Dynamic Optimization of On-Chip Memories for HLS Targeting Many-Accelerator Platforms

Argyris Kokkinis, *Student Member, IEEE*, Dionysios Diamantopoulos, *Member, IEEE*, and Kostas Siozios, *Senior Member, IEEE*

Abstract—Many-accelerator platforms have been introduced for maximizing FPGA's throughput. However, as the high saturation rate of the FPGA's on-chip memories limits the number of synthesized accelerators, frameworks for Dynamic Memory Management (DMM) that allow the synthesized designs to allocate/de-allocate on-chip memory resources during run-time have been suggested. Although, those frameworks manage to increase the accelerators' density by minimizing the utilized memory resources, the parallel execution of many-accelerators may cause severe memory fragmentation and thus memory allocation failures. In this work, a framework that optimizes the memory usage by performing memory defragmentation operations in HLS many-accelerator architectures that share on-chip memories is proposed. Experimental results highlight the effectiveness of the proposed solution to eliminate memory allocation failures due to memory fragmentation, reduce memory allocation failures up to 32% on average and decrease the memory size requirements up to 5% with controllable latency and resource utilization overhead.

Index Terms—FPGA, high-level synthesis, dynamic memory management, dynamic memory optimization, memory defragmentation

1 Introduction

Field-Programmable Gate Array (FPGA) in conjunction to the many-accelerator architectural template, has been suggested as a candidate solution towards achieving efficient task-level parallelism in the endeavor of maximizing performance per watt ratio [1]. In the past, The programming model of an FPGA was entirely based on register-transfer level descriptions instead of C/C++. Although the traditional design flow delivers performance and power optimized solutions, the recent shift in the computing paradigm that aims to shrink the project's development cycle without sacrificing the design's performance has led to the adoption of High-Level Synthesis (HLS) [2]. As many-accelerator systems rely heavily on the use of on-chip memories (called Block RAMS, or BRAMs[1]), their rapid starvation is a bottleneck to the design of circuits that will fully exploit the device's computational resources and acceleration capabilities, creating a memory-induced *Dark Silicon* [3].

Dynamic memory management (DMM) techniques for HLS have recently proposed that allow accelerators to allocate/de-allocate on-chip memory at run-time. These solutions rely on creating heaps of BRAMs that are shared among multiple accelerators. Although, a shared memory space undermines the accelerators' performance, it has the potential to increase their density up to 3.8× [3].

1. Xilinx BRAMs refers to on-chip dual port memories of 18Kbits.

- *Argyris Kokkinis and Kostas Siozios are with Physics, Aristotle University of Thessaloniki, 54124 Thessaloniki, Thessaloniki, Greece. E-mail: {arkokkin, ksiop}@auth.gr.*
- *Dionysios Diamantopoulos is with IBM Zurich Research Laboratory, 8803 Rüschlikon, Switzerland. E-mail: did@zurich.ibm.com.*

Manuscript received 20 June 2022; accepted 6 July 2022. Date of publication 12 July 2022; date of current version 28 July 2022.
This work was supported by E.C. funded program SERRANO under H2020 Grant 101017168. (Corresponding author: Argyris Kokkinis.)
Digital Object Identifier no. 10.1109/LCA.2022.3190048

Previous studies indicated that a limited number of heaps and memory allocators should be synthesized on the FPGA, and these memory hierarchies should be shared among the accelerators [3], [4]. Nevertheless, when multiple accelerators are executed in parallel, multiple memory requests of different allocation sizes are served by the same allocator, which may in turn lead to severe memory fragmentation and thus under-utilization of the BRAM resources [3]. The fragmentation of the shared memory space depends on the application's memory patterns and on their allocation sizes [3]. As the heterogeneity of the allocation sizes increases, it is more likely for a dynamic memory allocation request to fail due to lack of available contiguous free memory blocks. For the rest of this manuscript, the different memory allocation sizes that appear on a heap will be referred to as Distinct Allocation Sizes (DAS).

