
٥٥٥٥٥



**TECHNISCHE
UNIVERSITÄT
DRESDEN**

Faculty of Electrical and Computer Engineering Institute of Computer Engineering

Chair for Processor Design

Amruth Balachandran

Born on: 10th June 1997 in Salem (India)

Course: Nanoelectronic Systems

Matriculation number: 4988346

Matriculation year: 2020

A Non-Contiguous Memory Space Dynamic Memory Management unit for FPGAs

to achieve the academic degree

Master of Science (M.Sc.)

Supervising professors

Prof. Dr. Akash Kumar

Prof. Dr.-Ing. Diana Göhringer

Submitted on: 23rd August 2024



Task for the preparation of a Master Thesis

Course: Nanoelectronic Systems
Name: Amruth Balachandran
Matriculation number: 4988346
Title: A Non-Contiguous Memory Space Dynamic Memory Management unit for FPGAs

Objectives of work

In recent years, with the introduction of 'High-Level Synthesis' (HLS) tools such as 'Vivado-HLS' and 'Catapult' along with 'Dynamic Partial Reconfiguration' (DPR) has enabled multi-accelerator (reconfigurable) designs to take advantage of the limited resources on a FPGA. The trend towards multi-accelerator heterogeneous systems is fueled by the paradigm shift towards more aggressive 'Hardware/Software Co-design' solutions, to overcome the utilization/power-wall. One main challenge of multi-accelerator designs on FPGAs is the efficient utilization of limited resources such as 'Block Random Access Memory' (BRAM), which is often a bottleneck in the design of embedded systems. Modern FPGA EDA tools (both RTL and HLS) only allow static memory allocation, which causes pessimistic allocation of BRAM modules for any accelerator architecture's entire execution window. Static memory allocation is inefficient for applications with variable memory workload, imposing excessive area and power consumption overheads. Hence this work aims to implement a Dynamic Memory Management (DMM) unit in hardware to facilitate dynamic allocation of memory, with minimal internal fragmentation.


Focus of work

- Study the existing DMM unit designs and identify the internal fragmentation generated.
- Design a custom DMM architecture to reduce internal fragmentation of memory, with improved flexibility and scalability to support multi-accelerator systems.
- Create a structured testbench to verify and measure performance.
- Synthesis and Implementation of the custom DMM unit design (packaged as an IP) on a small-scale target FPGA based on ZYNQ 7020 chipset.
- Optional: Measure the contribution to throughput made by the custom DMM unit in a variable workload environment.

1st Examiner: Prof. Dr. Akash Kumar
2nd Examiner: Prof. Dr.-Ing. Diana Göhringer
Issued on: March 15, 2024
Due date for submission: August 23, 2024

Thomas
Mikolajick
Prof. Dr.-Ing. Thomas Mikolajick
Chairman of Examination Board

Digital unterschrieben
von Thomas Mikolajick
Datum: 2024.03.12
10:10:52 +01'00'

 Digitally signed by
Akash Kumar
Date: 2024.02.07
15:42:10 +01'00'
Prof. Dr. Akash Kumar
Supervising professor



Statement of authorship

I hereby certify that I have authored this A Non-Contiguous Memory Space Dynamic Memory Management unit for FPGAs independently and without undue assistance from third parties. No other than the resources and references indicated in this thesis have been used. I have marked both literal and accordingly adopted quotations as such. During the preparation of this thesis I was only supported by the following persons:

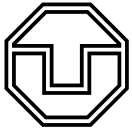
Prof. Dr. Akash Kumar

Prof. Dr.-Ing. Diana Goehringer

Additional persons were not involved in the intellectual preparation of the present thesis. I am aware that violations of this declaration may lead to subsequent withdrawal of the degree.

Dresden, 23rd August 2024

Amruth Balachandran



Abstract

This work proposes *VDMMU* (Virtual Dynamic Memory Management Unit), a novel *Dynamic Memory Allocation* (DMA) architecture for FPGA-based accelerators that aims to improve memory resource utilization and system throughput. The introduction of *High-Level-Synthesis* (HLS) tools has made realization of software algorithms in hardware (accelerators) quicker. Software algorithms often rely on dynamic memory management (DMA) to handle variable workloads efficiently, but HLS tools use pessimistic static memory allocation, leading to under-utilization of statically allocated memory resources. This creates a bottleneck in multi-accelerator systems due to quick exhaustion of on-chip memory like BRAMs with respect to other resources such as LUTs.

VDMMU architecture implements dynamic memory management for FPGAs to better utilize BRAMs across different accelerator workloads, while eliminating *external fragmentation* of the heap and providing scalable support for concurrent heap access in multi-accelerator systems. The novelty of VDMMU is in using *Benes Interconnect* to bidge the accelerators with the heap memory to support scalable parallel access, and the utilization of a *Reverse/Inverse Butterfly* network to maintain the page-table (virtual memory), resulting in upto 20% better heap memory utilization compared to architecture implementing a standard *Buddy-System*. When implemented on the PYNQ-Z2 dev board housing a *XC7Z020 SoC*, the VDMMU operating at 100MHz , achieves upto $4\times$ reduction in LUT consumption compared to an implementation using a *Complete Binary Tree* interconnect while providing 64 independent access channels to the heap. Where each individual channel is capable of *reading and(or) writing* to an address in heap. Furthermore, VDMMU exhibits an absolute allocation and deallocation latency of 2 and 3 cycles, respectively.

Contents

List of Figures	III
List of Tables	VI
List of Algorithms	VII
1 Introduction	1
1.1 Producer-Consumer Paradigm	1
1.2 Pipelining Paradigm	2
1.3 Transaction-Level Parallelism	2
1.4 Streaming Applications	3
1.4.1 Sparse-Sparse General Matrix Multiplication	3
1.4.2 Connected Components Labelling	3
1.4.3 Convolution	4
1.5 Motivation for Dynamic Memory Management	5
2 Background	7
2.1 Field-Programmable-Gate-Array	7
2.1.1 Look-Up-Table	7
2.1.2 LUT-RAM	8
2.1.3 Block-Random-Access-Memory	8
2.2 Static BRAM Allocation by HLS Tools	9
2.3 Dynamic Memory Allocation in Hardware	10
2.3.1 Dynamic Allocation Concept, Policies and Strategy	10
2.3.2 Heap Memory and Dynamic-Memory-Management Unit	11
2.3.3 Allocation and De-allocation Requests	11
2.3.4 Internal Memory Fragmentation	11
2.3.5 External Memory Fragmentation	11
2.3.6 First-Fit	11
2.3.7 Best-Fit	11
2.3.8 Next-Fit	12
2.3.9 Worst-Fit	12
2.3.10 Buddy-System	12
2.3.11 Bit-Mapped Fit	12
3 Related Work	13
3.1 Dynamic Memory Management Units	13
3.1.1 Modified Buddy System	13

3.1.2	SysAlloc	14
3.1.3	OLDMA and TLDMA	15
3.1.4	Free-List-Manager	15
3.1.5	DMM-HLS	16
3.1.6	DOMMU	17
3.2	Bottle-necks	17
4	Design Space Exploration	19
4.1	Optimization Parameters	19
4.2	Impact of Heap Memory Compactness	20
4.2.1	Python Testbench	21
4.2.2	Packaging Efficiency Measures	22
4.3	Virtual Dynamic Memory Management	23
4.4	Support for Parallel Memory Access	24
4.4.1	Interconnect	24
4.4.2	Complete Binary Tree	26
4.4.3	2D-Crossbar Array	26
4.4.4	Benes Network	27
5	Implementation	29
5.1	Heap Manager	29
5.1.1	Reverse Butterfly Structure	31
5.1.2	Access-Points Base Register	33
5.2	Interconnect	34
5.3	Mapper	35
5.3.1	Parallel Routing Algorithm	35
5.3.2	Chain Representation	36
5.3.3	Control Architecture	38
5.4	Heap and Translator	40
5.5	Optimization	41
5.6	Operation Modes	42
5.6.1	Stream-Port	42
5.6.2	Dual-Port	43
5.7	Parameterization	43
6	Results and Evaluation	45
6.1	Heap Manager Cost-Performance	45
6.2	Allocation and Deallocation Latency	46
6.3	Concurrent Heap Access Cost	47
6.4	VDMMU Resource Utilization	47
6.5	Evaluation Testbench	48
7	Conclusion and Future Work	53
7.1	Conclusion	53
7.2	Future Work	54
	Bibliography	55

List of Figures

1.1	Components (tasks) structured as a Kahn process network (KPN) using First-In-First-Outs (FIFOs). Task 3 consumes from both Tasks 1 and 2, while Task 4 consumes data produced by 3. [2]	1
1.2	Tasks A, B and C when un-piplined are executed sequentially. By pipelining the schedule a higher throughput is achieved [2].	2
1.3	Pipelining of runs with each tasks within a run represented as KPNs [2].	2
1.4	(a) Row-wise product computation of two matrices A and B. (b) Parallelism of row-wise product using multiple functional units to compute products for each row of matrix A simultaneously [34].	3
1.5	(a) The main image with objects of interest. (b) Threshold image used for Connected Component Labelling (CCL). (c) Grayscale result image after CCL. [29]	4
1.6	Architecture of a 3x3 Convolutional Neural Network (CNN) accelerator [11]. . .	5
1.7	Memory bottle-neck of Kmeans clustering algorithm. (a) Workload scalability. (b) Accelerator-Scalability. [10]	5
2.1	A Look-Up-Table (LUT)_3 realized using two LUT_2, where a and b are the two inputs of the boolean equation, while c is grounded to select the output of the bottom LUT_2. The outputs f_1 and f_2 are registered using the output Flip-Flops (FFs).	8
2.2	Internal structure of a LUT_4 with 4 inputs and 16 Static-Random-Access-Memory (SRAM) cells [5].	9
2.3	(a) Heap memory with free Minimum-Allocation-Units (MAUs). (b) Heap state after four allocations A, B, C and D (shown by coloured squares). (c) Heap state after B and D are de-allocated. (d) Visualisation of Internal and External fragmentation.	10
3.1	A visualisation of the Complete Binary Tree (CBT) ($N = 8$) with leaf nodes (square) occupied by MAUs, represented by a bitmap (1 = full; 0 = free), and each node (circle) in the tree representing AND or OR gates. The parallel search and address translation progresses in $O(\log_2[N])$ time due to $l = \log_2(N)$ levels of the CBT.	14
3.2	Representation of parallel memory access configuration for multi-accelerator system in [10]. Each sub-system (using Dynamic-Memory-Allocation (DMA)) within a cluster acquires a sequential memory access latency.	16
3.3	Representation of parallel memory access configuration for multi-accelerator system in [9]. Each sub-system (using DMA) within a cluster acquires a sequential memory access latency, but the each cluster is allocated on dedicated Block-Random-Access-Memorys (BRAMs).	17

4.1	Functional representation of the python testbench used to measure, packing efficiency and MAU utilization at allocation failure.	21
4.2	Visualisation of the heap's status pre and post deallocation of 'Var C' which occupies 2 MAUs in heap. Shown is the external fragmentation caused by Buddy-System because it lacks implicit coalescing, and the behaviour of an ideal heap management scheme with implicit coalescing (ideal heap utilization).	21
4.3	(a) Visualization of bottleneck caused due to external fragmentation; Dark columns present the available size in the heap, while light columns present the requested size. (b) Process completion measured in-terms of the total number of allocations performed (higher is better). (c) Measure of a heap's compactness with varying heap sizes.	22
4.4	(a) Visual representation of the conversion of a physical address space P_x into a virtual space V_x by a paging system applied on top of the Buddy-System. (b) Initial state of virtual and physical maps. (c) Post allocation of $V_4 \rightarrow P_2$. (d) Post deallocation of P_0 , the virtual map is re-applied to obtain a new state. The virtual space of (b) and (d) are identical but differ in physical maps.	23
4.5	(a) Representation of a CBT interconnect where each node is a 2×1 Multiplexer (MUX) with the root node as input source and leaf nodes as output destination. (b) 2D-Array interconnect with 4 Access-Points (APs) and BRAMs.	25
4.6	Representation of a Benes interconnect switching $8 N_{AP}$ to $8 N_B$. All nodes to the right of the bisector line (B_L) follow Destination-Tag-Routing (DTR) when those to the left are setup accordingly.	27
5.1	A visualization of the <i>VDMMU</i> system architecture with four APs. The control and data path are independent of each other owing to smaller critical path delays and (de)allocation requests appear from the Processign-System (PS).	29
5.2	Paging mechanism using a <i>Page-Table</i> acting as virtual memory (logical address space) to map non-contiguous <i>Frames</i> (P_X) (physical address space) into a contiguous virtual address space (V_X).	30
5.3	Heap manager operation overview: Management of virtual memory map by <i>Request Validation</i> for allocation requests and <i>Coalescing</i> post de-allocation request.	32
5.4	Breakdown of a single <i>Virtual Memory</i> element into <i>Status</i> and <i>Frame Pointer</i> flags.	32
5.5	Shown is the coalescing of the new fragmented full/free list [2], generated from [1] by deallocation of elements V_2 and V_3 , re-grouped into an unfragmented free/full list [3] using the <i>Barrel Shifter</i> and <i>Reverse Butterfly</i> structure.	33
5.6	(a) Reverse Butterfly structure for 8 elements with even and odd nodes, each encasing a single MUX. (b) Hardware Implementation of reverse butterfly using 2×2 switches each encasing 2 MUXs.	34
5.7	(a) Benes network for $N = 8$ inputs and outputs. (b) Recursive Clos network representation of a Benes network for $N = 8$. Neighbouring packets $\pi(1)$ and $\pi(2)$ are routed through separate subnetworks U and L in the first stage to avoid collision in phase B	35
5.8	A 2×2 crossbar switch $SW_{I(O)_{nk}}$. The switch $SW_{I(O)_{nk}}$ can take the value $sch \in 0, 1$ to represent bar and cross, respectively.	36
5.9	Bi-partite graph of equation (5.2) solved by 2 edge colours for upper and lower subnetworks. If node SW_{I_1} is n_i , then its neighbours n_a are SW_{I_0} and SW_{I_2} . Each b_k, b_{k+1} value corresponds to the destination addresses of a switch's inputs $k, k + 1$	37

