20], [21], [22]. In [17], an allocator based on the sequential fit was presented and employed in many-accelerator architecture to maximize memory usage efficiency. Reference [18] offered an allocator based on the segregated free lists for an allocation suitable for real-time applications. Other research, like [19] and [20], presented a BBS-based mechanism to develop high-speed allocators capable of managing many minimum allocable units (MAUs). Generally, choosing an appropriate hardware allocator depends on the available resources, expected performance, and required memory usage efficiency. In addition to the mentioned cases, different memory request patterns of applications are noticeable [15]. Therefore, there is no unique solution for all situations.

Since FPGAs are low-latency devices and are used as accelerators, employing DyMA algorithms with high allocation latency is unreasonable. Furthermore, some FPGAs, especially low-cost ones, have a limited amount of BRAMs, and therefore, wasting memory through internal fragmentation is not affordable. This paper focuses on the BBS mechanism since it has a rapid allocation response and low resource overhead for implementation compared to other approaches.

Hardware DyMA structures based on the BBS algorithm have at least one bit-vector, whose bit corresponds to one of the heap's MAU to hold the allocation status information. According to the request memory, the algorithm allocates memory mainly in the following three steps: 1) in the first step, the availability of free blocks suitable for the request size are searched, 2) if there is a free space, the start address of the first suitable free block is computed in the second step, and 3) in the last step, the bit-vector(s) is updated. In the context of BBS-based hardware allocators, effectively managing Memory Access Units (MAUs) requires addressing two critical concerns: enhancing performance and minimizing resource overhead. Additionally, there is a pressing need to mitigate internal fragmentation. To this end, we propose two memory allocation strategies:

- One-Level Dynamic Memory Allocator (OLDMA): This allocator avoids latency by minimizing the register insertion to break critical paths and take advantage of the speed of combinational logic. As a result, this is a low allocation latency design but can manage a heap with a limited number of MAUs.
- Two-Level Dynamic Memory Allocator (TLDMA): This allocator offers a novel approach to allocation information storage, along with a sophisticated two-level search method. By utilizing resource-sharing techniques, the TLDMA significantly reduces the overhead of consumed resources. This design is particularly suitable for managing heaps with numerous Memory Allocation Units (MAUs), making it a low-resource overhead solution compared to the first design. The proposed method is appropriate for managing a heap with many MAUs.

The two proposed schemes make noteworthy contributions in the following ways:

- Firstly, they introduce a novel mechanism that can allocate any number of MAUs without the requirement of rounding it up to the nearest power of two.
- Secondly, they enhance the hardware by replacing all multipliers with shift circuits and promoting resource sharing between components whenever feasible.
- Lastly, to maintain maximum frequency (Fmax), the combinational logic with high fan-outs is segregated into multiple subparts with lower fan-outs, thus enabling parallel execution and breaking critical paths.

The achievements of the paper are:

1) Both schemes have low resource utilization overhead compared to their counterparts.

2) Both schemes have high-performance allocation responses suitable for real-time application.

3) Both schemes significantly reduce internal fragmentation compared to standard BBS.

The rest of this paper is as follows: in Section II, we review the related works. In Section III, the proposed DyMA architectures are discussed. In Section IV, the implementation results on FPGA and the comparison will be conducted, and finally, in Section V, we conclude this paper.

## II. RELATED WORK

In this section, we review related works and highlight the pros and cons of each method.

Dessouky et al. [16] proposed a DyMA unit based on a fixed-size allocator called DOMMU, which is customized to apply the dynamic management of on-chip memory to parallel FPGA-based processing elements (PEs). In DOMMU, PEs cannot explicitly request memory because all allocation and deallocation are done automatically by adding/removing one page to/from PEs. The internal memory fragmentation is high when applications need variable sizes of memory. A crossbar interconnection switch is used to simultaneously access PEs to the memory blocks allocated to them, which is a limiting factor of Fmax and introduces a considerable amount of resource utilization.

Diamantopoulos et al. [17] studied DMM in many-accelerators systems and proposed HLS-DMM. The method is a DyMA based on the sequential first fit policy. The internal fragmentation rate of HLS-DMM is low due to the ability to allocate any desired number of MAUs. However, since this method sequentially searches for free memory, in addition to high allocation latency, the external fragmentation rate at the beginning of the heap increases as the procedure continues. So Kokkinis, in another study [22], uses HLS-DMM in a framework for optimizing memory by defragmenting memory.

Özer [18] proposed a free list manager (FLM) designed with the objective of allocation with minimum memory fragmentation by allocating non-continuous blocks and bounded allocation response time to be applicable in real-time systems. The FLM is limited to allocate memory with a size of 1 to 32 MAUs. However, a complicated address translator for accessing BRAM is needed, whose cost is extra cycles to access the BRAM. The high access latency of BRAM causes a decrease in the overall system performance.

Liang et al. [19] introduced an HLS dynamic memory management scheme called Hi-DMM. The method is a computer-aided design (CAD) tool and contains four DyMAs based on the BBS. Hi-DMM analyzes the C source code to replace all malloc/free functions with an appropriate DyMA instance. The DyMAs are designed to be high-performance and use the HLS directive in the implementation process for efficient implementation. Although all DyMAs have low allocation latency, the FPGA's resource consumption is high, and more importantly, Hi-DMM does not address the high internal fragmentation issue of the buddy system algorithm.

Xue and Thomas [20] proposed a DyMA called Sysalloc to improve the hardware allocator developed by Chang and Gehringer [21] to scale up the management capability. The Sysalloc is a BBS-based allocator that can manage dual-data rate (DDR) memories. So, Sysalloc can be used as a DyMA of internal and external memories. Nevertheless, Sysalloc suffers from high allocation latency around hundreds of cycles and does not address the high internal fragmentation of the buddy system algorithm.

Giambalanco et al. have presented a library of HLS allocators in [15] that includes five allocators. These allocators are tool independent, and some of them are efficient in memory usage. However, they have an allocation latency of hundreds of clock cycles that varies depending on the number of MAUs. According to the authors' claim, the proposed scheme aimed to reduce memory usage and area consumption and achieve high Fmax.

The characteristics of an ideal allocator include low fragmentation, as well as low allocation and maintenance latencies. For hardware realization, the implementation cost and Fmax are also noticeable. Our work attempts to improve these characteristics of the allocator, although compromise is inevitable.