The existing DMM frameworks for FPGAs [3], [4], [5], [6] emphasize on memory efficiency by selecting appropriate memory allocation schemes, but they do not provide mechanisms for dynamic memory optimization. In [5] a dynamic memory manager for FPGAs was proposed that allows allocation/de-allocation of BRAMs at run-time. BRAMs are grouped in logical pages but the fixed size allocation mechanism may lead to severe fragmentation. Liang *et al.* proposed Hi-DMM; a framework that uses a buddy system allocator for managing BRAMs [4]. Although this approach minimizes external fragmentation, it may cause memory under-utilization due to internal fragmentation [7], [8]. A framework for the automated transformation of C++ code into synthesizable HLS using the same buddy allocator is discussed in [9]. However, such allocation schemes are not considered suitable for memory sharing among workloads of heterogeneous allocation sizes due to internal fragmentation [8]. *Giamblanco et al.* discuss a library of HLS allocators that balance performance versus memory efficiency in [6]. The `gnumem` allocator, which performs memory block coalescing in an attempt to reduce fragmentation is also introduced in [6]. However, memory coalescing is not always applicable as it requires adjacent memory blocks to be unoccupied.

Throughout this manuscript we propose an HLS framework for dynamic optimization of the on-chip memory blocks that are used for dynamic memory allocation in many-accelerator systems. This work extends our former *MemLuv* implementation [3], the only available HLS DMM framework targeting many-accelerators mapped onto FPGA platforms, to support on-demand memory defragmentation.

Experimental results highlight the efficiency of the proposed solution, as it reduces significantly the Memory Allocation Failures (MAF) due to memory fragmentation. In detail, the proposed framework reduces overall MAF (due to memory fragmentation) up to 32% on average and decreases memory requirements (`i.e.` heap size) up to 5% for the same failure probability as compared to our former public-available MemLuv DMM framework [3].

2 Proposed Framework

This section describes the proposed memory optimization framework depicted in Fig. 1. This framework consists of two phases, namely the *offline* analysis and the *online* execution. Throughout this analysis two parameters that are critical for the design's implementation are discussed; *(i) fragmentation ratio:* this refers to the percentage that a heap is fragmented during the accelerators' execution and *(ii) activation threshold* (Θ)*:* a designer-defined value that determines the threshold at the *fragmentation ratio* that triggers the execution of the defragmentation task that is performed by a garbage collector.

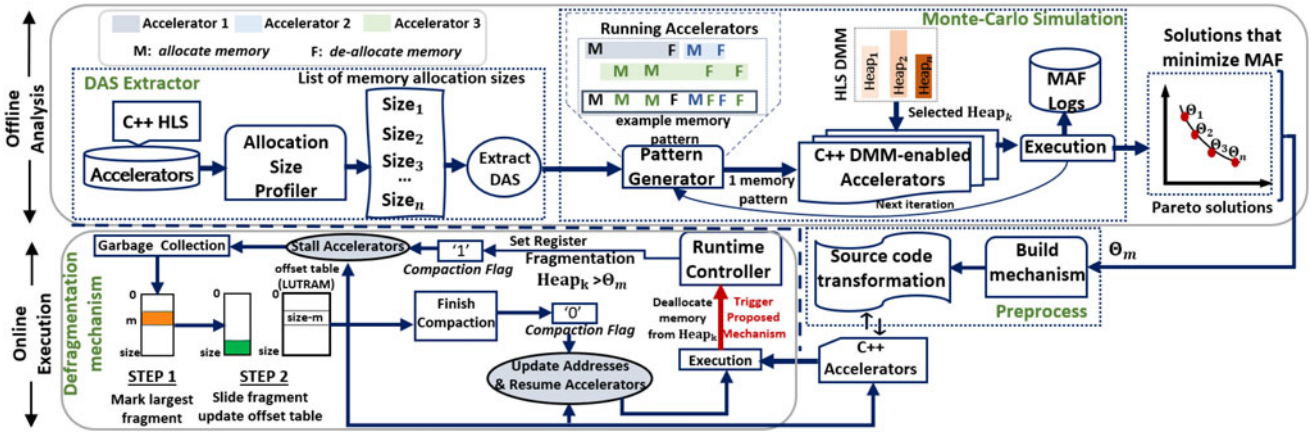

Fig. 1. Proposed framework for dynamic memory optimization.