5.10	Graphical representation of node's <i>scb</i> relationship with its neighbours. The chain leader is the node with the smallest subscript, marked with double circles.	38
5.11	Chain C_5 with leader BPE_5 merges with C_0 upon discovery by BPE_8 . BPE_8 and BPE_3 are neighbours, whose relationship affects all the <i>scb</i> (shown in red) in (previously) C_5 's <i>BPEs</i> .	38
5.12	Architectural view of a single stage of Benes Mapper in phase A for $N = 8$. Thick lines indicate bundles/buses used to transfer stage information to all Benes-Processing-Elements (BPEs).	39
5.13	Internal view of a BPE. Thick lines indicate buses.	40
5.14	(a) AP window configured to handle a single BRAM in dual-port mode. (b) AP window configured as a sliding window to handle two BRAMs simultaneously. The numbers next to the BRAM ports indicate the physical address used by the <i>Mapper</i> to tag memory ports	41
6.1	(a) LUT consumption (primary y-axis) and total BRAM memory included in heap (secondary y-axis) against the number of MAUs managed by Heap Manager. (b) Comparison of Effective allocation latency $Eff_{alloc} @ I_{req} = 2$, for this work's Virtual Dynamic-Memory-Management Unit (VDMMU) allocator against <i>OLDMA</i> [30], <i>TLDMA</i> [30], <i>FBTA</i> [23] and <i>FLM</i> [28].	46
6.2	(a) Normalized resource reduction achieved by Benes interconnect over CBT; The dashed line represents the maximum number of channels VDMMU can support for a given number of MAU managed. (b) Inter-BRAM Translation latency of Benes interconnect against the number of MAU managed; Baseline latency marked at $CBT = 1$.	47
6.3	(a) LUT utilization breakdown of VDMMU for various number of managed MAU, with roof-line value of channels supported. Maximum operating frequency marked in callout boxes. (b) Comparison between VDMMU and DOMMU [9] LUT consumption against varying number of supported access channels, when managing 32 MAU. (c) Impact of datawidth (8bit/32bit) on VDMMU LUT utilization for different number of access channels and MAU.	48
6.4	Functionality testbench showing DUT (red box), PS-VDMMU AXI-Lite interfaces DMEM-AXIL (green box) and HEAP-AXIL (blue box), emulated PL applications (orange box) and the Programmable-Logic (PL)-VDMMU custom interface STREAM-COMBINE to connect multiple PLAs to VDMMU.	50
6.5	(a) Allocation requests from PS to VDMMU for PLA_3 , PLA_0 and PLA_1 , respectively. (b) PS writing 4 bytes of <i>0xdeaddead</i> to BRAM 8 in heap; BRAMs 8 and 9 are allocated to PLA_2 in physical memory (heap) in order to hold 8000 bytes of data since, each BRAM is configured to hold 4096 bytes. (c) PLA_2 reading 4 bytes from BRAM 8 in the heap. (d) Deallocation of PLA_3 from heap and subsequent deactivation of AP_3 in translator (blue bounding box).	51
7.1	Four Sparse Matrix Multiplication accelerators (red box) attached to VDMMU (green box) in streaming mode, controlled by PS (black box).	54

List of Tables

4.1	Comparison of Interconnect characteristics. $*Ideal$	27
5.1	Resource consumption and Switching Latency for different AP-BRAM configurations.	41
5.2	Optimized Resource consumption and Switching Latency for different AP-BRAM configurations.	42
6.1	Characteristics of various contemporary DMA architectures. Effective Allocation Latency (Eff_{alloc}) is measured at a request interval (I_{req}) of 2. * = LUT consumption from the perspective of XC7Z020 SoC; # = LUT and F_{max} adjusted to 32 MAU with 8 access channels.	49
6.2	Parameter settings for the evaluated SP mode VDMMU.	49

List of Algorithms

1	Heap Management	31
2	Benes Processing Element	44

1 Introduction

The trend of growing computational power of System-On-Chip (SoC) continues to drive the creation of newer and increasingly complex applications, but in recent years there has been an aggressive paradigm shift towards Hardware/Software Co-design (HwSw) and 'Multi-accelerator Heterogeneous Systems' to overcome the utilization/power-wall [37, 8]. Acceleration of algorithms implemented in High-Level Languages such as C++ require the designer to carefully profile the algorithm, identify bottle-necks and accelerate individual parts in hardware to obtain higher performance (throughput) while minimizing energy consumption. The profiling of the algorithm could be done in software but the acceleration part almost always requires the use of Hardware-Description-Languages (HDLs) to efficiently map software functions to hardware implementations.

High Level Synthesis (HLS) aims to streamline HwSw by elevating the abstraction of HDLs to C++/SystemC, to facilitate faster design and verification cycles. In-order to achieve good Quality-of-Results (QoR), *Vitis HLS* [2] recommends three different paradigms (broadly two) to efficiently model parallel hardware operations using sequential software operations.

1.1 Producer-Consumer Paradigm

The system is divided into functions/components from software and hardware perspectives, respectively, and modeled as KPN figure 1.1 using FIFO or Ping-Pong (PIPO) buffers. This enables the HLS tool to identify overlap between functions and schedule the execution of non-overlapping tasks in parallel.

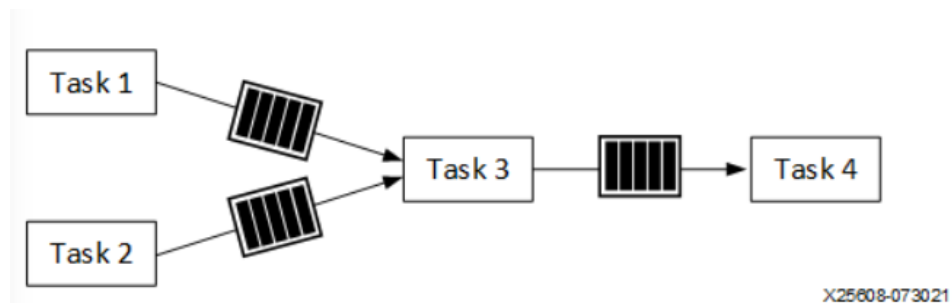


Figure 1.1: Components (tasks) structured as a KPN using FIFOs. Task 3 consumes from both Tasks 1 and 2, while Task 4 consumes data produced by 3. [2]

1.2 Pipelining Paradigm

The algorithm being accelerated is represented using functional sub-blocks, along with the control (calling order) mechanism in C++ using *for loops*. HLS then when prompted with the appropriate *# pragmas*, will unroll the loop, thus executing a sequential schedule on a parallel hardware with some latency (*Iteration-Interval (II)*). This improves the throughput as shown in figure 1.2.

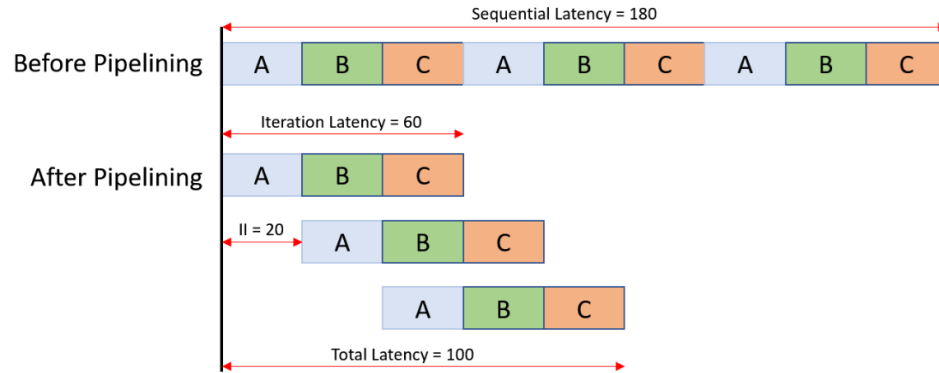


Figure 1.2: Tasks A, B and C when un-piplined are executed sequentially. By pipelining the schedule a higher throughput is achieved [2].

1.3 Transaction-Level Parallelism

Transaction-Level Parallelism (TLP) utilizes the previously discussed paradigms to achieve a schedule of the functional blocks (tasks) that reduces latency while meeting the timing and resource consumption constraints (figure 1.3). HLS tools operate on a task level to map functions to hardware, hence an object of a 'Class' in software would be realized as a functionally accurate hardware instance/component [2].

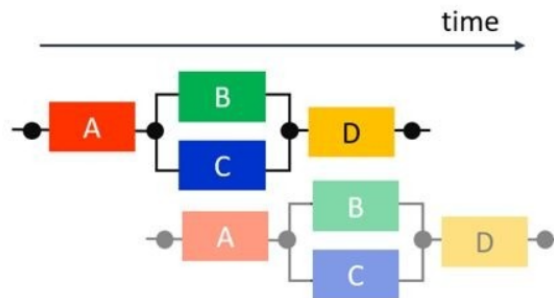


Figure 1.3: Pipelining of runs with each tasks within a run represented as KPNs [2].

Next, we shall explore some applications that can benefit from acceleration of their algorithms using HLS and how their workload can affect the resource consumption in an Field-Programmable-Gate-Array (FPGA).

1.4 Streaming Applications

Pipelining can be seen as applying scheduling between exclusive sets of (independent) functions, but HLS heavily relies on the 'Producer-Consumer' paradigm to ensure deterministic behaviour within a related set of functional blocks. The determinism of a system realized in HLS is important to achieve good QoR and take advantage of HLS's 'Hardware/Software Co-Simulation' features. Streaming applications can be modeled as KPNs ensuring determinism, hence any streaming application (modeled in software) is a good candidate for hardware acceleration using HLS. The complexity and resource utilization of the realized hardware is governed by the latency and timing requirements.

1.4.1 Sparse-Sparse General Matrix Multiplication

Sparse-Sparse General Matrix Multiplication (SpGEMM) plays a key role in graph analytics [31], recommendation networks [21] and scientific computation. The 'Gustavson's Method' (figure 1.4) is a prevalent algorithm choice for acceleration of SpGEMMs.

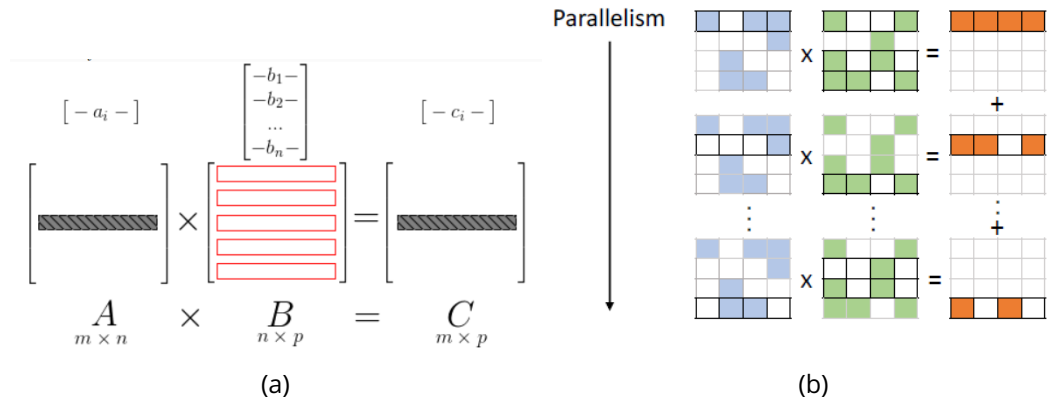


Figure 1.4: (a) Row-wise product computation of two matrices A and B. (b) Parallelism of row-wise product using multiple functional units to compute products for each row of matrix A simultaneously [34].

The algorithm may be accelerated by computing the result for each row in the resulting matrix C , in parallel as shown in figure 1.4b. A separate instance of the row-multiplier for each row in matrix C is realized in hardware and executed in parallel. In the case of SpGEMM the number of elements used by the multiplier for each result differs due to the sparsity of the matrix, suggesting at the variable workload seen by the hardware unit.

1.4.2 Connected Components Labelling

CCL examines an image to group its pixels into components based on connectivity, meaning all pixels within a connected component have similar intensity values and are connected in some manner. After identifying all the groups, each pixel is labeled with a gray level or a color

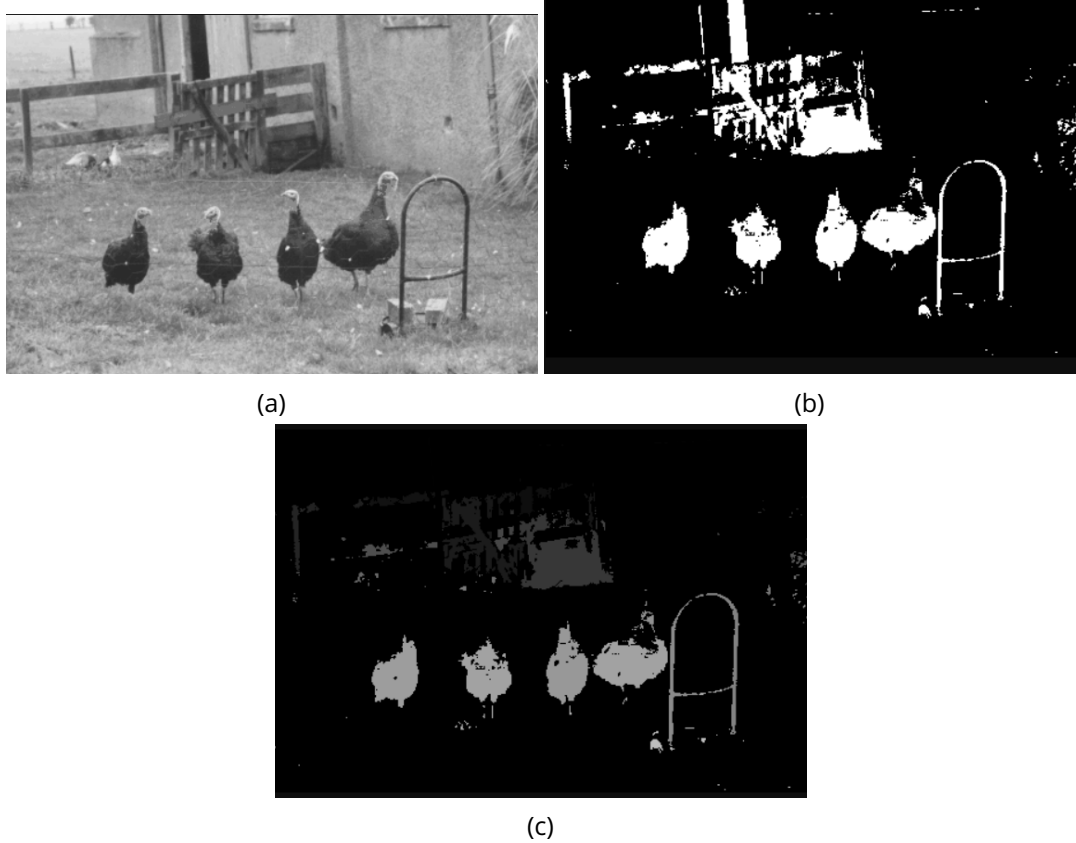


Figure 1.5: (a) The main image with objects of interest. (b) Threshold image used for CCL. (c) Grayscale result image after CCL. [29]

(color labeling) corresponding to its assigned component. This method is commonly used in image recognition and tracking.

Real-time application of CCL requires fast processing of the input image/video stream, which can be achieved using an FPGA to accelerate the algorithm [35, 20]. The execution workload of CCL varies with the image resolution and Region-Of-Interest (ROI).

1.4.3 Convolution

Convolution operations are prevalent in image processing and Neural Networks [11], where they are used to extract target features using a kernel. The process can be highly pipelined using a FPGA Digital-Signal-Processor (DSP) units to perform Multiply-Accumulate (MAC) operations in parallel as shown in figure 1.6.

The kernel weights and input/output data are fetched from external memory and stored on the FPGA's PL partition using BRAMs as buffer. In this situation, should the size of the kernel change, the requirements for the buffer will also change accordingly.