## III. PROPOSED DYNAMIC MEMORY ALLOCATORS

This section discusses on the proposed allocation methods and their corresponding implementations. Let's assume the input signals of the allocator are request-size, free-addr, and allocate. The request-size indicates the number of MAUs that are required to be allocated/deallocated. It indicates the number of bits that will be flipped in the bit-vector at the end of process. The free-addr is the start address of flipped bits in deallocation process. The allocate signal is used to indicate the allocation/deallocation command. The output signal, actual-addr, is the start address of allocated memory. The two proposed allocators are presented in the following.

### A. OLDMA

The first proposed allocator, called OLDMA, is shown in Fig. 1. As shown in the figure, this structure uses a bit-vector, called pBV, with a length of $2^n$ bits equivalent to the number of MAUs to hold their allocation status. The OLDMA comprises three main components: OR-TREE, Find-Address, and Bit-Flipper.

- In the allocation process, the OR-TREE component checks if there are continuous free blocks. If there are continuous free blocks, the Find-Address component will compute the start address of the first free block, called start-addr. Then, the Bit-Flipper updates the pBV.
- In the deallocation process, Bit-Flipper will flip an appropriate number of bits specified by the request-size to zero, starting from free-addr.

The performance bottleneck in BBS-based allocators is to search for allocable memory blocks and update the OR-GATE tree bit-vectors. The key ideas of the proposed OLDMA are accelerating these two procedures. The issue is done by exploring new methods for searching, performed by the Find-Address component, and updating the bit-vector, performed by the Bit-Flipper component. This paper develops an architecture for the Find-Address component to perform the searching process. Besides, our new method for updating the bit-vector is to employ the basic idea of bitmask to specify the flipped MAUs.

#### 1) OR-TREE COMPONENT

The main part of the OR-TREE component is the OR-GATE tree presented in [21] to determine if there are continuous free blocks. As shown in the article, each row in the tree has a depth and a bit-vector. The tree needs a depth parameter, called $D$, to specify which bit-vector must be searched and is calculated according to the following relation:

$$D = L - \text{ceil}(\text{log2}(\text{request} - \text{size})) \tag{1}$$

where $L$ is equal to log2(sizeof (pBV)). Based on the depth calculated by (1), the OR-TREE loads the bit-vector at depth $D$ of the OR-GATE tree into the Reg-Arr register. Then the Find-Address uses the Reg-Arr to calculate the start address of the first $2^{L-D}$ continuous zero-bits in the pBV. In our implementation, the logarithm operation is implemented using 2:1 multiplexers.

#### 2) FIND-ADDRESS COMPONENT

To reduce the critical path delay caused by the combinational implementation of the AND-GATE tree presented in [21] and to avoid inserting registers to break the critical path, which causes latency, as presented in [20], the proposed OLDMA uses a new architecture to extracts the first zero in the bit-vector at depth D of the OR-GATE. The basic idea is to design a new structure to reduce the critical path delay of finding the address.

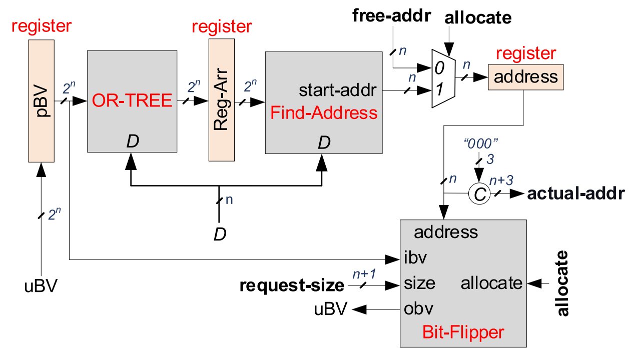

**FIGURE 1. Proposed OLDMA block diagram without depicting control signals by assuming MAU size equal to 8 words.**

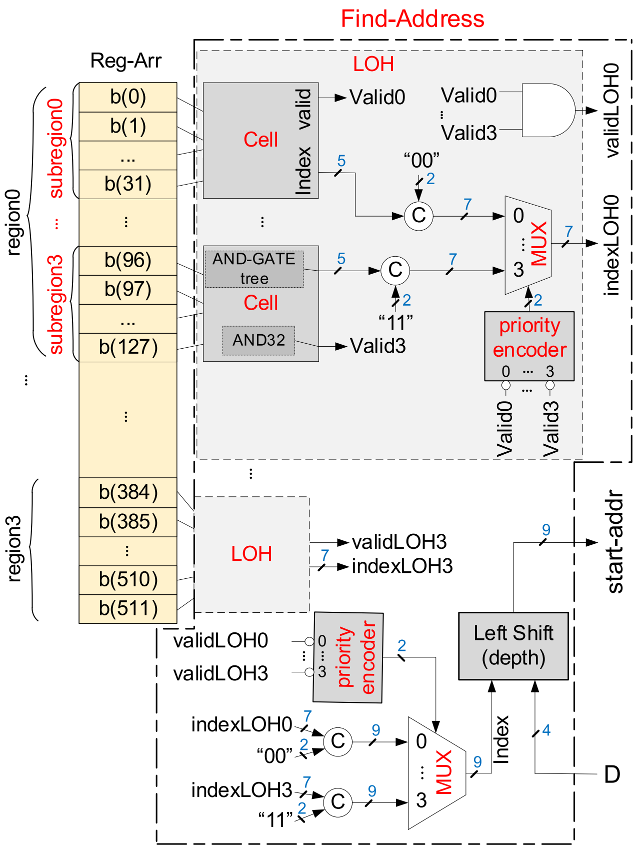

**FIGURE 2. Block diagram of proposed Find-Address component with two-level hierarchy. The Reg-Arr divides into 4 regions. Also, each 128-bit region divides into 4 subregions.**

The rest of the subsection discusses the implementation of the Find-Address component of the proposed OLDMA. As shown in Fig. 2, the Reg-Arr is divided into multiple regions and subregions in the proposed architecture. Let us assume there are four regions (region0 to region3), and each region is divided into four 32-bit subregions (subregion0 to subregion3), as shown in Fig. 2. Each subregion's first-zero-bit index, i.e., the index of the first bit with a zero value, in this architecture is extracted using a 32-bit AND-GATE tree. Since any subregion may not have a zero-bit, a signal named *valid*, calculated by AND32 in the figure, indicates the validity of the AND-GATE tree's output, i.e., index. The AND-GATE tree as well as AND32 are shown by a unit called Cell as shown in the figure.