2.1 Offline Analysis Phase

The analysis phase aims to determine the parameter Θ that enables the garbage collector to perform memory defragmentation. Initially, the memory allocation sizes from the HLS accelerators are extracted and their different values form the DAS. Then, a Monte-Carlo simulation analysis is performed to determine all the Θ that minimize MAF. The input to this simulation is memory patterns that correspond to an overlapping execution of those accelerators, as they are derived from the *pattern generator* block. This block pseudo-randomly generates `malloc/free` sequences of the extracted DAS, forming multiple memory patterns. Those are patterns for on-chip memory allocation (`malloc`) and de-allocation (`free`), as they are introduced and used in the DMM framework [3]. Based on the generated memory pattern, two C++ application instances are automatically instantiated, where only one of them uses the proposed memory optimization framework. Regarding the scope of this analysis, the functionality of HLS DMM for a given heap size $Heap_k$ is implemented with our former MemLuv framework [3].

The offline analysis is performed with 10,000 different memory patterns that are generated during the Monte-Carlo simulation and executed in an algorithmic simulation-based level. The percentage of MAF for varied activation thresholds Θ ranging from 0% to 100% are reported and compared between the two application instances. Based on this analysis, the Pareto curve that trades-off decrease in MAF with performance metrics is calculated. Note that lower Θ values impose frequent defragmentations, which in turn increases the application's execution latency. Hence, proper selection of Θ_m value has to be performed in order to minimize MAF and satisfy the application's requirements.

2.2 Execution Phase

During application's execution, the run-time controller monitors heap's $Heap_k$ *fragmentation ratio* and activates the garbage collector whenever it exceeds the Θ_m threshold. The proposed framework consists of three sub-components, namely: i) a distributed RAM structure, ii) an 1-bit flag register, and iii) the garbage collection mechanism depicted at lower-left part of Fig. 1.

The distributed RAM structure (called `offset table`) is implemented in Look-Up Tables (LUTs) and is used for updating the addresses of the relocated memory objects. HLS tools, such as Xilinx Vitis HLS and LegUp [10], support a single address space and therefore designs with pointer-to-pointer arithmetic are not synthesizable. Hence, every object in the `offset table` is linked to a heap memory element, indicating the number of positions that each live[2] object in the memory has been relocated. The flag register (called `compaction flag`) is a 1-bit wide register that is used for blocking the accelerators from performing memory access operations during defragmentation task. Due to the execution order unpredictability of parallel accelerators, the defragmentation algorithm may be triggered at any time.

Memory Optimization Algorithm.

The heap defragmentation task is performed based on a Mark-Compact garbage collection [11] algorithm. At the marking phase, the heap is linearly traversed and the live objects are marked by setting high the bits of a heap-size register; thus, the largest fragmented memory region is identified. Then, during the compaction phase, the garbage collector reallocates the utilized memory blocks to the bottom of the heap (by shifting the live memory objects at the bottom heap addresses), resulting the defragmentation of the heap's top. During the algorithm's execution the elements of the `offset table` that are linked to the heap's live memory objects are updated, keeping track of the positions that the allocated memory blocks have been relocated.

Source Code Modifications. To perform the described memory optimization operations, the MemLuv framework is enriched with two functions (and their associated APIs) that allow the accelerators to update the addresses of the relocated memory objects in the event of memory compaction:

```
void* Update(void* address, uint_t nbytes, uint_t heap_ID)
uint8_t CheckFlag(uint_t heap_ID)
```

, where the `checkFlag()` function returns the content of the `compaction flag` register for the corresponding heap (`heap_ID`), whereas the `Update()` function reassigns the address of the object that was previously stored in different memory region. At the back-end, the `Update()` function performs a linear search at the heap's `offset table` and calculates the object's new memory position by adding its previously assigned address with the sliding factor from the offset table. The stalls at the accelerators' execution are performed with a block of code before the DMM access, which checks the heaps' status (through the `checkFlag()` function).