Clearly, in many of the accelerator designs, BRAMs play a pivotal role by providing a buffer zone to mask the memory latency of external memory storage elements such as Dynamic-Random-Access-Memories (DRAMs). This marks the BRAMs as an important resource that must be used efficiently.

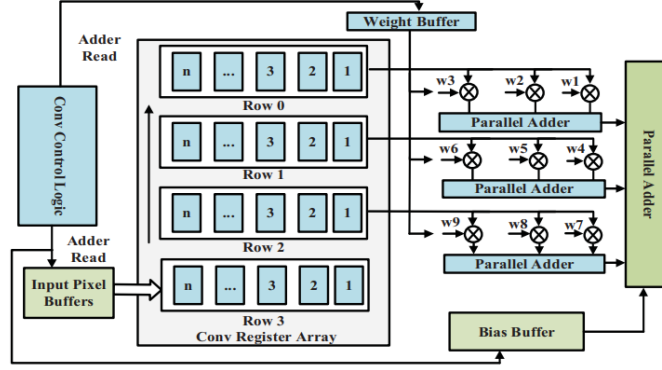


Figure 1.6: Architecture of a 3x3 CNN accelerator [11].

1.5 Motivation for Dynamic Memory Management

The memory footprint for any accelerator depends on the workload and access patterns of the algorithm being executed. Streaming applications are widely preferred due to their sequential and predictable memory access patterns, which simplifies the design complexity of an accelerator from a memory management standpoint. But, to mitigate the effect of variable workloads, which is orthogonal to memory access patterns, a *Dynamic Memory Management* approach must be adopted as done in the software counterpart (PS).

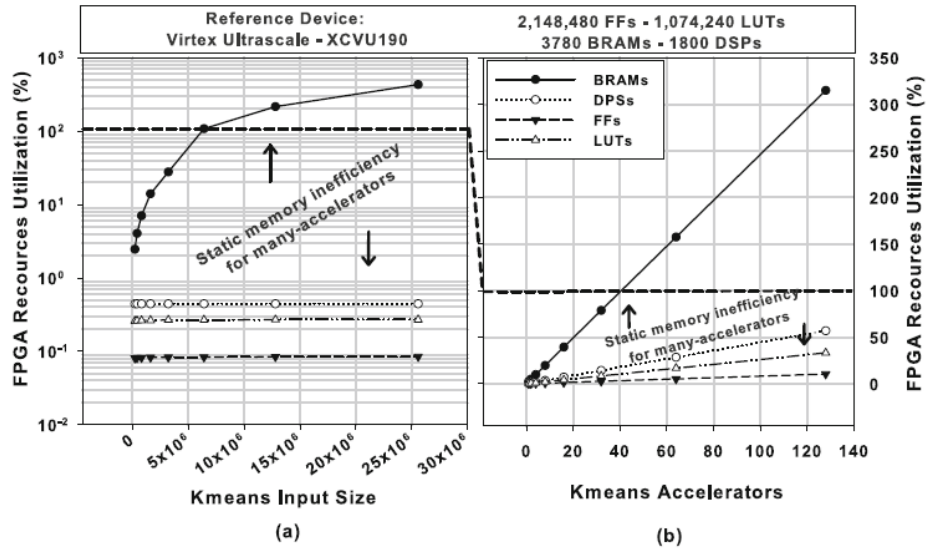


Figure 1.7: Memory bottle-neck of Kmeans clustering algorithm. (a) Workload scalability. (b) Accelerator-Scalability. [10]

Shown in figure 1.7 is a representation of how BRAM resource consumption grows with multi-accelerator systems and at the time of this work, HLS only supports '*Static Memory Allocation*' and provides no '*Dynamic Memory Allocation*'. Therefore, to improve upon BRAM utilization in multi-accelerator systems, this work explores the existing solutions for '*Dynamic BRAM Management*' and provides an hardware solution to efficiently manage small heaps of

BRAMs while supporting parallel access from multiple accelerators.

The focus of this work will be aimed towards efficient small-scale memory management (controlling smaller heaps) as opposed to many other contemporary work that target large-scale heaps of memory. The reasoning behind this is to tackle the BRAM resource utilization problem in a distributed fashion by managing multiple smaller heaps to achieve, better overall resource utilization. The goals and contributions of this thesis will be as follows:

- Implement a hardware Dynamic-Memory-Management Unit (DMMU) capable of handling small scale heaps with a maximum of 64 allocable units.
- Provide low latency *allocation*, *de-allocation*, (address) *translation* and *memory access* to all allocable units in the heap.
- Achieve a high *packaging efficiency* across all memory request patterns.
- Support parallel access to the heap to augment *multi-accelerator* systems with DMA, without introducing serial communication induced latency.
- Provide memory abstraction and DMA framework for HLS components to ease design process of streaming applications.

The rest of the work will provide supporting arguments for the small-scale distributed approach with emphasis on parallel communication, and will be organised as follows:

1. chapter 2 shall provide a brief introduction to FPGAs and their advantages. Explore the reason behind why HLS supports only *Static Memory Allocation*, the conditions for BRAM assignment when mapping C++ to software and the need for a hardware implementation of *Dynamic Memory Allocation*.
2. chapter 3 will present the contemporary approaches towards *Dynamic Memory Management* (both large-scale and small-scale) on FPGAs.
3. chapter 4 will build and support the reasoning behind design choices of this work.
4. chapter 5 explains the architecture.
5. And finally, the proposed *Dynamic Memory Management* system will be evaluated in chapter 6.

2 Background

This chapter will provide a brief introduction of FPGAs and its advantages, followed by the *Memory Structures* in contemporary FPGAs, to establish the reasoning behind why HLS supports static memory allocation. After which, a brief introduction of *Vitis's* BRAM inference conditions will be presented, followed by an introduction to dynamic allocation techniques used in software. Furthermore, the need for dynamic memory allocation in hardware will be discussed.

2.1 Field-Programmable-Gate-Array

As the name suggests, FPGA is a collection of gates that can be re-programmed (*in-field*) to realize any boolean equation. Since any digital circuit such as *Adders*, *Multipliers*, *Shifters*, etc, can be realized on an FPGA, the non-recurring engineering cost and time-to-market are significantly reduced. The process of *layout*, *fabrication* and *Verification* stages of a custom Application-Specific-Integrated-Circuit (ASIC) implementation are bypassed to produce a functional system on the FPGA within a short period of time.

The reconfigurability of an FPGA enables exact implementation of application specific hardware, instead of the conventional one-solution fits-all architecture presented by *CPUs* and *GPUs*. As a result, FPGAs exhibit higher efficiency by implementing instruction-free streaming hardware [34] or a processor overlay with customized architecture. But, this flexibility comes at the cost of maximum attainable operation frequency and increased area consumption when compared with ASIC implementation of the same architecture [5].

2.1.1 Look-Up-Table

LUTs are the crux of the FPGA allowing it to replicate any boolean logic function. Any digital circuit/block can be perceived as a collection logic gates and memory elements (FFs) connected together in a predefined order (system architecture) to perform some computation and produce deterministic results. Hence, the digital block can be represented by a collection of boolean equations. Consider the boolean equation for a 2 bit *Adder*:

$$f_1(a, b) = a \wedge b$$

$$f_2(a, b) = a \oplus b$$

where: a and b are the inputs; f_1 and f_2 are sum and carry outputs, respectively.

We can build the *Half-Adder* using 2 logic gates (XOR and AND gates for sum and logic, respectively) or using a single *related-fracturable* LUT_3 as shown in figure 2.1. Each LUT is a collection of 1-bit SRAM cells paired with a multiplexer to select the memory cell base on the inputs a , b and c . figure 2.2 shows the internal structure of a LUT_4 with 4 inputs and 16 SRAM cells. The advantage of utilizing LUTs is that, one can realize upto 2^{2^n} boolean functions using a single LUT _{n} .

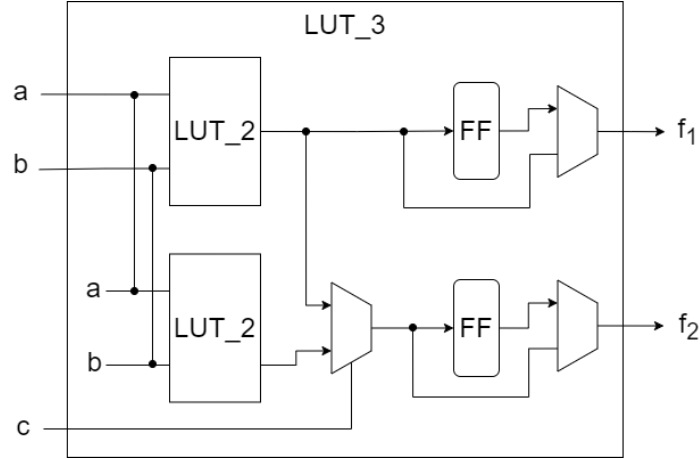


Figure 2.1: A LUT₃ realized using two LUT₂, where a and b are the two inputs of the boolean equation, while c is grounded to select the output of the bottom LUT₂. The outputs f_1 and f_2 are registered using the output FFs.

2.1.2 LUT-RAM

As shown in figure 2.2, LUTs are predominantly used to store data in their SRAM cells (*Read-Only-Memory*). Some FPGA models provide supplementary circuitry to write to LUTs, refurbishing them as a form of on-chip data storage. But, this conversion of generic LUTs to *LUTRAM* comes at the cost of increased area (due to supplementary circuitry), which does not scale efficiently for larger memory requirements, thus cementing the importance of BRAMs resources in a FPGA.

2.1.3 Block-Random-Access-Memory

As discussed in section 1.4 a multitude of accelerator applications require varied forms of intermediary storage elements to perform computation. The memory requirements of accelerators is volatile and demands custom solutions for different scenarios. Therefore, it becomes important to provide some form of on-chip storage that is fast (low-latency) and sufficiently large. Block-Random-Access-Memory (BRAM) aims to bridge this gap of low-latency, large on-chip storage, with reasonable scaling factor by utilizing SRAM cells. A BRAM consists of an ASIC type SRAM-base memory core, adorned with additional peripheral circuitry to make them more configurable and connect them with the PL with minimum latency,

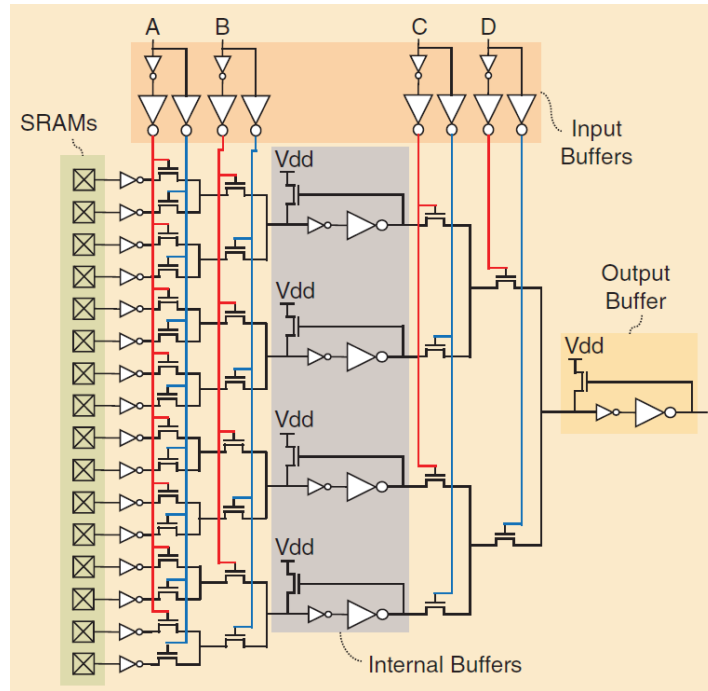


Figure 2.2: Internal structure of a LUT_4 with 4 inputs and 16 SRAM cells [5].

2.2 Static BRAM Allocation by HLS Tools

Designers typically need various types of RAM in a design, all of which must be implemented using the fixed BRAM and LUT-RAM resources on the chip. If designers were forced to figure out the optimal way to combine BRAM and LUT-RAM for each memory configuration and write Verilog to implement them, it would be time-consuming and would also restrict the design to a specific FPGA architecture.

Instead, vendor CAD tools include a RAM mapping stage that translates the logical memories in the user's design into the physical BRAMs and LUT-RAMs on the chip. The RAM mapper selects the physical memory implementation (including memory type, width, and port characteristics) and generates any additional logic needed to combine multiple BRAMs or LUT-RAMs to realize each logical RAM. It is for this reason, HLS tools require information of the total memory utilization of the architecture during *compile-time*, to efficiently automate the process of logical RAM generation.

Vitis HLS provides a few BRAM inference and synthesis guidelines [2] to help designers to understand the behaviour of the compiler, hinting which parts of the C++ code might be inferred as BRAMs. Arrays in software are mapped to *registers* or *memory* depending on the array's *size* and *access-pattern*. Typically, arrays are implemented as *RAM*, *ROM* or *registers* after synthesis. Arrays in the top-function's arguments are implemented as ports providing access to external memory, and internal arrays exceeding 1024 elements are realized as BRAMs.

Furthermore, FIFOs and PIPOs are ubiquitous in HLS designs and are predominantly implemented as BRAMs due to their large depth. Thus, requiring a larger effort from the designer to carefully structure memory-access datapaths to optimize BRAM utilization.

2.3 Dynamic Memory Allocation in Hardware

DMA has been extensively studied for decades at a software level, to support real-time workload requirements for software applications while minimizing physical memory resource usage. Before applying DMA at a hardware level, the significance of DMA at software and hardware levels must be established.

At a software level, DMA is commonly used to accommodate variables with unknown *lifetime* and un-bounded *size* during compile time to provide flexibility to the programmer and abstract memory management. Whereas, for accelerators on FPGAs, the *lifetime* of a memory variable (array) exists for as long as the accelerator is executing and only changes when the accelerator is removed/replaced physically (Dynamic-Partial-Reconfiguration (DPR)), hence, trying to abstract on-chip memory (BRAM) based on *lifetime* will be challenging.

From section 1.4 and section 2.2, the importance of a memory variable's (FIFO in the case of streaming applications) *size/bound* based DMA becomes evident. Therefore, this work will focus on implementing DMA to improve BRAM consumption efficiency and, abstract memory management for memory-variables of unknown bound, where the bound actively changes with workload during execution.

2.3.1 Dynamic Allocation Concept, Policies and Strategy

Dynamic-Memory-Allocation (DMA) occurs by splitting and coalescing of a common chunk of memory called the *Heap*. During an allocation request, a suitable size of memory is fetched from the heap and provided to the requesting application, and vice-versa during de-allocation. Memory allocation policies describe how a (de)allocation request is processed, and the strategy describes the algorithm behind splitting and coalescing of heap memory [39]. The policy and strategy employed determines the allocation complexity, speed and fragmentation of heap memory.