The index of first-zero-bit in the region is computed using the concatenation of the highest priority subregion's index with a valid index and the corresponding subregion number, assuming that the highest priority subregion number is 0. The process is performed by a priority encoder for each subregion. So far, the index of first-zero-bit in each region is computed by LOH units. Finally, a similar procedure that computes the whole region's first-zero-bit is employed to compute the first-zero-bit of the Reg-Arr. The following equation is used to compute the start-addr of the first block of $2^{L-D}$ continuous zero-bits in the pBV [21].

$$\text{start}-\text{addr} = 2^{L-D} \times \text{index} \tag{2}$$

where $D$ is the depth calculated by (1), start-addr is the start address of the first $2^{L-D}$ continuous zero-bits in the pBV, and index shows the least significant zero-bit in the Reg-Arr. Since the MAUs size could be greater than 1, the actual address of the allocated block in a heap, represented by actual-addr, which is returned to the application, is calculated as follows:

$$\text{actual}-\text{addr} = \text{start}-\text{addr} \times \text{sizeof(MAU)} \tag{3}$$

In the case of allocation / deallocation, the start-addr/free-addr is entered into the Bit-Flipper to update the pBV.

### 3) BIT-FLIPPER COMPONENT

As performance issues have been found to be connected to the method of updating the OR-GATE tree in BBS-based allocators, the OLDMA presents a novel approach that steers clear of the traditional iterative update mechanism used in previous works. Specifically, the OLDMA doesn't store the OR-GATE tree in memory but instead integrates it as a combinational circuit. This allows the tree to be updated instantaneously when the relevant pBV changes, and without memory access. The result is a significantly faster update process. Besides, updating only pBV offers more control to flip any desired number of bits compared to the marking downward employed in [19] and [20]. The marking downward approach updates the OR-GATE tree from the layer located in the allocation stage to the bottom layer. In the approach, the input combination of the OR logic must be determined by knowing the result of OR logic which is not possible in all cases, and over-allocation is inevitable.

In the updating process, the bits corresponding to the allocated and deallocated MAUs flip to '1' and '0', respectively. Bit-Flipper adopts the basic idea of bitmask to specify the flipped MAUs. As shown in Fig. 3, to specify the flipped bits, the pBV is divided into $N$ groups with size $S$. So, assigning values to pBV's bits is done based on the group type of the bits. Since each group belongs to one of the following types: macro-region, micro-region, or const-region, the type of each group must be determined first. All bits of

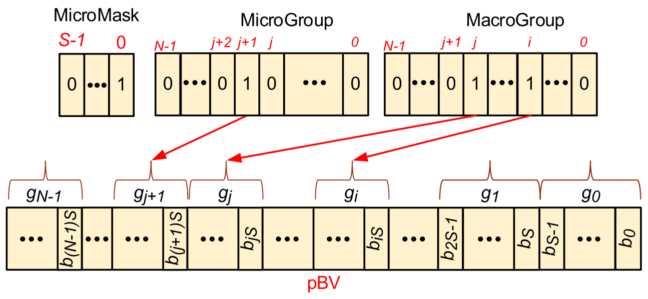

**FIGURE 3.** Example of updating pBV'bits based on the group type by assuming that the pBV is divided into N group with S-bit size.

**Algorithm 1** Bit-Flipper (ibv, size, address, allocate)

```
1: MacSize = floor(size÷)$S$
2: MacAddr = floor(address÷)$S$
3: MacroGroup = $2^{\text{MacAddr}}\times(2^{\text{MacSize}}-1)$
4: MicroGroup = $2^{\text{MacAddr+MacSize}}$
5: MicSize= size%$S$
6: MicroMask= $(2^{\text{MicSize}}-1)$
7: MicroMask'= firstcomplement(MicroMask)
8: for idx in 0 to $N$-1:
9:     for idyin 0 to $S$-1:
10:        $i$= idx×$S$+idy
11:        if MacroGroup [idx] ORMicroGroup[idx] = 1 then
12:            if allocate = 1 then
13:                Tmp = ibv[$i$]ORMicroMask[idy]
14:                obv[$i$] = TmpORMacroGroup[idx]
15:            Else
16:                Tmp= ibv[$i$]ANDMicroMask'[idy]
17:                obv[$i$]=TmpANDNOT(MacroGroup[idx])
18:            end if
19:        Else
20:            obv[$i$] = ibv[$i$]
21:        end if
22:    end for
23: end for
24: returnobv
```

the macro-region group are flipped, but in the micro- region group, just some bits will flip. The bits of the const-region do not change at all. Multiple macro-regions may exist in each (de)allocation, but at most, one micro-region may exist. Two vectors are used to specify the macro-regions and micro-region, called MacroGroup and MicroGroup, respectively. Furthermore, MicroMask is used to represent the flipping bits in the micro-region.

The proposed Bit-Flipper component is implemented based on Algorithm 1, which receives the pBV (ibv), the number of bits that must be flipped (size), the start addresses of flipped bits (address) which in the case of allocation and deallocation is the start-addr returned by the Find-Address component, and free-addr, respectively, and (de)allocation command (allocate = 1 for allocation and allocate = 0 for deallocation). In Algorithm 1, to calculate MacroGroup in line 3, two parameters, MacSize and MacAddr, are needed, which indicate the number and start address of macro-region groups, respectively. The MicroGroup value is also calculated in line 4 to indicate the micro-region group. Also,

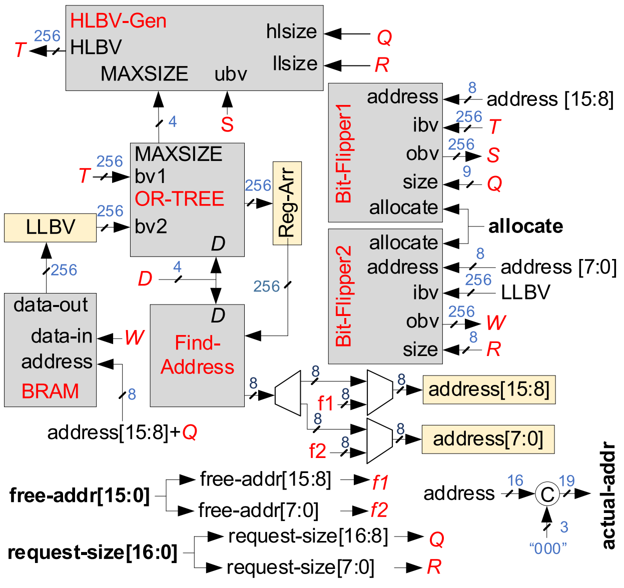

**FIGURE 4.** TLDMA block diagram without representing control unit signals. The $2^{16}$-bit pBV is divided into 256 subBVs with size $2^8$ bits.

to specify the flipped bits in the micro-region, the MicroMask vector is calculated in line 6, which only needs the number of flipped bits in that micro-region. Lines 8-21 assign a value to the bits of the pBV according to their corresponding group and allocation command.