3 Evaluation Results

The proposed mechanism was developed as an open-source solution[3] and evaluated on Xilinx Vitis HLS 2020.1 with Alveo U200 FPGA acceleration card. The evaluation procedure was performed based on [6], where memory allocation patterns are classified into three categories: i) *square*, ii) *triangular*, and iii) *random*. More

2. As *live* we consider the already allocated memory objects.

3. https://github.com/ArgyKokk/MC-DMM-Analysis

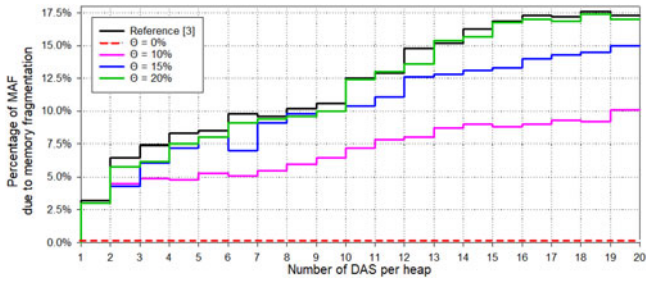

Fig. 2. Evaluation of MAF due to memory fragmentation.

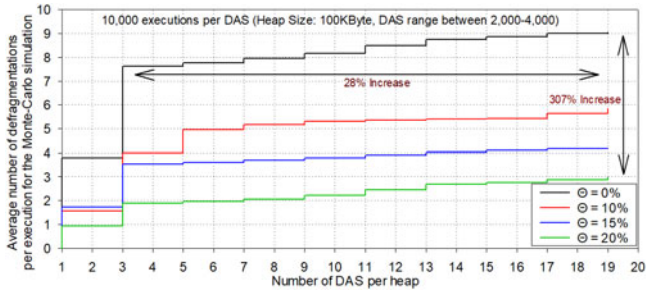

Fig. 3. Average executions of the proposed defragmentation algorithm for different DAS and activation thresholds.

precisely, in many-accelerator domain square memory patterns typically correspond to sequentially executed accelerators. In this case, the heap fragmentation is zero, and hence there is no benefit from the proposed memory optimization mechanism. On the other hand, random memory patterns correspond to allocation/de-allocation sequences that would appear during parallel execution of many accelerators, while triangular memory patterns can be considered as a subcase of the random patterns at our many-accelerator scenario. Hence, *random* memory patterns are considered in this work.

For evaluation purposes K-means accelerators with a dynamic allocation size ranging between 2,000 to 4,000 (8–16 KBytes) from the publicly available library of accelerators `Vineyard`, were used [12]. Noted, that the proposed solution is agnostic of the accelerators' type, only the DAS of those accelerators are required for the offline heap fragmentation analysis phase.[4] Without affecting the general applicability of the proposed framework, the dynamic memory management (`malloc()` and `free()` requests) was performed using the *MemLuv* DMM [3].

Initially, we explore the efficiency of the proposed mechanism to minimize the MAF metric. Fig. 2 summarizes the results of this analysis for activation thresholds Θ ranging from 0% to 20%. The reference solution for this analysis (depicted with black color line) refers to the *MemLuv* DMM [3]. According to this analysis, as we increase the threshold's Θ value (`i.e.`, execute the memory defragmentation task more rarely), the MAF also increase towards the reference solution without the proposed framework. On the other hand, lower Θ values result to higher probability for concurrent execution of multiple accelerators without MAF. In case we have a Θ=0%, then the defragmentation algorithm is executed after a `free()` command (`i.e` de-allocation of memory blocks) causes a heap's *fragmentation ratio* higher than zero. Although this selection maximizes memory efficiency, the frequent memory compaction imposes an overhead to the accelerators' latency due to the execution stalls, which has to be concerned during the implementation phase.

4. For instance, the DAS=2 execution scenario refers to the pseudo-random selection of 2 K-means accelerators (within the defined allocation range) per Monte-Carlo iteration.

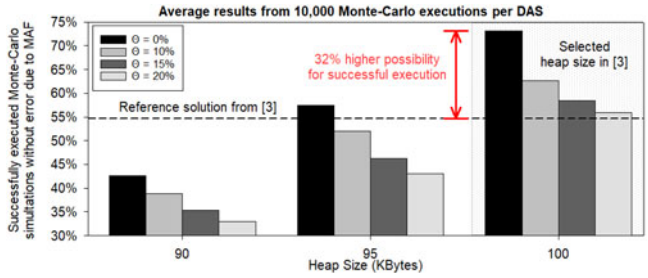

Fig. 4. Successful executions at different heap sizes and activation thresholds for 20 DAS.