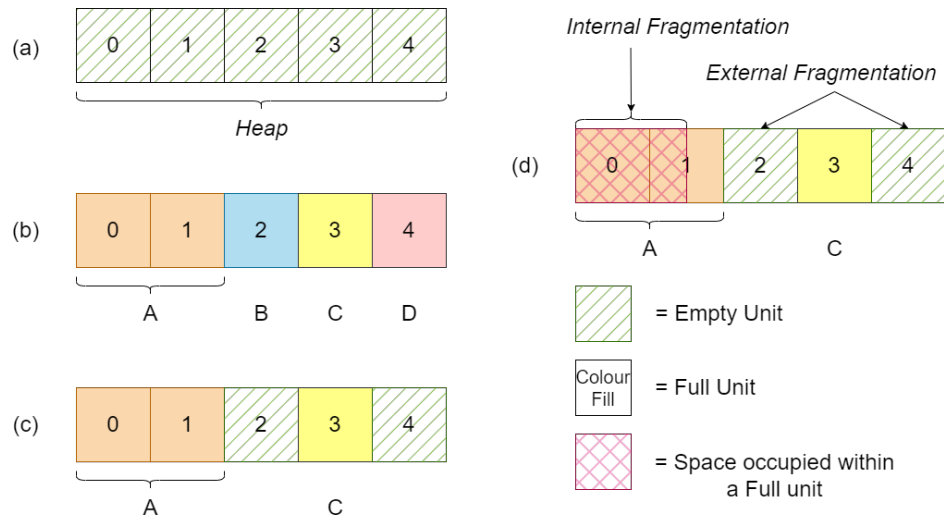


Figure 2.3: (a) Heap memory with free MAUs. (b) Heap state after four allocations A, B, C and D (shown by coloured squares). (c) Heap state after B and D are de-allocated. (d) Visualisation of Internal and External fragmentation.

2.3.2 Heap Memory and Dynamic-Memory-Management Unit

The *Heap* (figure 2.3) is a collection of M Minimum-Allocation-Unit (MAU), each of size n that together produce a heap of size $M.n$. Each MAU can be either *empty* (allocated) or *free* (un-allocated). On top of the heap lies the DMMU, which is responsible for managing the heap and handling (de)allocation requests from accelerators on the FPGA.

2.3.3 Allocation and De-allocation Requests

A process that wishes to obtain a virtual address space from the physical address space (heap), must send an allocation request to the DMMU. Upon receiving a request the DMMU is responsible for allocating MAUs to the process. On the other hand, the DMMU must safely de-allocate the MAUs in the presence of a de-allocation request.

2.3.4 Internal Memory Fragmentation

Individual allocation requests must be represented by a factor of MAU size. Hence the DMMU may only accommodate a request as a multiple of MAUs. *Internal fragmentation* is when a process does not utilize the complete allocated address space as shown in figure 2.3 (d).

2.3.5 External Memory Fragmentation

Consider the example shown in figure 2.3, the heap is progressively allocated with requests from process A, B, C and D, respectively, followed by the de-allocation of process B and D. Now, consider another allocation request of size $2MAU$ from process E appears at the DMMU when the heap is in the state as shown in figure 2.3 (d). Although there are 2 MAUs free, they are fragmented (non-contiguous) and the DMMU cannot allocate the request due to *External fragmentation* of the heap.

2.3.6 First-Fit

This algorithm searches the empty-list (unoccupied memory) and allocates the first large enough block to fit the allocation request. The allocated block from the free-list is split into two parts, one to fit the request and the other to append to the free-list. As a result this algorithm causes a considerable amount of external fragmentation due to constant reduction of MAU size.

2.3.7 Best-Fit

Best-Fit searches the whole free-list to find the appropriately size block in heap for the allocation request. This minimizes internal fragmentation, but at the cost of a longer runtime required to traverse through the complete free-list to find a suitable free block.

2.3.8 Next-Fit

This algorithm is an optimized version of First-Fit - it saves the position of the previous allocation in the free-list and resumes the search for the succeeding allocation request from the saved location. This helps reduce time-cost of the allocator but still suffers from large external fragmentation as First-Fit.

2.3.9 Worst-Fit

Worst-Fit searches through the whole free-list to find the largest free block and splits it similar to Free-Fit. Therefore the free blocks after the split are larger than the Free-Fit algorithm, thus reducing the external fragmentation. But, this algorithm also has a high time-cost due to the full list search.

2.3.10 Buddy-System

A Buddy-System splits a free block from the free-list into two smaller blocks, which can be recursively split until they are too small to split. The Buddy-System follows the Best-Fit algorithm and finds the free block of appropriate size to satisfy the allocation request. The resulting blocks (one allocated and the other free) from a split are buddies and associated together. This helps the Buddy-System to quickly identify buddies and coalesce them together into a bigger free-block, if both buddies are free. Thus, Buddy-Systems are fast and excel at quick searching, splitting and coalescing of the free-list.

2.3.11 Bit-Mapped Fit

The status (empty or full) of a MAU can be represented using a single bit and its value corresponding to empty (0) and full (1). In a Bit-Mapped Fit, a *Bitmap* is created to hold the status of all MAUs. The allocator is only required to operate on the bitmap to find the appropriate free block for allocation. This method has bounded worst time-cost because, irrespective of the allocation request sizes, the bitmap size is constant. However, this method is not used in software DMA due to slowdown incurred by bitmap address translations to pointer addresses.

FPGAs excel at parallel computations and accelerators are designed to take full advantage of parallelism, wherever possible. Therefore simply applying a sequential DMA algorithm tailored towards software will result in sub-par performance due to latency introduced by sequential operation. Hence, the software algorithms must be realized in hardware to take advantage of the parallelism and cater to multiple accelerators, simultaneously. Chapter 3 will discuss about the contemporary methods deployed on FPGAs to realize DMA.

3 Related Work

This chapter will explore the related work and simultaneously establish the two major bottle-necks of hardware implementations of DMA in FPGAs. Section 3.1 explores contemporary DMMU architectures and report on the bottle-neck of *External fragmentation* of heap memory, or on the *Cross-connect* resource overhead. Section 3.2 briefly summarizes the bottle-necks and the path taken by chapter 4 to address the bottle-necks.

3.1 Dynamic Memory Management Units

Early hardware implementations of DMA were made feasible by the advent of Bit-Mapped Fits (section 2.3.11) to implement a Binary Buddy-System (section 2.3.10). Studies on hardware implementation of the Buddy-System was first studied by Puttkamer in the 1970s [38], but suffered from a linear time-complexity of $O(n)$. Later, Chang and Gehringer [7] implemented a modified version of the Buddy-System in hardware, that was capable of achieving a bounded time complexity of $O(\log_2[n])$. The novelty of the approach by Chang and Gehringer stemmed from the use of a CBT (figure 3.1), to perform the Best-Fit search in a Buddy-System as well as the splitting and coalescing of the heap memory by flipping bits in a bitmap. The combination of a Binary Buddy-System and a Bit-mapped Fit, paired with dedicated circuitry, consequently led to the parallelisation/acceleration of DMA algorithms.

3.1.1 Modified Buddy System

The Modified Buddy-System (MBS) [7] is composed of three base hardware structures, *AND* tree, *OR* tree and the *Bit-flipper* tree.

- The *OR* tree is tasked with finding the smallest power of 2 that can accommodate the allocation request. Each level of the *OR* tree handles 2^l MAUs, where l represents the level in the tree. If the corresponding *OR* gate at level l in the tree is 0 (free), this implies there are 2^l MAUs free which can be contiguously allocated.
- After identification of the least number of MAUs required to allocate a request (Best-Fit), the *AND* tree is responsible for finding the address of the first free MAU at the corresponding level l .
- The *Bit-flipper* is a dedicated CBT with a predefined behaviour (logic) for each node, which performs the setting (1/full) and un-setting (0/free) of bits on the MAU list bitmap in parallel, starting from the root node.

The software implementation of DMA must perform coalescing of the free-list to update the address pointers. In contrast, the MBS implements the free-list using a bitmap, which automatically coalesces adjacent free MAUs into a bigger unit. This makes the MBS very fast compared to its software counter-part, but suffers from larger external fragmentation as only adjacent free MAUs are coalesced. Consequent work [6] reduces external fragmentation and increases allocation efficiency by 10% at the cost of more hardware to implement the buddy system.

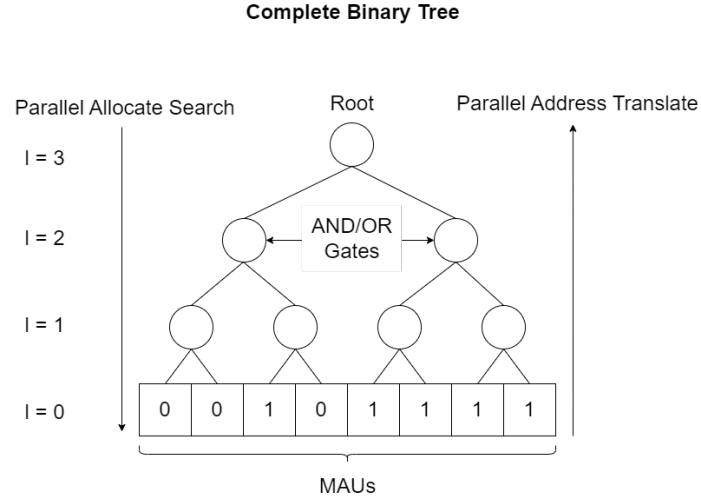


Figure 3.1: A visualisation of the CBT ($N = 8$) with leaf nodes (square) occupied by MAUs, represented by a bitmap (1 = full; 0 = free), and each node (circle) in the tree representing *AND* or *OR* gates. The parallel search and address translation progresses in $O(\log_2[N])$ time due to $l = \log_2(N)$ levels of the CBT.

3.1.2 SysAlloc

SysAlloc [40] aims to manage a large scale of MAUs ranging from BRAMs to DDR memory. SysAlloc is based on MBS and manages the heap through a bitmap and communication is handled using a memory-mapped bus. This categorizes SysAlloc into a serial communication paradigm, with proportional increase in latency introduced by the growth of simultaneous accelerator accesses to the heap.

The novelty of SysAlloc is its ability to manage large scale heaps without a drastic increase in FPGA resources. SysAlloc achieves this performance by storing the MBS *AND-OR* gate nodes values as a bitmap (memory-based tree) in memory (BRAM or DDR), rather than in registers or at the gates themselves, as done in other works employing MBS.

To minimize latency introduced by access to memory structure used to store the bitmaps, SysAlloc utilizes a *k*-nary *Cluster Tree* storage structure to capitalize on the bandwidth of data transfers, in which a group of nodes from the CBT are clustered together into a bitmap and stored for future access. Upon an allocation request, the corresponding cluster is retrieved from memory and operated upon by the minimal hardware implementation of MBS.

3.1.3 OLDMA and TLDMA

Sadeghi et al.,[30] propose two architectures, *One-Level DMA* (OLDMA) and *Two-Level DMA* (TLDMA). In the MBS, the first free MAU (0 in the bitmap) address is computed using an *AND* tree, in-order to reduce the critical path, OLDMA applies a novel address determination mechanism by splitting the MAU bitmap into multiple sub-regions, each of which has its own '0' (free-MAU) finding hardware. This breaks the critical path and improve allocation latency, but limits OLDMA to 512 MAU blocks.

Their second work TLDMA is designed to handle a larger range of MAUs (upto 64K). They achieve this by using a similar approach to section 3.1.2 to represent the CBT as a cluster of hierarchical nodes. In contrast to *SysAlloc*, they build the whole tree in hardware, saving up on expensive memory access, thus reducing the allocation latency of the system at the cost of marginally increased implementation complexity.

3.1.4 Free-List-Manager

In an attempt to further decrease fragmentation in MBS based architectures, *Free-List-Manager* (FLM) [28] proposes a creative solution. The heap is split, once into blocks, and in-turn each block is split into sub-blocks of different classes (sizes). In other words, FLM splits the heap into MAUs and each MAU into sub-MAUs. Therefore, each allocation request is satisfied by a combination of sub-MAUs.

To efficiently allocate the hierarchical sub-MAUs, FLM maintains a *link-vector*, which is a bitmap description of each sub-MAU and the links between non-contiguous sub-MAUs within the MAU. Hence a MAU is only full when all the sub-MAUs have been allocated. A FIFO for each class/size of sub-MAUs is maintained and upon receiving an allocation request, the top element of the corresponding class is popped and the corresponding bitmap is issued to the *AND-OR* tree.

The FLM architecture significantly reduces external fragmentation by supporting allocation of non-contiguous MAUs from heap using link-vectors, but this method comes at the cost of latency introduced due to address translation required to successfully map virtual addresses to non-contiguous physical address.

The architectures *SysAlloc*, *OLDMA*, *TLDMA* and *FLM* use the *Buddy-System* (section 2.3.10) as the base allocation strategy. The Buddy-System is a form of *Segregated free list* strategy and achieves fast allocation but suffers from high external fragmentation. The previously discussed works handle a large number of MAUs and focus on sharing the heap across all requesting processes.

This results in two problems for streaming applications (section 1.4) and accelerators. The first is, every accelerator must share the same AP to access the heap, forcing a sequential memory access and negatively impacting parallelism. Secondly, streaming applications tend to use large sized FIFOs to handle incoming/outgoing streams, hence an individual MAU can be large (as big as the whole BRAM) and a very large count in the heap is not necessarily required.

The next set of architectures will take a different approach from the previously discussed works and provide more insight into a distributed and parallel approach towards DMA in FPGAs.

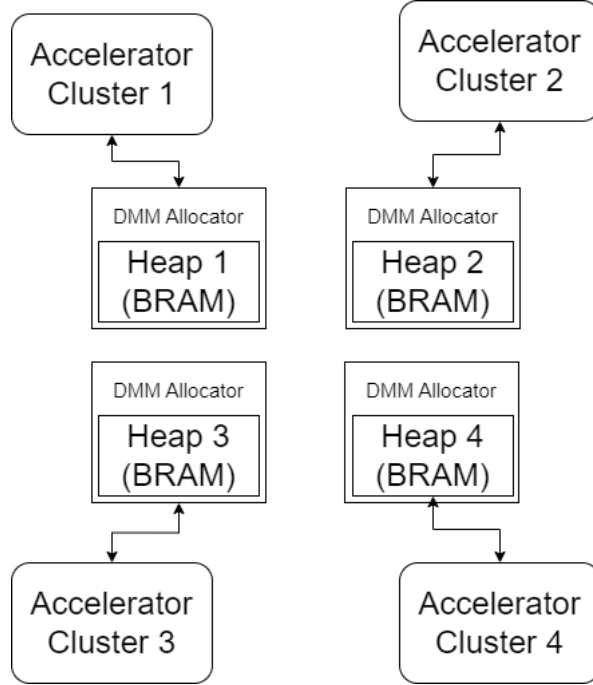


Figure 3.2: Representation of parallel memory access configuration for multi-accelerator system in [10]. Each sub-system (using DMA) within a cluster acquires a sequential memory access latency.

3.1.5 DMM-HLS

Diamantopoulos et al., in [10] present a *Sequential list* based DMMU implementing the *First-Fit* strategy for allocations in the heap. There are no objective values provided for the allocation operation i.e., the number of cycles consumed to return a valid allocated address in the heap. However, they have accounted there is a latency overhead of $1.2x$ incurred per accelerator in a 16-accelerators, 16-heaps configuration. This configuration is used to measure the allocation latency since there is no latency overhead caused by sequential access to heap through the shared bus.