In the algorithm, the inner loop is repeated over the group bits (0 to $S$-1), and the outer loop is repeated over the groups (0 to $N$-1). In line 9, each group type is checked, and if it is a macro-region or micro-region, the corresponding bits are assigned by a value accordingly. In the case of allocation, the value of flipped bits is calculated and assigned in lines 13 and 14. In the case of deallocation, the value of flipped bits is calculated and assigned in lines 16 and 17.

The proposed OLDMA allocates memory with very low latency. Nevertheless, by increasing the pBV size, resource consumption becomes the bottleneck of the OLDMA. To overcome this shortcoming, the second allocator is proposed.

## *B. TLDMA*

To manage a large number of MAUs, the proposed TLDMA design is motivated by the hierarchical structure idea based on the group tree [20], except that TLDMA constructs the

group tree rather than storing them in memory. Therefore, the allocation process is performed with less memory access, resulting in a lower allocation latency. As mentioned in the previous subsection, updating from the bottom layer to the top layer of the OR-GATE tree allows more control to allocate any desired number of MAUs. The TLDMA utilizes this approach for updating.

Fig. 4 depicts the block diagram of TLDMA composed of these components: BRAM, Find-Address, OR-TREE, two Bit-Flipper, and HLBV-Gen. The Find-Address and Bit-Flipper components were discussed in the previous

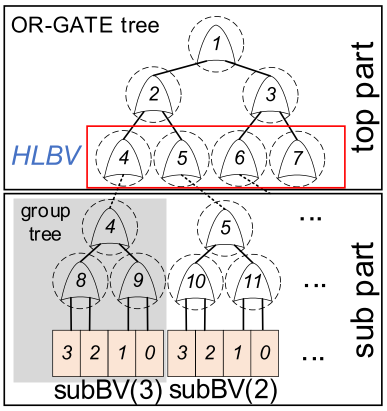

**FIGURE 5.** Top and sub parts of the OR-GATE tree in the proposed TLDMA scheme.

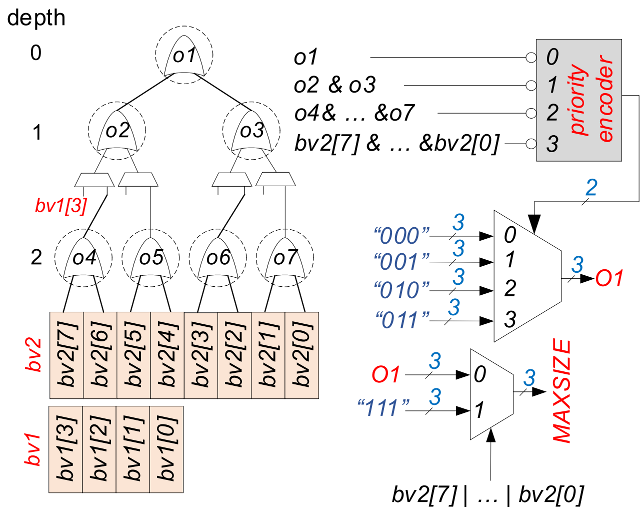

**FIGURE 6.** Proposed OR-TREE block diagram. Assuming that size of bv1 and bv2 is $2^2$, and $2^3$ respectively, and MAXSIZE has 3-bits.

subsection. TLDMA divides the pBV into multiple subBV, where the number of subBV $s$ is a power of two, and stores them in a BRAM, with the word size equal to the subBV size. The rest of this section explains the new components, steps of allocation, and update processes.

#### 1) OR-TREE COMPONENT

In the two-level search mechanism, the OR-GATE tree consists of two parts as shown in Fig. 5. If the requested size is larger than the size of a subBV, a bit-vector named HLBV takes a value such that the top part of the OR-GATE tree is constructed, and the memory search is performed in a specific layer in the top part. If the requested size is smaller than the subBV size, the memory search is performed in a particular layer in the sub part. In this case, the HLBV specifies the subBVs with a free node in the desired layer. Then the OR-GATE tree corresponding to the first allocable subBV is formed, and the memory search resumes. In both cases, the largest allocable continuous zero-bits (LACZB) of each subBV is used to generate the HLBV. In the two-level search mechanism, the OR-GATE tree is formed once for HLBV and once for subBV at two different times.

The proposed OR-TREE in TLDMA manages two different bit-vectors. As depicted in Fig. 6, this structure employs resource sharing between two separate trees. Let's assume two OR-GATE constructed on top of the two bit-vectors bv1 and bv2 with the size equal to $2^2$ and $2^3$, respectively. In this case, the top two layers of the larger tree share with the smaller tree. When the smaller tree is selected, the value of the bit-vector at depth 1 is set based on the bit-vector bv1. Otherwise, the value of the bit-vector at layer 1 is based on the underlayer bit-vector at depth 2. Also, the proposed OR-TREE calculates the MAXSIZE for the larger bit-vector according to the following equation.

$$\begin{aligned}\text{MAXSIZE} = (!\text{LACZB})?2^{\text{ceil}(\log_2(L)+1)} \\ -1 : L-\text{ceil}(\log_2(\text{LACZB}))\end{aligned} \tag{4}$$

In (4), $L$ is a constant and equal to $\log_2(\text{sizeof(subBV)})$.

#### 2) HLBV-GEN COMPONENT

The general operation of this component is that a bit vector called HLBV is generated in the first step of the allocation process by using the corresponding MAXSIZE of each subBV stored in the matrix called HLRA and according to the requested size. Each bit of HLBV corresponds to the availability of a subBV, '1' indicates subBV is occupied, and '0' shows it is free. The OR-TREE component takes HLBV and generates the top layers of the OR-GATE tree. The comparator employed in this component is implemented according to the following equations.

$$r = (\text{hlsize} > 0)\,?0 : L-\text{ceil}(\log_2(\text{llsize})) \tag{5}$$

$$\text{result}_i = (\text{MAXSIZE}_i > r)\,?1 : 0 \tag{6}$$

In (5), the hlsize is the number of totally occupied subBVs, and the llsize is equal to the remainder of dividing request-size by subBV size. This way, the output bit-vector called HLBV is generated by concatenating the comparison results. After each memory allocation or in the deallocation process, MAXSIZEs corresponding to the subBVs involved in the allocation/deallocation process must be updated. The two update modes are as follows.