TABLE 1
Overhead of the Defragmentation Mechanism

Heap Size	LUTs	FFs	Min.Latency	Avg.Latency	Max.Latency
2 KB	2,977	1,566	40 us	56 us	73 us
4 KB	3,232	1,663	79 us	88 us	0.14 ms
32 KB	6,345	1,787	0.64 ms	0.94 ms	1.17 ms
100 KB	25,345	2,467	1.98 ms	2.93 ms	5.18 ms
120 KB	29,654	2,843	2.34 ms	3.27 ms	5.47 ms

In order to study more thoroughly this topic, we evaluate the number of executions of the defragmentation garbage collection mechanism and therefore the frequency of the execution stalls. More specifically, Fig. 3 plots the activation frequency of the garbage collector for different Θ thresholds as a function of the number of DAS per heap. Based on this, the number of times that the defragmentation mechanism is activated for a given Θ is almost agnostic to the number of DAS per heap (assuming that we have at least 2 DAS per heap). This is caused since the number of executions depends mainly on its activation threshold Θ. An increase at the number of DAS may rise the activation frequency of the defragmentation mechanism up to 28% ($\Theta = 0\%$ plot in Fig. 3). However this increase in negligible in comparison to the 307% rise at the number of the mechanism's executions that appears between the $\Theta = 0\%$ and $\Theta = 20\%$ plots.

Next, we explore the efficiency of the proposed framework at different heap sizes. For this purpose, three different heap sizes (90, 95 and 100KBytes) and four activation thresholds are considered for the K-means use-case scenario. Fig. 4 reports the percentage of the Monte-Carlo executions for the evaluated use-case where fragmentation MAF do not occur, regarding 20 DAS (this is the maximum number of unique allocation sizes per application considered throughout this analysis). For demonstration purposes we also plot the reference solution without memory defragmentation, as it is retrieved with the *MemLuv* DMM [3] for a heap size of 100KBytes.

Based on this analysis, all the activation thresholds Θ for a heap size of 100KBytes achieve higher percentage of successfully executed Monte-Carlo simulations as compared to the reference solution. More specifically, the $\Theta = 0\%$ configuration results to an average of 32% higher possibility for accelerators execution without MAF. In addition to that, the $\Theta = 0\%$ threshold results to the maximum ratio for the Monte-Carlo analysis, even for the 95KBytes heap size. This is due to the complete memory defragmentation that is performed with the proposed framework. Hence, in this case a decrease of up to 5% in the heap size can be achieved without affecting the probability of inducing MAF. By decreasing the heap size even further this percentage drops due to the shrinked memory space which causes an increase in the MAF. Regarding the $\Theta = 10\%$ to $\Theta = 20\%$ thresholds the percentages for the same heap configurations are lower due to the partial defrgamentation which is an immediate effect of the less frequent activation of the garbage collector.

TABLE 2
Performance Evaluation of 20-Kmeans

Device	Computational Platforms				
	Intel CPU	Nvidia GPU[5]	Alveo U200 FPGA		
	Xeon Gold 5218R	Tesla T4	Static	DMM [3]	Proposed
Energy[6]	3.3 J	5.2 J	does	2.2 J	2.7J
Frequency	2.1 GHz	585 MHz	not	250 MHz	250MHz
Fragm. MAF	N/A	N/A	fit	$\leq 17.5\%$	0

Finally, we provide the area overhead (at Alveo U200) due to the implementation of the proposed framework for different heap sizes, as well as the minimum, maximum and average induced latency (at 250 MHz) of one execution stall (i.e one execution of the garbage collector). The results of this analysis for different heap sizes are summarized in Table 1. In detail, as the heap's size is increased, the utilized resources and the latency also rise, as more LUTs and FFs are required for the implementation of the `offset table` structure and more clock cycles are needed for the execution of the memory compaction algorithm. Additionally, The garbage collector's maximum synthesizable clock frequency is 360 MHz on Alveo U200; therefore accelerators that operate at frequencies below the 360 MHz are not affected by the proposed framework.