DMM-HLS addresses the problem of parallel memory access by allocating a separate heap to each accelerator, resulting in memory access latency comparable to HLS's static memory allocation. The improvement in memory access latency come at the price of increased FPGA resource consumption due to DMM allocator instance per heap as shown in figure 3.2, which limits the maximum accelerators that can be deployed simultaneously. The throughput of certain algorithms benefit from sharing the same heap while certain algorithms perform efficiently by having individual heaps [10], suggesting DMA in hardware is tightly coupled with the memory access pattern of accelerators.

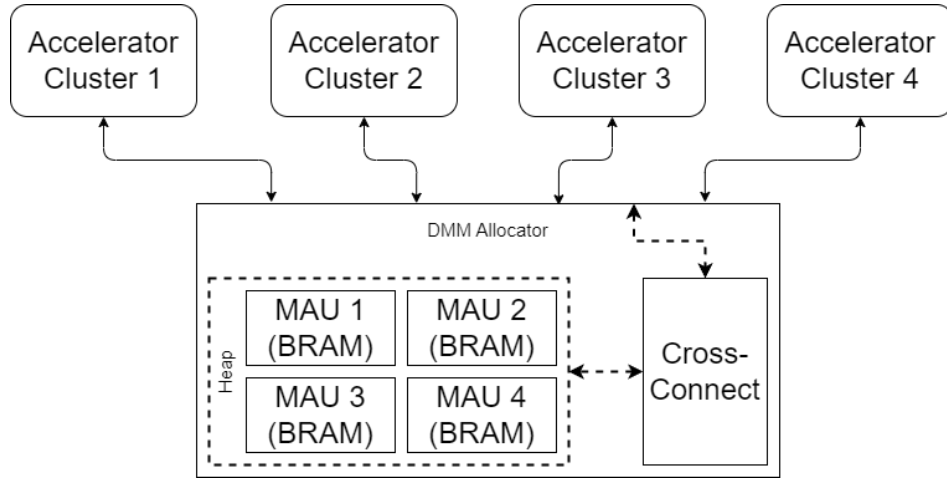


Figure 3.3: Representation of parallel memory access configuration for multi-accelerator system in [9]. Each sub-system (using DMA) within a cluster acquires a sequential memory access latency, but the each cluster is allocated on dedicated BRAMs.

3.1.6 DOMMU

Embedded applications demand low-latency memory access operations, thus [9] aims to provide this by enabling run-time DMA of BRAM modules to accelerators. As show in figure 3.3 the heap consists of a collection of BRAMs, which are collectively managed by a single DMMU. This alleviates the resource overhead caused by multiple instances of heap managers in [10].

The allocation mechanism is decoupled from the memory access mechanism (cross-connect), improving access latency per accelerator. DOMMU utilizes a standard multiplexer CBT to access individual BRAMs in the heap. The switching mechanism consumes 50% of the total resources consumed to implement DOMMU causing a resource bottle-neck. The authors suggest utilizing the LUTs to build an efficient decoder to access memory [13], but this method requires DPR for implementation and is not feasible for real-time re-configuration due to the high re-programming latency of re-configurable LUTs [3].

3.2 Bottle-necks

Parallel memory access always incurs a resource consumption overhead in the form of extra DMMU units [10] and(or) cross-connect required for simultaneous heap access [9]. Each BRAM is inherently limited by the number of access ports [1]. Hence for DMA with focus on parallel access, each BRAM must be treated as the MAU. When treating each BRAM as a MAU, introduces *Internal fragmentation* for accelerator applications that do not utilize the complete BRAM. Internal fragmentation is minimum For streaming applications with demand for large FIFOs and PIPOs as they tend to utilize the whole BRAM. Therefore it becomes important to focus on reducing *External fragmentation* while handling a smaller collection of BRAMs in the heap. If internal fragmentation is to be reduced, any of the above solutions can be deployed hierarchically by treating each allocated BRAM as a heap. The other major bottle-neck contributor is the expensive *Cross-Connect* implemented to access the heap. This will be addressed by exploring different interconnection networks in chapter 4.

4 Design Space Exploration

This chapter will describe the parameters/metrics of interest, explore how they may be improved with respect to constraints such as, streaming friendly architecture, parallel memory access support and scalability. Firstly, this chapter will explore the efficacy of *Buddy-System* against a optimized *First-Fit* allocation scheme at maximizing *Packing Efficiency*. Followed by the study of an accelerator model designed and synthesized solely in HLS, to better understand the need for memory level abstraction and how static memory allocation negatively impacts BRAM resource consumption. And lastly, explore *Switching-Networks* for parallel memory access support.

4.1 Optimization Parameters

There are many parameter that affect a DMA system, such as, *Internal fragmentation*, *External fragmentation*, *Allocation latency*, *Access latency*, to name a few. Since the size of a single MAU is the complete BRAM, internal fragmentation (section 2.3.4) will not be of concern and the focus will lie on minimizing external fragmentation (section 2.3.5). The rest of the metrics of interest will be as follows:

Packaging Efficiency To quantitatively measure external fragmentation caused by an allocation scheme, packing efficiency is introduced as a metric. Packing efficiency P_{eff} is measured as a ratio of 'Number of MAUs allocated' M_a over 'Heap size' H_s . This metric will help expose the presence of external fragmentation in the heap (figure 2.3).

$$P_{eff} = \frac{M_a}{H_s}$$

Allocation Latency The time taken (cycles) by the DMMU to present a valid address (virtual/physical) to the requesting process, once a allocation request has been received.

Deallocation Latency The number of cycles taken by the DMMU to process a deallocation request from a process.

Translation Latency The number of cycles consumed by a DMMU to translate a virtual memory address (from the accelerator) to a physical memory address (in the heap). Translation latency can be further classified into:

- **Intra-BRAM Latency** is introduced for translation of virtual addresses that fall within the same BRAM. That is, *Intra-BRAM* latency applies if the previous and current virtual addresses lie within the scope of the same BRAM/MAU in heap.

-
- **Inter-BRAM Latency** is introduced for translation of virtual addresses spanning across BRAM boundaries, i.e., previous and current virtual addresses lie in different MAUs.

Access Latency The DMMU may add latency over the memory bus, but BRAMs have an inherent access latency of 1 *cycle* for better timing.

Translator In order to access the heap, the accelerator must convey its memory request to an *Access-Point*, which in-turn will pass the address to the corresponding BRAM. In the case of a DMMU operating on virtual addresses, an accelerator's access request must be translated into a physical address for the BRAM. The number of parallel memory accesses possible is directly related to the number of translators supported by a DMMU.

4.2 Impact of Heap Memory Compactness

The impact on heap memory resource (BRAM) utilization due to the Buddy-System mechanism must be inspected to visualize if improving heap memory compactness (P_{eff}) has benefits. A testbench is built in python to loosely simulate, generalized heap memory request patterns, of an scientific-application's (*Sci - Comp*) *major variables* (highest memory footprint) [15]. The simulation is carried out for heap memory sizes (H_s) ranging from 8 - 128 MAUs, with each major variable described by the three qualities:

1. **Size** – The size the variable will occupy in memory.
2. **Lifetime** – The time interval between a variable's allocation and deallocation from the heap. The lifetime of a variable describes, for how long a variable will occupy space in the heap memory.
3. **Request Pattern** – An application/process may have multiple major variables of different sizes and lifetimes. The order of allocation and deallocation directly affects the compactness of the heap memory.

Following the study in [15], each *Sci - Comp* process's memory footprint in the heap across the execution period of the process, may be loosely represented by three classes of major variables, *a*, *b* and *c*, respectively depending on the variable's size and lifetime. Class *a* variables are small and short lived, they make up $\geq 60\%$ of the major variables in an application, followed by class *b* ($\approx 30\%$) variables which are medium sized and lastly, big and long lived class *c* ($\leq 10\%$) variables.

For example, if an application is assumed to have 16 major variables, then a plausible variable distribution is 10 class *a*, 5 class *b* and 1 class *c* variables, respectively. Similarly, the major variables of individual classes have a distinct lifetime distribution, $\approx 10\%$ of a process's execution lifetime for class *a* (very short lived) and $\approx 80\%$ for a class *c* (very long lived) variable.

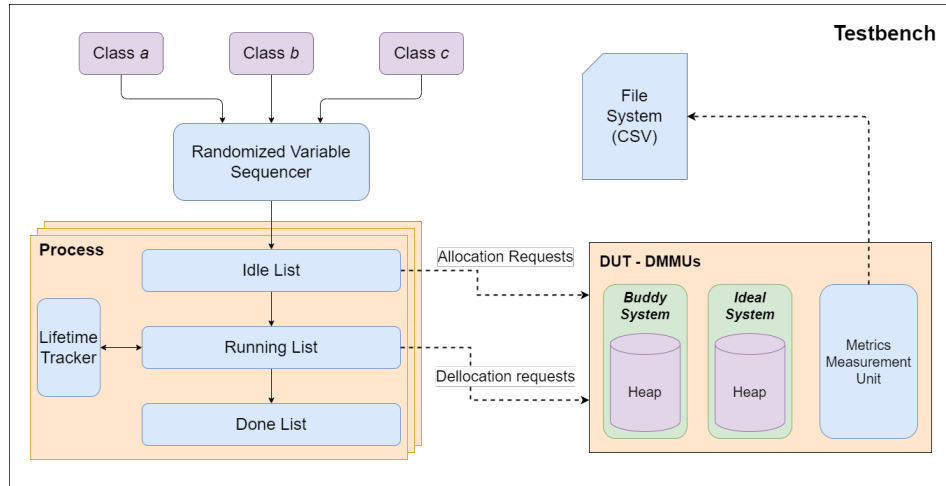


Figure 4.1: Functional representation of the python testbench used to measure, packing efficiency and MAU utilization at allocation failure.

4.2.1 Python Testbench

The testbench (figure 4.1) creates a synthetic load of (de)allocation requests in-order to measure the performance of Buddy-System and Ideal DMMU models. To model random request pattern, a *Random Variable Sequencer* generates a randomized sequence of variables (from each class) and places them in a *process's Idle-list*. The distribution of variables from each class and the total number of variables are predefined. A running process will move the idle variables to *Running-list* and generate an allocation request to the Device-Under-Test (DUT), which consists of two DMMUs. The *Parameter Tracker* monitors the lifetime of each variable in the running-list. When a variable expires, it is moved into the *Done-list* and a deallocation request is generated.

A DMMU in the DUT ceases operation when it fails to allocate an allocation request. The rejection of an allocation request signifies the DMMU can no longer accommodate variables in the heap, and the *Measurement Unit* is notified to capture the DMMU metrics. The *Ideal-System* exhibits an ideal behaviour of achieving maximum packing efficiency by immediately coalescing free MAUs together after a deallocation. Thus, completely avoiding external memory fragmentation as shown in figure 4.2.

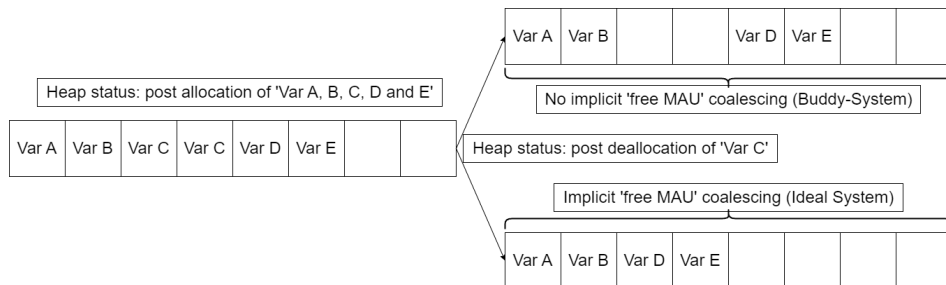


Figure 4.2: Visualisation of the heap's status pre and post deallocation of 'Var C' which occupies 2 MAUs in heap. Shown is the external fragmentation caused by Buddy-System because it lacks implicit coalescing, and the behaviour of an ideal heap management scheme with implicit coalescing (ideal heap utilization).

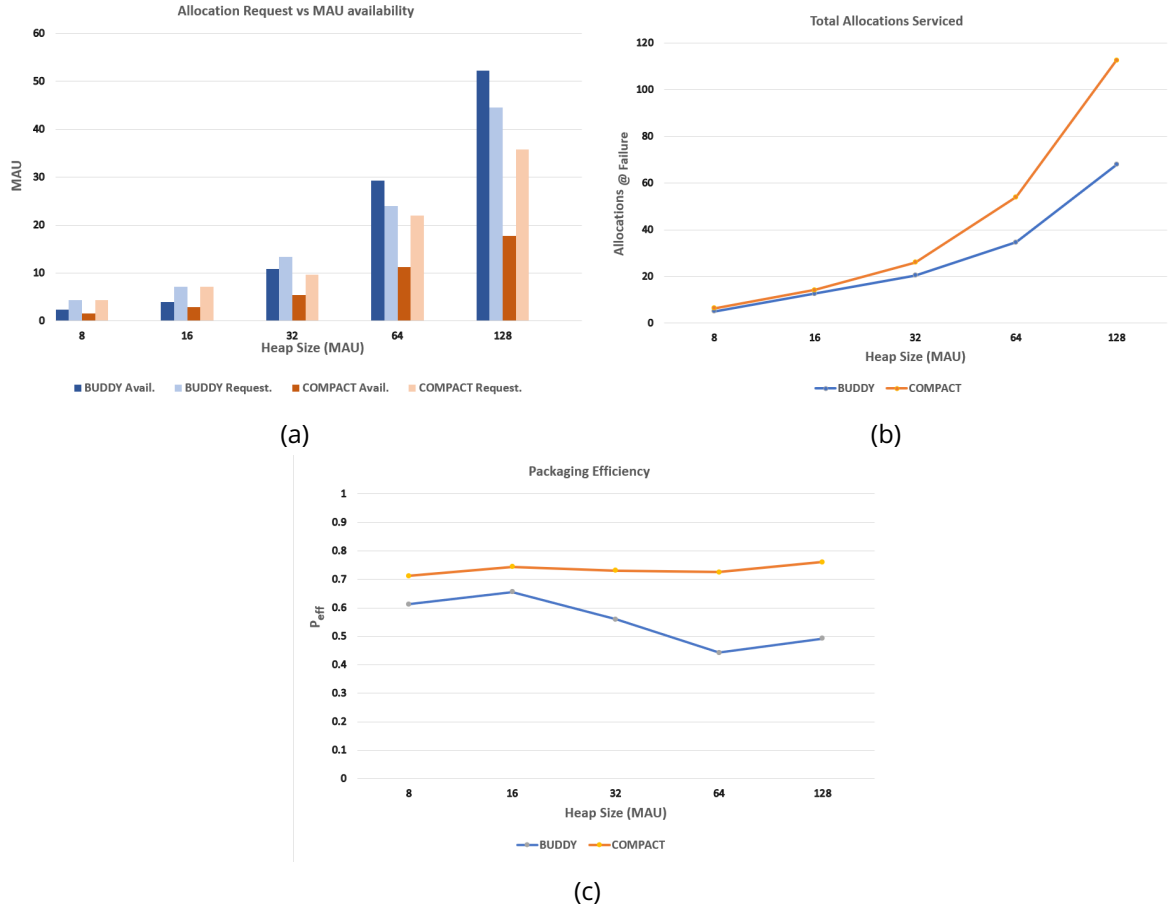


Figure 4.3: (a) Visualization of bottleneck caused due to external fragmentation; Dark columns present the available size in the heap, while light columns present the requested size. (b) Process completion measured in-terms of the total number of allocations performed (higher is better). (c) Measure of a heap's compactness with varying heap sizes.