- umode1: In this mode, in the allocation process, if the $i$th bit of ubv, the input signal to the HLBV-Gen, is '1', the updated value of the $\text{MAXSIZE}_i$ is $2^{\text{ceil}(\log_2(L)+1)}-1$. In the deallocation process, if the ith bit of ubv is '0', the updated value of the $\text{MAXSIZE}_i$ is 0.
- umode2: In this mode, only MAXSIZE corresponding to the subBV loaded in the LLBV register is updated using MAXSIZE computed by the proposed OR-TREE.

#### 3) ALLOCATION AND UPDATE STEPS

As previously discussed, in TLDMA, memory allocation is done in three steps. The steps and active components in each step are described in the following.

**TABLE 1.** Hardware characteristics of the dynamic memory allocators.

| | OLDMA | TLDMA | FBTA [19] | HTA [19] | Sysalloc [20] | lutmem [15] | Memory optimization framework [22] | DOMMU [16] | FLM [18] |
|---|---|---|---|---|---|---|---|---|---|
| FPGA device | Zynq-7000 SoC XC7Z020 | Zynq-7000 SoC XC7Z020 | Zynq-7000 SoC XC7Z020 | Zynq-7000 SoC XC7Z020 | Zynq-7000 SoC XC7Z020 | Stratix V 5SGXEA7N2F45C2 | Alveo U200 acceleration card | Virtex-5 LX110T | Kintex-7 XC7K325T |
| Max LUT | 4.08% | 2.66% | 19% | 23% | 8% | 0.26% | 0.71% | 17% | 11.9% |
| Fmax (MHZ) | 100 | 100 | 100 | 100 | 100 | > 500 | unknown | 140 | 175 |
| Allocation Latency | 3 | 8 | 7-8 | 7-19 | 81-83 | > 100 | 20% of execution time | Unknown | 21 |
| Maximum MAUs | 512 | 64K | 512 | 64K | 32K | 4K | 120K | 40 | 32 |
| Memory utilization | yes | yes | no | no | no | yes | yes | no | yes |

#### *a: STEP1*

In step1, according to the requested size, the HLBV-Gen generates the HLBV which specifies the appropriate subBVs. Since the requested size may be larger than a subBV, an adequate number of continuous free subBVs must be available. Otherwise, the allocation process will fail. The OR-TREE loads the bit-vector at depth $D$ of the OR-GATE tree into the Reg-Arr and Find-Address calculates the start address of the first suitable group of subBV $s$. The Find-Address's output is loaded into the most significant bits of the address register. Since the requested size is not always a factor of the subBV size, to prevent memory wastage due to internal fragmentation, the last subBV of the group is loaded into the LLBV register to be processed in step2. In cases where only one subBV is required, the subBV is loaded into the LLBV register after calculating the address of the first appropriate subBV. In this step, the bv1 is selected in the OR-TREE. The start-addr calculated by Find-Address is loaded into the most significant bits of the address register.

In step1, the $D$ assigned to the OR-TREE and Find-Address is calculated by (1) where the request-size is the number of totally occupied subBVs in the allocation.

#### *b: STEP2*

As discussed earlier, since multiple subBV $s$ may need to be allocated, after computing the address of the first suitable group of subBV$s$, all the subBV$s$ in the group except the last subBV will be occupied thoroughly.

So, to speed up the updating process, in this step, the corresponding MAXSIZE$s$ of the subBVs specified by the Bit-Flipper1 output are updated according to umode1. To continue the allocation process in step2, by selecting the bv2, the OR-TREE loads the bit-vector at the $D$ of the OR-GATE tree into the Reg-Arr and Find-Address calculates the start address of the first suitable group of bits. The calculated address by Find-Address is loaded into the least significant bits of the address register. When the allocation and updating processes in step2 are finished, the BRAM and LLBV are updated by Bit-Flipper2 output.

In step2, the $D$ assigned to the OR-TREE and Find-Address is calculated by (1) where request-size is the number of flipped bits in LLBV.

#### *c: STEP3*

In the third step, the corresponding MAXSIZE of the subBV loaded into the LLBV is generated by the OR-TREE component and sent to the HLBV-Gen to update the HLRA matrix according to the umode2.

## IV. EVALUATION AND ANALYSIS

To evaluate the designs in terms of resource utilization, we implemented the proposed designs in Zynq-7000 SoC XC7Z020 and set the frequency at 100 MHZ. To synthesize and simulate the proposed allocators, we use Vivado suite version 2019.2. The Sysalloc presented in [20], FBTA and HTA presented in [19] are selected as the methods for comparison. We consider resource utilization (LUT and BRAM utilization), as well as allocation and update latencies as the metrics to evaluate the DyMAs. Another metric is memory utilization because the objective of using DyMAs is to reduce memory inefficiency. To evaluate memory usage, the applications and DyMAs emulator have been implemented using Python. The required parameters, such as the actual requested size, allocated memory size, and the largest allocated address, are monitored in the software. The allocation latency is the number of clock cycles that an application must wait for memory allocation.

On the other hand, the update latency is the number of clock cycles required for an allocator to update its bit-vector. So, the (de)allocation request will not be responded until the update is finished.

Table 1 reveals the hardware characteristics of existing allocators. In the table, memory utilization means reusing the memory space while keeping the internal fragmentation low.