For comparison purposes, Table 2 reports the energy consumption, clock frequency and the MAF due to memory fragmentation in different platforms (CPU, GPU, FPGA) regarding the evaluated K-means scenario. For this analysis a $\Theta = 0\%$ is used in the proposed design, which leads to more executions of the garbage collector and therefore increases the consumed energy. However, even in this case there is a 33% and 57% decrease at the consumed energy compared to a CPU and a GPU execution respectively while the MAF due to fragmentation are zero.

In our previous DMM framework [3] if a MAF occurred then the affected accelerator would stall its execution until enough memory blocks are de-allocated by the other executed accelerators [3]. As a result, a MAF causes an unpredictable increase at the accelerators' latency. Therefore, based on Table 2 the 2.2 J that are consumed by the accelerators in framework [3] may increase unpredictably in case of a MAF.

4 Conclusion

A novel framework to support dynamic memory optimization for HLS, was introduced. The proposed solution performs on-demand memory defragmentation on heaps implemented in BRAMs. Experimental results showcase the effectiveness of the proposed mechanism to minimize MAF due to memory fragmentation for a controllable latency and area overhead. Additionally, we achieve to reduce heap's size requirements without affecting the probability of allocation failures.

References

[1] G. Tagliavini, D. Cesarini, and A. Marongiu, "Unleashing fine-grained parallelism on embedded many-core accelerators with lightweight OpenMP tasking," *IEEE Trans. Parallel Distrib. Syst.*, vol. 29, no. 9, pp. 2150–2163, Sep. 2018.

[2] M. W. Numan, B. J. Phillips, G. S. Puddy, and K. Falkner, "Towards automatic high-level code deployment on reconfigurable platforms: A survey of high-level synthesis tools and toolchains," *IEEE Access*, vol. 8, pp. 174 692–174 722, 2020.

[3] 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.–Dec. 2015.

[4] 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 Des. Integr.*, vol. 37, no. 11, pp. 2555–2566, Nov. 2018.

[5] G. Dessouky, M. J. Klaiber, D. G. Bailey, and S. Simon *et al.*, "Adaptive dynamic on-chip memory management for FPGA-based reconfigurable architectures," in *Proc. 24th Int. Conf. Field Programmable Log. Appl.*, 2014, pp. 1–8.

[6] N. V. Giamblanco and J. H. Anderson, "A dynamic memory allocation library for high-level synthesis," in *Proc. 29th Int. Conf. Field Programmable Log. Appl.*, 2019, pp. 314–320.

[7] J. L. Peterson and T. A. Norman, "Buddy systems," *Commun. ACM*, vol. 20, no. 6, pp. 421–431, Jun. 1977.

[8] P. R. Wilson *et al.*, "Dynamic storage allocation: A survey and critical review," in *Proc. Int. Workshop Memory Manage.*, 1995, pp. 1–116.

[9] J. Lau, A. Sivaraman, Q. Zhang, M. A. Gulzar, J. Cong, and M. Kim, "HeteroRefactor: Refactoring for heterogeneous computing with FPGA," in *Proc. IEEE/ACM 42nd Int. Conf. Softw. Eng.*, 2020, pp. 493–505.

[10] A. Canis *et al.*, "LegUp: High-level synthesis for FPGA-based processor/accelerator systems," in *Proc. 19th ACM/SIGDA Int. Symp. Field Programmable Gate Arrays*, 2011, pp. 33–36.

[11] R. Jones *et al.*, *The Garbage Collection Handbook: The Art of Automatic Memory Management*. London, U.K.: Chapman & Hall/CRC, 2011.

[12] E. Koromilas, C. Kachris, D. Soudris, F. J. Ballesteros, P. Martinez, and R. Jimenez-Peris, "Modular FPGA acceleration of data analytics in heterogenous computing," in *Proc. Int. Conf. Des. Autom. Test Eur. Conf. Exhibit.*, 2019, pp. 626–629.