4.2.2 Packaging Efficiency Measures

Shown in figure 4.3 is the representation of packing efficiency in the simulated Buddy-System (Buddy in graph) and Ideal-System (Compact in graph), visualized in three different graphs. All the metrics are measured at DMMU failure, i.e., no more allocations can occur without a deallocation. The first graph (figure 4.3a) depicts the number of MAUs occupied in heap (dark column - *BUDDY/COMPACT Alloc*) and the failure causing allocation requests size (light column). As the heap size grows, the Buddy-System fails to allocate requests in-spite of having free MAUs present in the heap (dark column greater than light column). This signifies external fragmentation becomes a bottleneck with growing heap in the Buddy-System.

Another metric show in figure 4.3b is the total number of allocations performed for a process before reaching failure. The number of allocations successfully serviced relates directly to the completion of a process, where a process is considered to be complete when all the processes in the *Idle-List* move to the *Done-List* (figure 4.1). Therefore, the Ideal-System progresses a process further before running out of heap compared to the Buddy-System.

Finally, shown in figure 4.3c is the measure of compactness achieved by the two DMMUs. The Ideal-System exhibits an efficiency of 73% consistently across all heap sizes, whereas the Buddy-Systems P_{eff} falls significantly as the heap size increases due to external fragmentation of heap. Hence, it is worth employing an Ideal-System to achieve better BRAM utilization at the cost of increased DMMU complexity.

4.3 Virtual Dynamic Memory Management

The Ideal-System discussed above can be realized by converting the physical address management of the Buddy-System into an virtual address management system. Borrowing the concept of *Paging* from software DMA, the shortcoming of external fragmentation in the Buddy-System can be remedied. Accelerator applications exhibit a multitude of memory request patterns, which tends to be the primary driver behind external fragmentation of the heap. By mapping the physical memory (BRAM address map) onto a virtual address space, the compactness of the heap can be maximized. Thus, enabling the DMMU to allocate requests as long as there are enough free MAUs present in the heap.

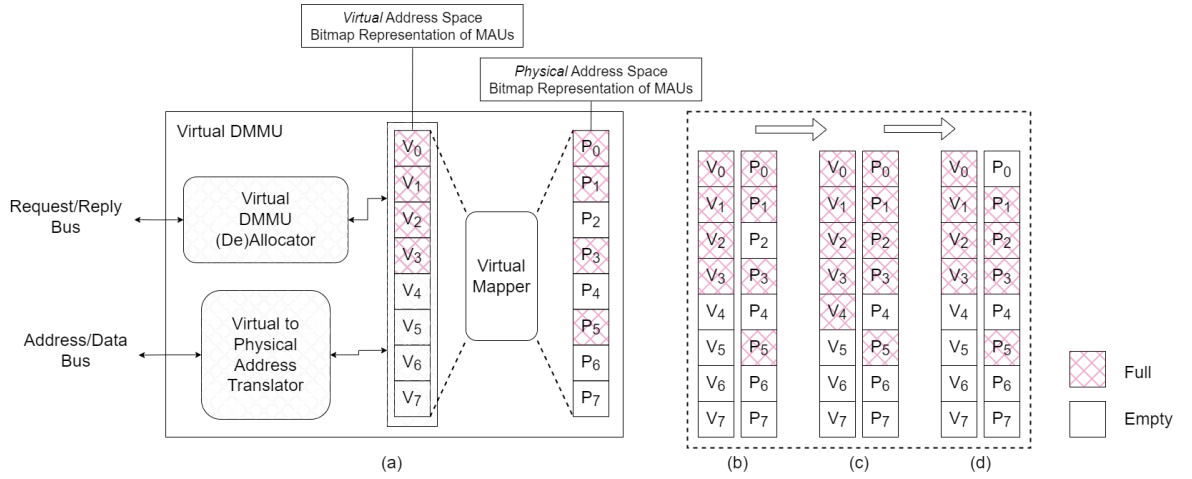


Figure 4.4: (a) Visual representation of the conversion of a physical address space P_x into a virtual space V_x by a paging system applied on top of the Buddy-System. (b) Initial state of virtual and physical maps. (c) Post allocation of $V_4 \rightarrow P_2$. (d) Post deallocation of P_0 , the virtual map is re-applied to obtain a new state. The virtual space of (b) and (d) are identical but differ in physical maps.

Paging maps the physical address space of the heap onto a virtual address space which is completely contiguous from the perspective of the application accessing the heap. The paging mechanism splits the physical space of the heap into fixed sized *Frames* (MAU) which are later allocated to an application upon request. The frames need not be contiguous in physical space, but will always be contiguous from the perspective of the application in virtual space.

Shown in figure 4.4, is a rudimentary visualisation of paging applied to the Buddy-System. The physical space contains frames (MAUs) P_x , mapped by the *Virtual Mapper* into a virtual space where all the MAUs V_x are contiguous. Therefore, the physical memory locations P_2 , P_4 , P_6 and P_7 although non-contiguous, will appear contiguously as V_4 , V_5 , V_6 and V_7 in virtual space. Therefore, any allocation request that is serviceable will be satisfied by allocating

contiguous frames in virtual space, which may be a sequence of non-contiguous frames in physical space.

In-addition to completely exterminating external fragmentation of heap, paging provides the advantages of abstracting the memory organisation of the heap from applications by enabling applications to address memory locations using a *logical address*. This means the application requires no knowledge about the physical address offset and can simply access the heap using its logical address, thus reducing overhead seen by processes. But, this comes at the cost of increased complexity due to, physical to virtual address mapping, logical to physical address translation and auxiliary mechanisms required to maintain the *Page-Table* used for address translation.

4.4 Support for Parallel Memory Access

One of the main goals of this work is to supplement DMA with multiple simultaneous connections to the heap (APs). Previously explored architectures [7, 40, 30] and [28] do not concern themselves with providing parallel access since they focus on sharing a single BRAM as a heap and are inherently limited by the number of ports present in a BRAM [1]. The architecture from [10] provides a solution for parallel access by allocating accelerators among multiple heaps (BRAMs) and [9] tackles parallel access with a MUX based approach.

A key concern when providing parallel access to multiple BRAM ports is to keep the solution scalable. As seen from [9], the resource consumption of a MUX based solution simply does not scale well with growing number of APs. Hence, this section will explore few widely used *Switching Networks* and their growth complexity, with the MUX based approach (CBT) as the baseline.

4.4.1 Interconnect

At a hardware level, the signals (data/address) transmitted to and from the accelerator, must go through the AP and to the heap using some form of interconnect. An interconnect [33] is a well defined arrangement of switches/wires that can be configured to establish a connection from any input port to any output port. Interconnects can be widely categorised as *Packet* or *Circuit* switching.

A packet switched interconnect's switches can route packets (data) from input (source) to output (destination), through paths that may change during run-time. This provides better flexibility and congestion (traffic) management, at the cost of added communication latency. Whereas, a circuit switched interconnect establishes a (non-conflicting) dedicated path from input to output port prior to data transfer. Since the communication channel is established prior to data transfer, the critical path in a circuit switched interconnect is smaller as the control and datapath are separated. Hence a circuit switched network can achieve lower latency.

In conclusion, a packet switching is most suitable for large interconnects with small data-packets and circuit switching is best for small interconnects with large data-packets [24]. Since the aim is to have minimum latency and multiple dedicated connections in parallel, this work

will apply circuit switching concepts to interconnects. To find a suitable interconnect, the following factors, *Latency*, *Depth*, *Scalability* and *Setup-Time*, must be considered:

Depth The depth of an interconnect specifies the number of switches/nodes a packet must traverse in-order to reach its destination from the source. This factor contributes directly to the access latency (section 4.1) of the DMMU.

Scalability The provision of contention free communication between multiple sources and destinations demands the availability of multiple physical paths. Hence a scalable interconnect must achieve contention free operation (provided proper setup), between its inputs (APs) and outputs (BRAMs) with minimal FPGA resource utilization. This factor impacts the number of translators (section 4.1) a DMMU can support.

Setup-Time The establishment of a dedicated channel/link between source and destination in a circuit switched interconnect, results in an overhead called the setup-time. The setup-time affects the translation latency (section 4.1) of a DMMU.

The usage of an unsuitable interconnect will cause a bottleneck, not only in terms of resource utilization but, also in terms of maximum achievable performance of the DMMU. Therefore, a viable interconnect is expected to exhibit the following characteristics:

- The interconnect must have a small *Depth* to facilitate fast and low *Latency* communication with BRAM elements.
- Exhibit a suitable scaling factor, preferably $\log_2(H_s)$, where H_s is the heap size or the number of BRAM elements. This is required to minimize resource overheads caused by implementing support for parallel memory access.
- The interconnect must have a small *Setup-time*, so that any new connection from AP to BRAM, may be setup in the shortest time possible.

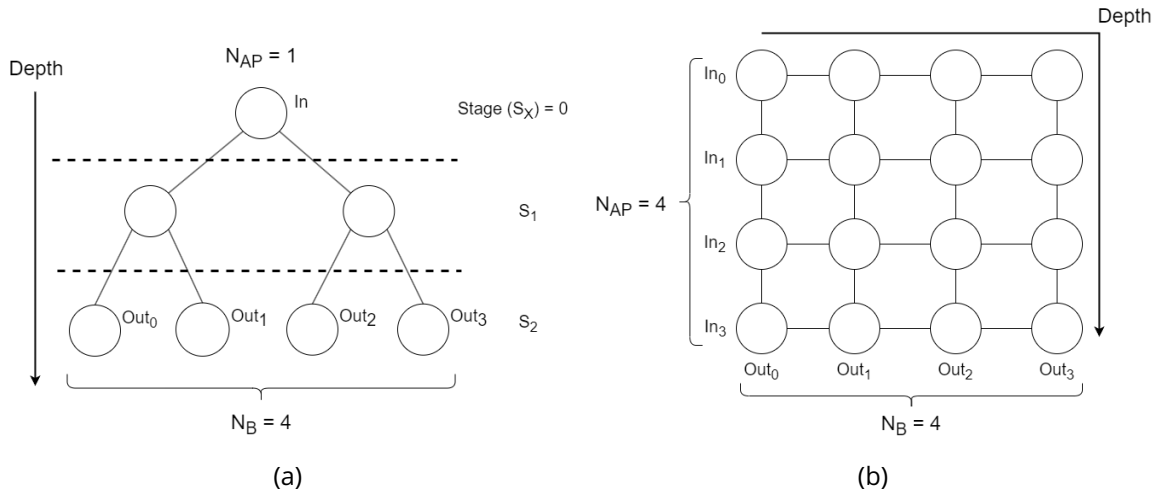


Figure 4.5: (a) Representation of a CBT interconnect where each node is a 2×1 MUX with the root node as input source and leaf nodes as output destination. (b) 2D-Array interconnect with 4 APs and BRAMs.

4.4.2 Complete Binary Tree

The decoder and MUX based interconnect used in [9], for simplicity can be modeled as a collection of 2×1 MUXs arranged as a CBT, with the AP situated at the root and BRAMs at the leaf nodes. FPGAs optimize the CBT using LUTs to realize MUXs [32, 26], hence the *Cost-Complexity* (C_C) described here will be a worst-case estimate. Therefore, the other interconnects will also express their C_C in terms of the number of 2×1 MUXs utilized.

The CBT model (figure 4.5a) is the fastest solution as it exhibits a depth of $\log_2(H_s)$ and has a setup time of 1 cycle due to DTR. But it achieves poor cost-complexity with scaling for higher counts of APs because the CBT requires a single root node per AP. Let the total number of BRAM elements in the heap be N_B and the number of APs be N_{AP} . If both number of BRAMs and APs are expressed as a power of 2 and considering N_{AP} is commonly a factor of N_B [9, 10], then the cost-complexity of CBT interconnect model with respect to scaling N_{AP} is as follows:

$$\begin{aligned}
 \text{Heap Size } (H_S) &= N_B \\
 \text{Access Points } (N_{AP}) &= N_B \cdot x^{-1} \\
 \text{MUX count } (MUX_C) &= N_B - 1 \\
 \text{Cost-Complexity } (C_C) &= N_{AP} \cdot MUX_C \\
 &= N_B^2 \cdot x^{-1} - N_B \cdot x^{-1} \\
 &= N_B^2 - N_B \quad ; \text{ When } N_{AP} = N_B
 \end{aligned}$$

4.4.3 2D-Crossbar Array

The 2D-Mesh (figure 4.5b) is a *strictly non-blocking* network in a uni-cast system, i.e., a path from source to destination can always be setup without disturbing existing connections. In-addition, 2D-Mesh provides better path redundancy between source and destination, i.e., multiple pathways are possible but need not necessarily be the shortest path. For the above reasons, 2D-Mesh networks are most suitable for networks with a lot of traffic and congestion, but they suffer from poor scalability.

A single node in 2D-Mesh comprises a cross-bar switch realized by two 2×1 MUXs. Furthermore, the number of nodes N_n grows with $O(N_B^2)$ resulting in high cost-complexity and poor scalability but can inherently support $N_{AP} = N_B$ parallel APs.

The setup time of the interconnect is not trivial to compute as in the case of CBT, each individual nodes contribution to complexity depends on the algorithm used to map connections from source to destination. Therefore for simplicity, it is assumed the control-path (setup process) is synchronous and sets up a stage per cycle (ideal). This results in an approximate setup time of $O(N_B)$, which are the number of stages (rows/columns).

$$\begin{aligned}
 N_n &= N_B^2 \\
 MUX_C &= 2 \cdot N_n \quad ; \text{ Each node contains two } 2 \times 1 \text{ MUXs} \\
 C_C &= 2 \cdot N_B^2
 \end{aligned}$$

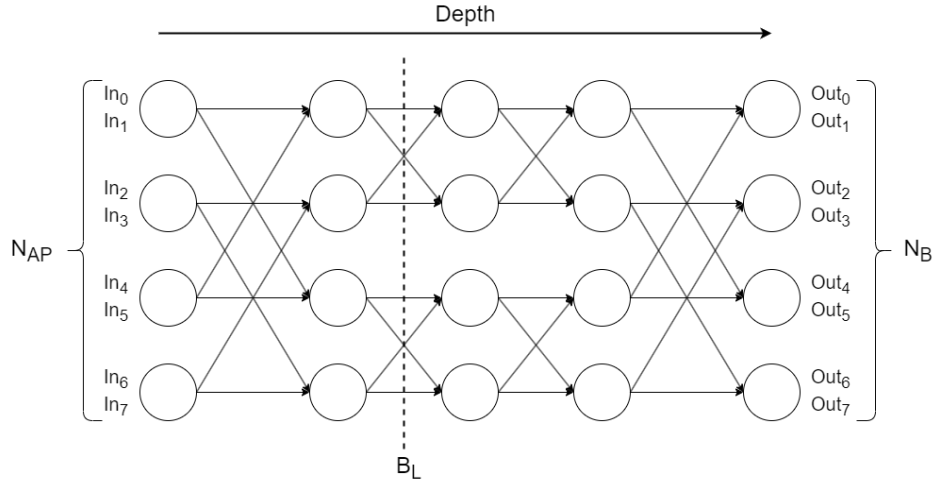


Figure 4.6: Representation of a Benes interconnect switching $8 N_{AP}$ to $8 N_B$. All nodes to the right of the bisector line (B_L) follow DTR when those to the left are setup accordingly.