### *A. RESOURCE UTILIZATION*

For each allocator, the BRAM and LUT utilization corresponding to different numbers of MAUs is shown in Fig. 7.(a) and Fig. 7.(b), respectively. The implementation parameters,

**TABLE 2.** **Configuration parameters of the proposed OLDMA.**

| pBV size | number of hierarchy level | region size | subregion size | group size |
|---|---|---|---|---|
| 32 | 0 | - | - | - |
| 64 | 1 | 32 | - | 32 |
| 128 | 1 | 32 | - | 32 |
| 256 | 2 | 128 | 32 | 32 |
| 512 | 2 | 128 | 32 | 32 |

**TABLE 3.** **Configuration parameters of the proposed TLDMA.**

| number of *subBV* | *LLBV* size | number of hierarchy level | region size | subregion size | group size |
|---|---|---|---|---|---|
| 32 | 32 | 0 | - | - | - |
| 64 | 64 | 1 | 32 | - | 32 |
| 128 | 128 | 1 | 32 | - | 32 |
| 256 | 256 | 2 | 128 | 32 | 32 |

i.e., region and subregion sizes, the number of hierarchical levels in the Find-Address, the group size in Bit-Flipper, and the number and size of subBVs in the TLDMA, are given in Tables 2 and 3. All allocators except OLDMA use BRAMs to store bit-vector. As shown in Fig. 7.(a), in the worst case, the BRAM utilization of TLDMA is 2.86% of FPGA's BRAMs, around 40% lower than HTA [19] and Sysalloc [20]. The OLDMA does not need any BRAM, while FBTA [19] needs up to 24% of available BRAMs, as its counterpart. As depicted in Fig. 7.(b), the LUT utilization of both OLDMA and TLDMA increases exponentially when the number of MAUs increases. So, they are limited to manage a certain number of MAUs. Both OLDMA and TLDMA consume considerably fewer LUTs compared to their counterparts. To (de)allocate 512 and 64K MAUs, FBTA [19] and HTA [19] consume around 19% and 23% of FPGA LUTs compared to 4.08% and 2.66% for OLDMA and TLDMA,respectively.

## B. ALLOCATION LATENCY

There are two types of latencies: allocation and update. The allocation latency of both OLDMA and TLDMA is constant and equal to 3 and 8 clock cycles, respectively. OLDMA is faster than TLDMA at the cost of more resource utilization because TLDMA searches the *HLBV* first and then searches the LLBV. The HTA [19] and Sysalloc [20] have variable allocation latency, depending on the requested size and MAUs number, but FBTA [19] has a constant allocation latency of 7-8 cycles depending on the MAUs number. Nevertheless, the actual latency of an allocator is equal to the summation of the allocation and update latencies. Let's assume that an allocator has the allocation and update latencies of $M$ and $N$ clock cycles, respectively. When two applications request memory allocation with $I$ clock cycles interval, i.e., the second request came $I$ clock cycles after the first request, in cases where $I \leq M + N$, the second application must wait for $2 \times M + N - I$ clock cycles until the memory is allocated to it. So, as depicted in Fig. 7.(c), the actual latency of OLDMA and TLDMA with a request interval of 2 clock cycles is 7 and

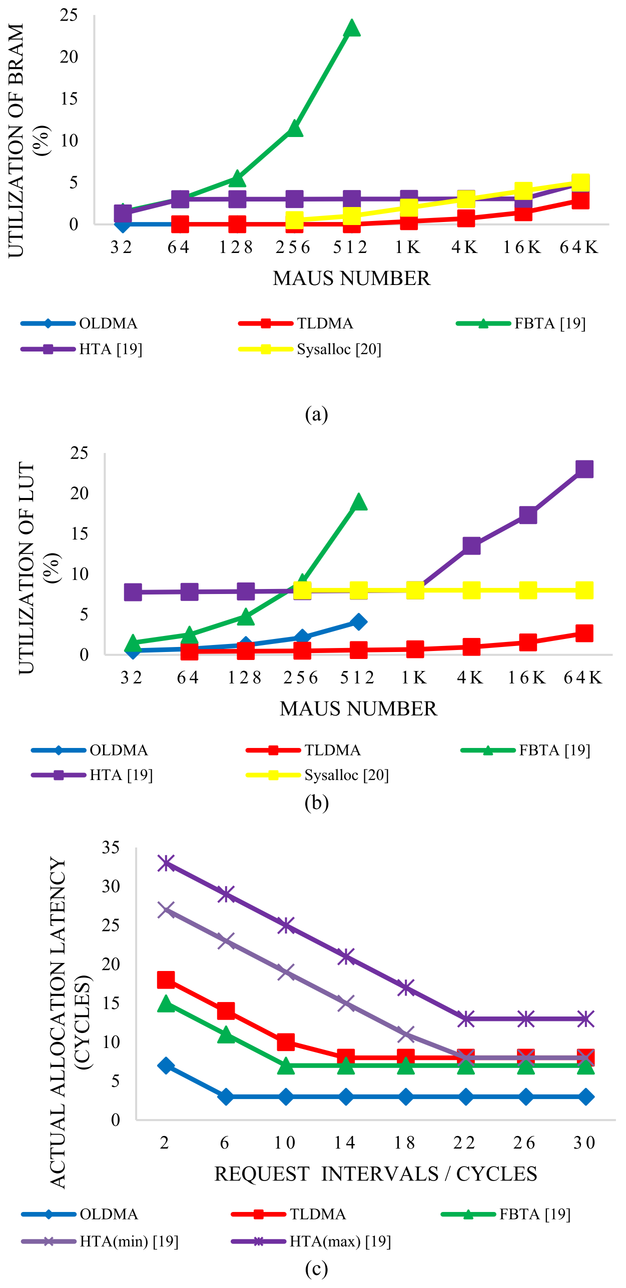

**FIGURE 7.** **Resource utilization compared to FBTA [19], HTA [19], and Sysalloc [20]. a) BRAM utilization. b) LUT utilization. c) actual allocation latency.**

19 clock cycles, respectively. TLDMA has higher latency due to its structure, which needs to update multiple bit-vectors but is not much greater than other allocators.

## C. MEMORY UTILIZATION

Although OLDMA and TLDMA can allocate any desired number of MAUs, if the requested size is not a factor of the MAU size, the internal fragmentation may not be zero. The internal fragmentation rate depends on the MAU size and distribution of requests [23]. Since internal fragmentation is the main factor of memory waste in the BBS algorithm [21],

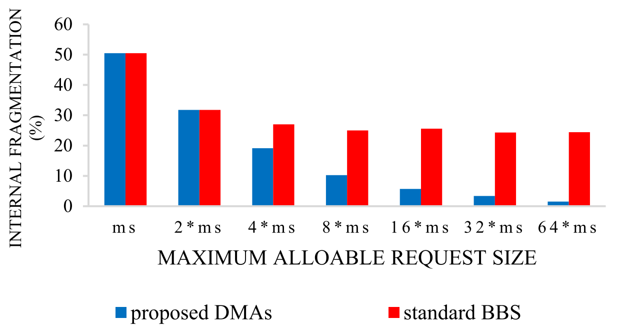

**FIGURE 8.** Comparison of internal fragmentation of proposed DyMA with standard binary buddy system, where "ms" is the MAU size.

we create a software-desired request pattern generator with uniform request size distribution to evaluate internal fragmentation. The internal fragmentation metric, considered in this paper, is the ratio of over-allocated memory to totally allocated memory [21]. As shown in Fig. 8, the internal fragmentation rate in the proposed DyMAs goes to zero by expanding the requested size range. Although external fragmentation depends on the memory usage footprint of an application, two widely used machine learning applications have been used in this study to evaluate the effect of reducing the internal fragmentation rate on memory consumption. The evaluation metric for external fragmentation is the highest allocated address [21]. The k-means algorithm is the first algorithm implemented and tested based on [21]. The SVM algorithm [24] is also implemented, and datasets a1a to a9a [25] are selected to train a model. The memory usage monitoring shows that the proposed DyMAs allocate 28.71% less memory than the standard BBS because it has a lower internal fragmentation rate. The lower internal fragmentation causes a smaller highest address allocated, such that the highest memory address in the k-means and SVM algorithms is 17.94% and 14.71% smaller when using the proposed DyMA. However, it is possible that in some particular footprints [21], the proposed DyMA fails to allocate the memory where the standard buddy system works fine.

The proposed DyMA is designed to be used in HLS, making it versatile for use in both conventional number system hardware (like two's complement, and binary-coded decimal [26]) and unconventional number system hardware, such as redundant or residue number system [27], [28].

## V. CONCLUSION

This paper presents two high-speed and low-area hardware DyMA schemes based on the BBS algorithm. These schemes have been designed to ensure a fast response time without internal fragmentation. To accomplish this, we introduced new solutions such as minimizing memory access, utilizing low fan-out circuits to break the critical path of combinational logic, and reorganizing the critical path for parallel implementation. Our OLDMA scheme prioritizes rapid allocation, which can be achieved by dividing the heap into fewer but larger MAUs, providing applications with an efficient allocator. Alternatively, the TLDMA scheme is optimized to manage numerous MAUs while consuming very few resources. OLDMA and TLDMA schemes require at most 4.08% and 2.66% of Zynq-7000 SoC XC7Z020 FPGA LUTs for implementation, considerably lower than other reported DyMAs, without compromising the allocation or maintenance latencies. The proposed allocators also improve the allocation latency compared to the other allocators, such that OLDMA and TLDMA have at least ×2, ×1.8 allocation latency improvement over their counterparts, respectively. The internal fragmentation rate in both schemes has been diminished by allocating 28% less memory than the standard BBS algorithm.

## REFERENCES

[1] C. Liu, "YOLOv2 acceleration using embedded GPU and FPGAs: Pros, cons, and a hybrid method," *Evol. Intell.*, vol. 15, no. 4, pp. 2581–2587, Dec. 2022, doi: 10.1007/s12065-021-00612-y.

[2] E. Nurvitadhi, G. Venkatesh, J. Sim, D. Marr, R. Huang, J. O. G. Hock, Y. T. Liew, K. Srivatsan, D. Moss, S. Subhaschandra, and G. Boudoukh, "Can FPGAs beat GPUs in accelerating next-generation deep neural networks?" in *Proc. ACM/SIGDA Int. Symp. Field-Program. Gate Arrays*, New York, NY, USA, Feb. 2017, pp. 5–14, doi: 10.1145/3020078.3021740.

[3] L. Guo, J. Lau, Z. Ruan, P. Wei, and J. Cong, "Hardware acceleration of long read pairwise overlapping in genome sequencing: A race between FPGA and GPU," in *Proc. IEEE 27th Annu. Int. Symp. Field-Program. Custom Comput. Mach. (FCCM)*, San Diego, CA, USA, Apr. 2019, pp. 127–135, doi: 10.1109/FCCM.2019.00027.

[4] A. Canis, J. Choi, M. Aldham, V. Zhang, A. Kammoona, J. H. Anderson, S. Brown, and T. Czajkowski, "LegUp: High-level synthesis for FPGA-based processor/accelerator systems," in *Proc. 19th ACM/SIGDA Int. Symp. Field Program. Gate Arrays*, New York, NY, USA, Feb. 2011, pp. 33–36, doi: 10.1145/1950413.1950423.

[5] M. Fazlali, A. Zakerolhosseini, A. Shahbahrami, and G. Gaydadjiev, "High speed merged-datapath design for run-time reconfigurable systems," in *Proc. Int. Conf. Field-Program. Technol.*, Sydney, NSW, Australia, Dec. 2009, pp. 339–343, doi: 10.1109/FPT.2009.5377678.

[6] M. Fazlali, M. K. Fallah, M. Zolghadr, and A. Zakerolhosseini, "A new datapath merging method for reconfigurable system," in *Reconfigurable Computing: Architectures, Tools and Applications* (Lecture Notes in Computer Science), vol. 5453, J. Becker, R. Woods, P. Athanas, and F. Morgan, Eds. Berlin, Germany: Springer, 2009, pp. 163–174, doi: 10.1007/978-3-642-00641-8_17.

[7] M. Fazlali, A. Zakerolhosseini, M. Sabeghi, K. Bertels, and G. Gaydadjiev, "Data path configuration time reduction for run-time reconfigurable systems," in *Proc. ERSA*, Jul. 2009, pp. 323–327.

[8] M. Platzner and N. Wehn, Eds., *Dynamically Reconfigurable Systems: Architectures, Design Methods and Applications*. New York, NY, USA: Springer, 2010.

[9] D. Diamantopoulos, S. Xydis, K. Siozios, and D. Soudris, "Mitigating memory-induced dark silicon in many-accelerator architectures," *IEEE Comput. Archit. Lett.*, vol. 14, no. 2, pp. 136–139, Jul. 2015, doi: 10.1109/LCA.2015.2410791.

[10] P. R. Wilson, M. S. Johnstone, M. Neely, and D. Boles, "Dynamic storage allocation: A survey and critical review," in *Memory Management* (Lecture Notes in Computer Science), vol. 986, H. G. Baler, Ed. Berlin, Germany: Springer, 1995, pp. 1–116, doi: 10.1007/3-540-60368-9_19.

[11] W. T. Comfort, "Multiword list items," *Commun. ACM*, vol. 7, no. 6, pp. 357–362, Jun. 1964, doi: 10.1145/512274.512288.

[12] K. C. Knowlton, "A fast storage allocator," *Commun. ACM*, vol. 8, no. 10, pp. 623–624, 1965, doi: 10.1145/365628.365655.