4.4.4 Benes Network

A Benes network is a *rearrangeably non-blocking* network [13], i.e., in a uni-cast network, a path from source to destination may always be setup by rearranging existing connections. A Benes network is built by placing two butterfly networks back-to-back. Similar to 2D-Mesh a Benes node comprises two 2×1 MUXs, but it boasts a much smaller node growth (N_n) factor of $O(\frac{N_B}{2} \cdot \log_2[N_B] - 1)$. The Benes exhibits a depth and setup (ideal) time of $2 \cdot \log_2(N_B - 1)$. The cost-complexity for Benes when $N_{AP} = N_B$ is $N_B \cdot 2 \cdot \log_2(N_B) - 1$. Table 4.1 consolidates the *depth*, *cost-complexity* and *setup time* of the discussed interconnects.

Interconnect	Depth	Cost-Complexity	Setup-Time
CBT	$\log_2(N_B)$	$N_B^2 - N_B$	1^*
2D-Array	$2 \cdot N_B$	N_B^2	N_B^*
Benes	$2 \cdot \log_2(N_B - 1)$	$N_B \cdot 2 \cdot \log_2(N_B) - 1$	$2 \cdot \log_2(N_B - 1)^*$

Table 4.1: Comparison of Interconnect characteristics. **Ideal*

The CBT exhibits the smallest depth/latency $\log_2(N_B)$ and fastest setup time, but suffers square growth in cost-complexity. The 2D-Array performs poorly in all departments but that is because the strengths of 2D-Array networks lie in providing low congestion connections while being strictly rearrangeable. Since the this work's system is framed as uni-cast, i.e., at any given time, each AP is connected to a unique BRAM in the heap. The input-output permutations (possible connections) are unique, removing the need for a strictly rearrangeable network and consequently does not take advantage of 2D-Array's strength. Therefore the 2D-Array is discarded due to its large cost-complexity.

Finally, the Benes network is selected as the most suitable candidate due to its $\log_2(N_B)$ scaling of cost-complexity. The setup time of Benes shown in table 4.1 is an over-simplification (hence ideal) and is further explored along with the implementation of Benes network on an FPGA in chapter 5.

5 Implementation

The implementation of *VDMMU* (figure 5.1) follows a bottom up approach with the process centered around the *Heap Manager*. The rest of the modules *Translator*, *Mapper* and *Interconnect* are developed around the *Heap Manager*. This chapter will introduce the various modules present in *VDMMU* and evaluate the operation of the system on a *PYNQ-Z2* board hosting the *Zynq7020* SoC.

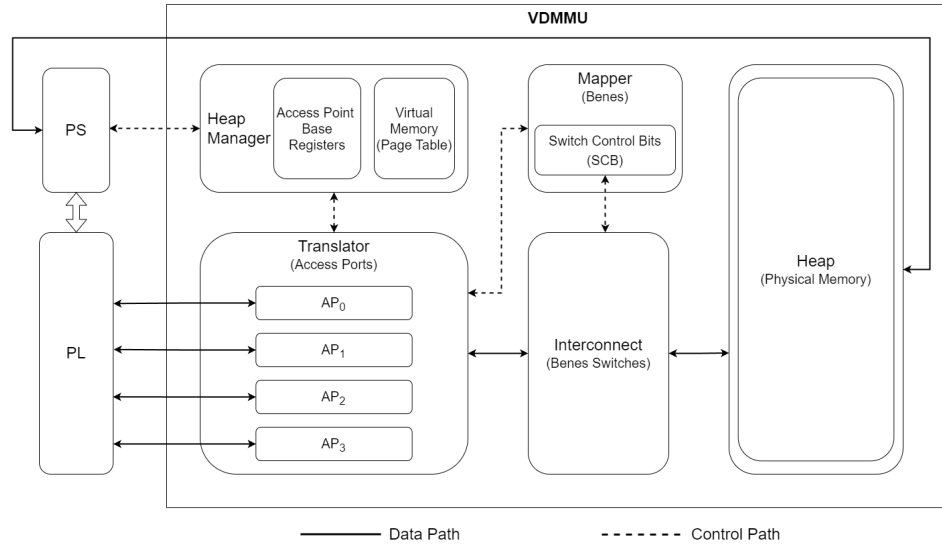


Figure 5.1: A visualization of the *VDMMU* system architecture with four APs. The control and data path are independent of each other owing to smaller critical path delays and (de)allocation requests appear from the *PS*.

5.1 Heap Manager

The *Heap Manager* is tasked with performing (de)allocation of MAUs, where each request received is processed sequentially. This includes *Request Validation*, *Status Manipulation* and *Coalescing* of the *Free/Full* list of heap memory. The manager serves the requests on a first-come-first-serve basis in which all Processing Elements (PEs) have the same priority. The implementation of the *Heap Manager* begins with the exploration of the novelty of Modified Buddy-System [7], in using the CBT structure to perform *Request Validation*, *Free address* search and MAU bitmap *Status Manipulation* in a single cycle (parallel operation), achieving low latency operation.

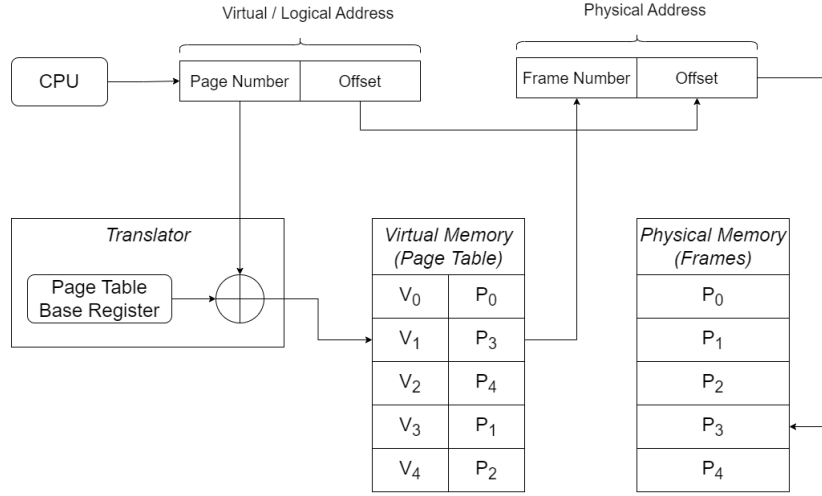


Figure 5.2: Paging mechanism using a *Page-Table* acting as virtual memory (logical address space) to map non-contiguous *Frames* (P_X) (physical address space) into a contiguous virtual address space (V_X).

To obtain minimum (de)allocation latency while performing paging (figure 5.2), i.e., finding a free MAU in virtual memory (page table register pointing to a BRAMs physical port number) and setting the MAU status (free/full) must be done in parallel. Hence the search for a combinatory structure similar to the CBT is necessary to perform paging with low latency.

First we monitor the behaviour of a simple paging mechanism that allocates memory in heap using the *First Fit* algorithm (section 2.3.6) to search the free list of the page table (virtual memory). Since the size of each MAU in the virtual memory (page table) is the same, the fastest implementation of *First Fit* would be to assert the *Request Validity* R_V of an incoming allocation request of size AR_B by checking if:

$$R_V = \begin{cases} 1, & \text{if } AR_B \leq N_B - F_P ; \text{ where } F_P = \text{first available free MAU in virtual memory.} \\ 0, & \text{otherwise} \end{cases}$$

If virtual memory is represented as an array of MAU pointers (physical address) and similar to [7, 10, 28, 30] the free list is extracted from a bitmap to perform (de)allocation. The free list must consist all free MAUs in succession in-order for single cycle allocation to be possible, else the *First Fit* algorithm must run multiple cycles, traversing the virtual memory bitmap finding a suitable collection of free MAUs in the heap that satisfy the *Request Validity* equation above. Therefore the virtual memory must be implemented as an rearrangeable array, where the free list is contiguous as shown in figure 5.3.

To keep the free list contiguous, the virtual memory pointers (registers) must be coalesced after deallocation(s) to reflect the new compacted heap state. The compacting of virtual memory can be performed by grouping full MAU pointers to the left and free pointers to the right to create a contiguous full/free list as shown in figure 5.3. A simple algorithm (algorithm 1) can be used to compact the heap by computing the shift value for full/free lists and perform parallel shift operations on the independent lists followed by an update of the virtual memory with new pointer values.

Algorithm 1 Heap Management

```
Virtual Memory Status =  $V_{STAT}$   $\triangleright$  Free/Full status of a MAU (figure 5.4).  
Virtual Memory Pointer =  $V_{PTR}$   $\triangleright$  Pointer to MAU/frame in physical memory (figure 5.4).  
Free Pointer =  $F_P$   $\triangleright$  First Free MAU in Free list (figure 5.3).  
Translator Base Registers =  $TBR$   $\triangleright$  Page Table Base Register for each AP in Translator.  
if Request == Allocate then  
  if  $R_V == 1$  then  
     $V_{STAT} \leftarrow Update$   
     $F_P \leftarrow Update$   
     $TBR \leftarrow Update$   
    Return  $ACK$   
  end if  
else if Request == Deallocate then  
   $V_{STAT} \leftarrow Update$   
   $V_{PTR} \leftarrow Update$   
   $F_P \leftarrow Update$   
  Return  $ACK$   
end if
```

5.1.1 Reverse Butterfly Structure

Each pointer consists of the status (free/full) of the MAU along with its physical address (figure 5.4), hence a single shifter is required to shift $N_B \times \log_2(N_B)$ bits. The shifting can be accomplished using *SRL* structures on the FPGA but their time complexity scales with $O(N_B)$, alternatively a single cycle shift is possible using *Barrel Shifters* but requires an additional cycle (step) to perform concatenation of non-shifted pointers (figure 5.5). In contrast, a reverse-butterfly structure [12] is capable of shifting multiple blocks (group of full/free MAUs) of elements in a single cycle. Therefore this work utilizes the reverse butterfly structure to perform coalescing of free list / compaction of virtual memory, resulting in low latency (de)allocation mechanism.

To understand how the reverse-butterfly (figure 5.6) is capable of shifting blocks of elements smoothly without congestion, i.e., collision of data through the network, the following two properties *Even-Odd Trees* and *Scatter Distance* (S_d) must be established.

Even-Odd Trees

A (reverse) butterfly network may be seen as a clever arrangement of two recurring tree structures as shown in figure 5.6, one CBT responsible for switching all packets with even destination and the other for odd destinations. For example, consider a packet wishes to traverse to destination 4'' from source node 1 using DTR:

$$\pi(1) = 4'' = 3'b100(\text{in binary})$$

The packet $\pi(x) = 4''$ will traverse the structure only through the even nodes, irrespective of its source node x . This suggests any given set of destination addresses can be split into two independent sets, even (green) and odd (blue), respectively.

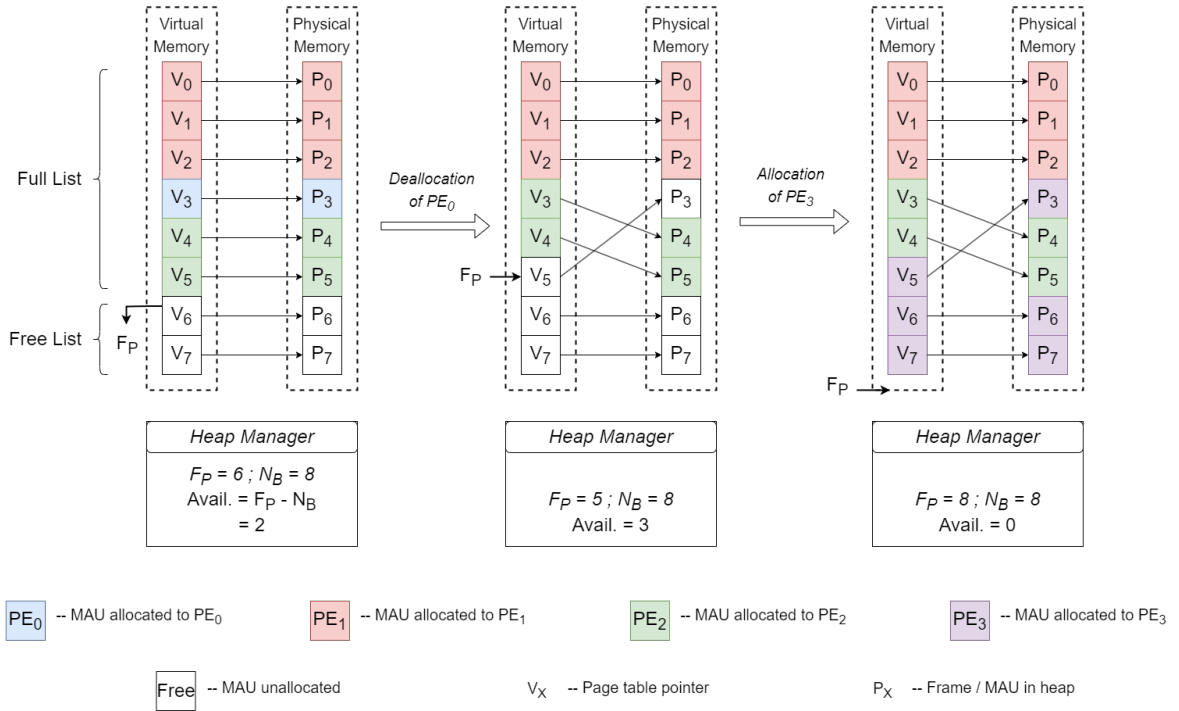


Figure 5.3: Heap manager operation overview: Management of virtual memory map by *Request Validation* for allocation requests and *Coalescing* post de-allocation request.

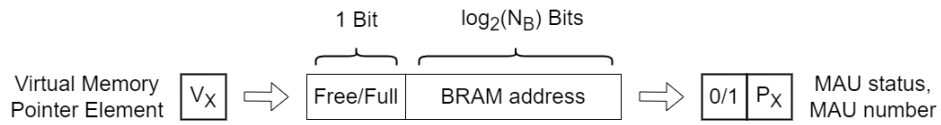


Figure 5.4: Breakdown of a single *Virtual Memory* element into *Status* and *Frame Pointer* flags.