[13] D. S. Hirschberg, "A class of dynamic memory allocation algorithms," *Commun. ACM*, vol. 16, no. 10, pp. 615–618, Oct. 1973, doi: 10.1145/362375.362392.

[14] K. K. Shen and J. L. Peterson, "A weighted buddy method for dynamic storage allocation," *Commun. ACM*, vol. 17, no. 10, pp. 558–562, Oct. 1974, doi: 10.1145/355620.361164.

[15] N. V. Giamblanco and J. H. Anderson, "A dynamic memory allocation library for high-level synthesis," in *Proc. 29th Int. Conf. Field Program. Log. Appl. (FPL)*, Barcelona, Spain, Sep. 2019, pp. 314–320, doi: 10.1109/FPL.2019.00057.

[16] G. Dessouky, M. J. Klaiber, D. G. Bailey, and S. Simon, "Adaptive dynamic on-chip memory management for FPGA-based reconfigurable architectures," in *Proc. 24th Int. Conf. Field Program. Log. Appl. (FPL)*, Munich, Germany, Sep. 2014, pp. 1–8, doi: 10.1109/FPL.2014.6927471.

[17] D. Diamantopoulos, S. Xydis, K. Siozios, and D. Soudris, "Dynamic memory management in Vivado-HLS for scalable many-accelerator architectures," in *Applied Reconfigurable Computing* (Lecture Notes in Computer Science), vol. 9040, K. Sano, D. Soudris, M. Hübner, and P. Diniz, Eds. Cham, Switzerland: Springer, 2015, pp. 123–136, doi: 10.1007/978-3-319-16214-0_10.

[18] C. Özer, "A dynamic memory manager for FPGA applications," M.S. thesis, Graduate School Natural Appl. Sci., Middle East Tech. Univ., Çankaya, Turkey, 2014. [Online]. Available: http://etd.lib.metu.edu.tr/upload/12617472/index.pdf

[19] T. Liang, J. Zhao, L. Feng, S. Sinha, and W. Zhang, "Hi-DMM: High-performance dynamic memory management in high-level synthesis," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 37, no. 11, pp. 2555–2566, Nov. 2018, doi: 10.1109/TCAD.2018.2857040.

[20] Z. Xue and D. B. Thomas, "SysAlloc: A hardware manager for dynamic memory allocation in heterogeneous systems," in *Proc. 25th Int. Conf. Field Program. Log. Appl. (FPL)*, London, U.K., Sep. 2015, pp. 1–7, doi: 10.1109/FPL.2015.7293959.

[21] 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, doi: 10.1109/12.485574.

[22] A. Kokkinis, D. Diamantopoulos, and K. Siozios, "Dynamic optimization of on-chip memories for HLS targeting many-accelerator platforms," *IEEE Comput. Archit. Lett.*, vol. 21, no. 2, pp. 41–44, Jul. 2022, doi: 10.1109/LCA.2022.3190048.

[23] J. L. Peterson and T. A. Norman, "Buddy systems," *Commun. ACM*, vol. 20, no. 6, pp. 421–431, Jun. 1977, doi: 10.1145/359605.359626.

[24] C.-C. Chang and C.-J. Lin, "LIBSVM: A library for support vector machines," *ACM Trans. Intell. Syst. Technol.*, vol. 2, no. 3, pp. 1–27, Apr. 2011, doi: 10.1145/1961189.1961199.

[25] R.-E. Fan. (Apr. 2021). LIBSVM. National Taiwan University. [Online]. Available: https://www.csie.ntu.edu.tw/ cjlin/libsvmtools/datasets/

[26] M. Fazlali, H. Valikhani, S. Timarchi, and H. T. Malazi, "Fast architecture for decimal digit multiplication," *Microprocessors Microsyst.*, vol. 39, nos. 4–5, pp. 296–301, Jun. 2015, doi: 10.1016/j.micpro.2015.01.004.

[27] H. Mahdavi and S. Timarchi, "Improving architectures of binary signed-digit CORDIC with generic/specific initial angles," *IEEE Trans. Circuits Syst. I, Reg. Papers*, vol. 67, no. 7, pp. 2297–2304, Jul. 2020, doi: 10.1109/TCSI.2020.2978765.

[28] S. Timarchi and M. Fazlali, "Generalised fault-tolerant stored-unibit-transfer residue number system multiplier for moduli set $\{2^n-1, 2^n, 2^n+1\}$," *IET Comput. Digit. Techn.*, vol. 6, no. 5, pp. 269–276, Sep. 2012, doi: 10.1049/iet-cdt.2011.0075.

**MOHAMAD MEHDI SADEGHI** was born in Shiraz, Fars, Iran, in 1995. He received the B.S. degree in electronic engineering from the Shiraz University of Technology (SUTECH), Fars, in 2018, and the M.Sc. degree in digital electronic engineering from Shahid Beheshti University (SBU), Tehran, Iran, in 2022. His main research interests include computer architecture and high-performance computing.

**SOMAYEH TIMARCHI** received the B.Sc. degree in computer engineering from Shahid Beheshti University, the M.Sc. degree in computer system architecture from the Sharif University of Technology, in 2005, and the Ph.D. degree in computer system architecture from Shahid Beheshti University, in 2010. She also performed studies on computer arithmetic as a Postdoctoral Researcher with the Computer Engineering Laboratory, Delft University of Technology. She joined the Department of Electrical Engineering, Shahid Beheshti University, as an Assistant Professor and was promoted to Associate Professor. She is currently a Lecturer in computer science with the University of Hertfordshire. She has authored or coauthored more than 50 publications in journals and conference proceedings. Her research interests include computer arithmetic, residue and redundant number systems, low-power digital circuits for signal processing, cryptography and IoT applications, and VLSI design.

**MAHMOOD FAZLALI** received the Ph.D. degree in computer architecture from Shahid Beheshti University (SBU), in 2010. He performed postdoctoral research on reconfigurable computing systems with the Computer Engineering Laboratory, Delft University of Technology (TU Delft), till 2012. He joined the Department of Data and Computer Sciences, SBU, as an Assistant Professor. He is currently a Lecturer in computer science with the University of Hertfordshire. He published more than 40 papers in reputable journals and scientific conferences, especially on high-performance computing. His research interests include high-performance computing, parallel processing, and big data processing. He is an Associate Editor of *Array* (Elsevier) as well as a Reviewer of several IEEE, ACM Transactions, and Elsevier and Springer journals.