Scatter Distance

A conflict in routing arises when two packets wish to travel through the same data-path when exiting a stage as shown in figure 5.6 (b). In stage 2, packet $\pi(0)$ (red) and $\pi(2)$ (green) have a conflict since they both would like to travel through the left path, resulting in congestion as only one packet may traverse through a path at any given time. On the other hand, a packet $\pi(4)$ with the same destination as $\pi(2)$, observes no conflict during its travel, suggesting conflicts are a function of both source and destination values.

The *Scatter Distance* S_d is a stage relative measure of how far apart the sources (launch nodes) of two packets $\pi(a)$ and $\pi(b)$ should be in order to avoid collision. *Scatter Distance* is also a function of both source and destination addresses and can be computed by counting the number of stages a packet $\pi(a)$ shares with packet $\pi(b)$ as follows:

Let $\pi(a) = \pi(0)$ and $\pi(b) = \pi(2)$.

Let $RMB(x)$ be a function that returns the bit-position of the rightmost '1' of x .

Hence from the perspective of Stage 1:

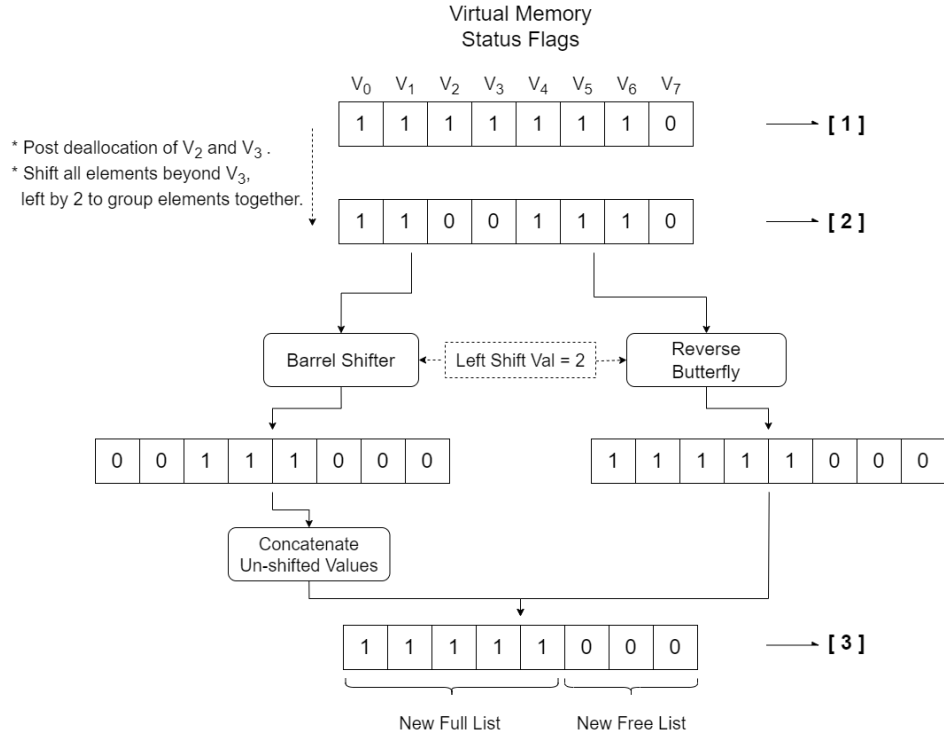


Figure 5.5: Shown is the coalescing of the new fragmented full/free list [2], generated from [1] by deallocation of elements V₂ and V₃, re-grouped into an unfragmented free/full list [3] using the *Barrel Shifter* and *Reverse Butterfly* structure.

$$\begin{aligned}\pi(0) &= 0'' = 3'b000 \\ \pi(2) &= 4'' = 3'b100 \\ S_d &= RMB(0'' \wedge 4'') = 2\end{aligned}$$

Therefore, to avoid collision between two packets $\pi(a)$ and $\pi(b)$ whose destinations are 0'' and 4'', respectively, the source nodes must be spread apart by a distance of 2 or greater. That is, if $\pi(a)$ is launched from switch 0 (path in red), then $\pi(b)$ must be launched from switch 2 (path in blue) or higher as shown in figure 5.6 (b), to avoid routing conflicts during DTR. Otherwise, there will occur a conflict as shown by the green path, if S_d criteria is not met.

The holes (external fragmentation) created in virtual memory due to deallocations results in the movement of a group of 1's to the left as shown in figure 5.5. Each element (1) has a unique destination and the blocks do not overlap each other, ensuring the scatter distance for each element is satisfied and shifting of elements may take place through DTR without conflicts.

5.1.2 Access-Points Base Register

Every time the free list is coalesced, the full list also changes resulting in new virtual memory locations for the still allocated elements. Hence a register to hold the location of the beginning

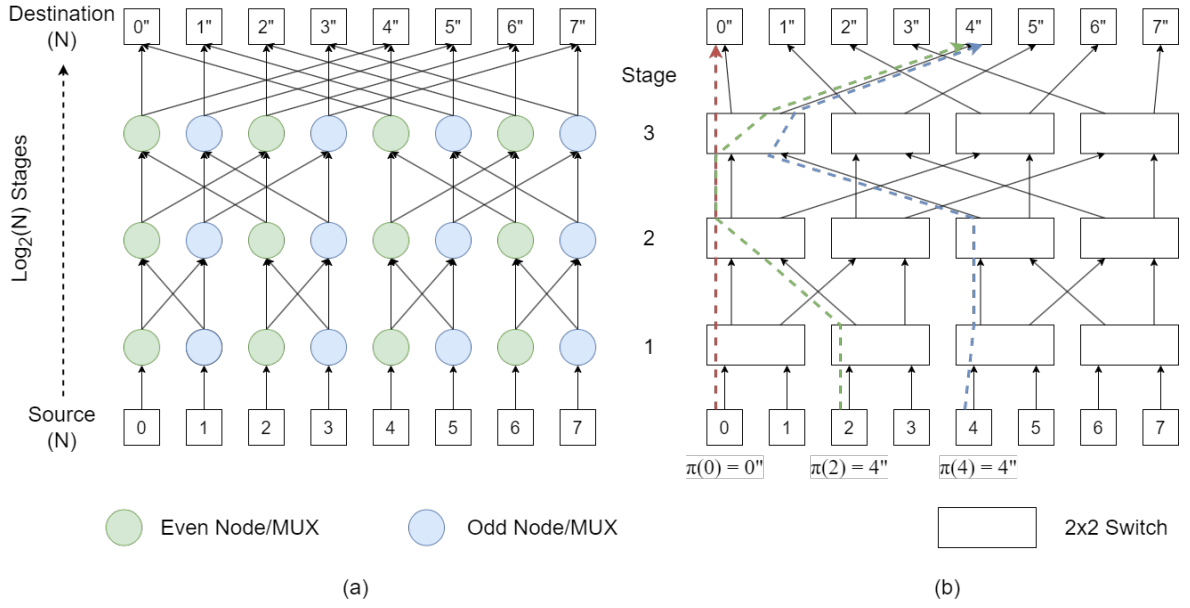


Figure 5.6: (a) Reverse Butterfly structure for 8 elements with even and odd nodes, each encasing a single MUX. (b) Hardware Implementation of reverse butterfly using 2×2 switches each encasing 2 MUXs.

of a set of allocated MAUs in virtual memory is required. The *Access-Point Base Register* (figure 5.1) similar to the *Page Table Base Register* (figure 5.2) helps keep track of the base address of a virtual block of memory. Each AP has an individual base register to help the *Translator* module select the appropriate pointer value from virtual memory.

5.2 Interconnect

The interconnect bridges the APs from *Translator* with the heap (BRAM ports) memory. In chapter 4 the *Benes Network* is chosen as the preferred interconnect due to ability to provide multiple parallel connections with relatively low latency and exhibits a smaller resource scaling factor. The Benes Network (figure 5.7) (a) is a combination structure consisting of two phases A and B, respectively.

Phase A is a butterfly network, responsible for scattering the source node packets to appropriate input nodes of phase B to ensure the distance between two neighbouring packets satisfies the *Scattering Distance* S_d . Any two packets $\pi(a)$ and $\pi(b)$ are neighbours if their destinations share the same output switch, shown in figure 5.7 (a) are two neighbouring packets $\pi(1)$ and $\pi(2)$ with destinations 0'' and 4'', respectively. Phase B is a reverse butterfly network which can route packets without conflicts using DTR, provided stage A scattered the neighbouring packets to appropriately.

A Benes network is a recursive *Clos* network [36] (figure 5.7) with $N = 2^n$ inputs and outputs [16]. The Benes interconnect is sought after in parallel computing, multi-processor systems and Network-On-Chip (NoC) due to its relatively reduced construction complexity, high throughput and low latency switching capabilities [14]. Although the Benes network provides better scalability and re-arrangeably non-blocking connections to any input/output

permutation as described in chapter 4, it adds an additional layer of complexity in terms of setting up the routes from source to destination in bounded time, i.e., maintain uniform latency and throughput.

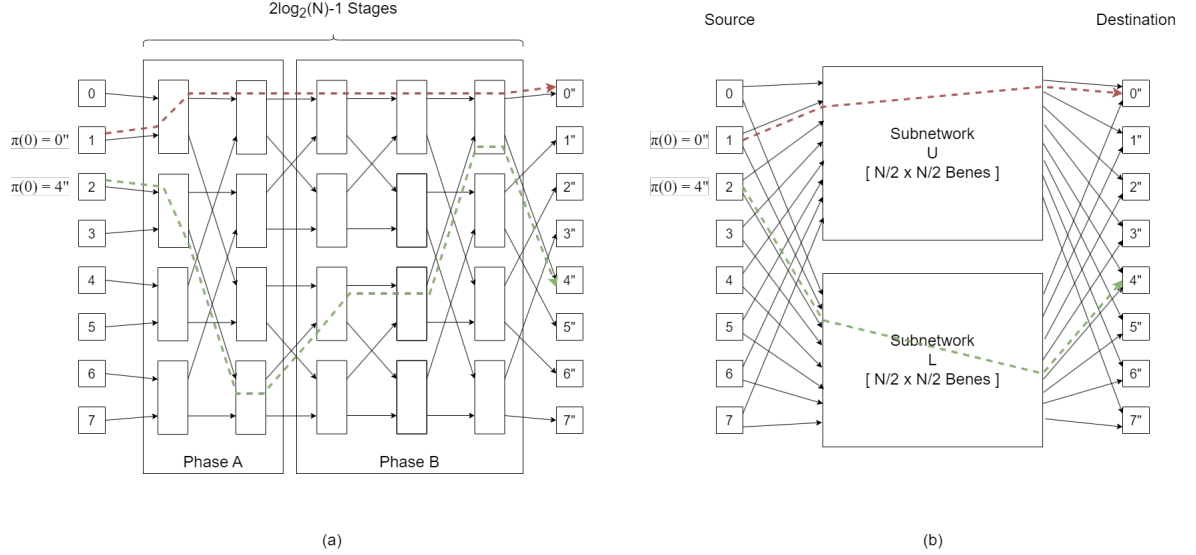


Figure 5.7: (a) Benes network for $N = 8$ inputs and outputs. (b) Recursive Clos network representation of a Benes network for $N = 8$. Neighbouring packets $\pi(1)$ and $\pi(2)$ are routed through separate subnetworks U and L in the first stage to avoid collision in phase B .

5.3 Mapper

To cater for accelerator applications where bounded latency values are preferred, requires the implementation of a dedicated hardware module capable of setting up routes from APs to BRAM ports, through the Benes interconnect, with minimum latency and overhead. The *Mapper* is responsible for computing the Switch-Control-Bit (SCB) of each 2×2 switch in the Benes interconnect for any given permutation of inputs and outputs, with minimum latency. Since phase B utilizes DTR, the computation overhead for phase B is minimal and can be done in $O(1)$ time, i.e., in a single cycle. Therefor the latency of the Mapper is predominantly determined by the underlying algorithm used to compute a non-conflicting set of SCBs for phase A switches in Benes interconnect.

5.3.1 Parallel Routing Algorithm

The setup (route determination) time of the Mapper must be minimal to quickly establish connection between APs and BRAM ports, upon a connection request from the *Translator* module. This eliminates sequential algorithms such as the looping algorithm [27], which solves the permutation network $\pi(n_i) : n_j ; n_{i,j} \in N$, where N is the number of input and output ports. The looping mechanism achieves a time complexity of $O(N)$ on a single BPE which does not scale well for larger N , suggesting the search for a parallel algorithm.

A parallel approach using multiple *Completely-Interconnected* BPEs in [25] achieves a time complexity $O((\log_2(N))^2)$ but suffers from the inability to route partial permutations, i.e., the algorithm suspends processing when an idle connection is encountered. During DMA multiple APs may be idle depending on their allocation status in heap. Therefore the routing algorithm must be capable of handling partial permutation. Lee and Liew [22] introduce a parallel algorithm capable of handling both full and partial connection permutation in $O((\log_2(N))^2)$ time complexity.

Lee's algorithm has been implemented by Jiang and Yang [17] on an FPGA with a centralized control unit for $N = 8 - 32$. Preceding Jiang and Yang, Kai et al. [18] implement Lee's algorithm on the FPGA using a distributed approach which reduces the hardware complexity from $O(N^2)$ to $O(N(\log_2(N))^2)$ [19]. Clearly, a distributed approach of the control path scales better for growing N . This work's implementation is also based on Lee's algorithm, supports both full and partial permutations and exhibits a distributed control structure across multiple BPEs.

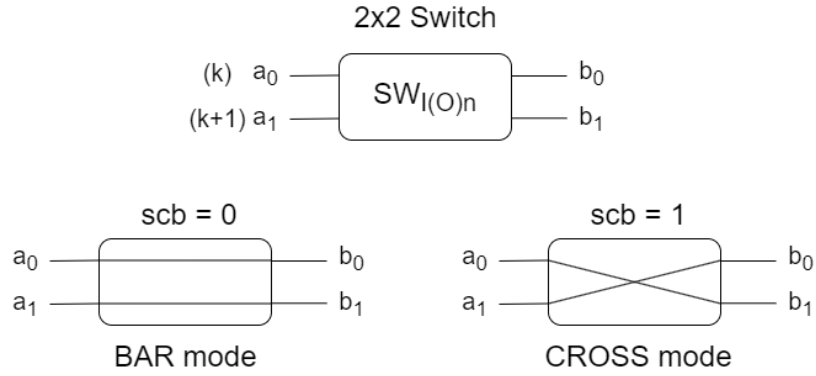


Figure 5.8: A 2×2 crossbar switch $SW_{I(O)nk}$. The switch $SW_{I(O)nk}$ can take the value $scb \in 0, 1$ to represent bar and cross, respectively.

5.3.2 Chain Representation

A Benes network is re-arrangeably non-blocking, due to the inter-dependency between existing connections and new connections. The dependency between inputs and outputs can be studied by representing the network permutation as a 2-colourable *Bi-partite* graph [22]. Consider the permutation π from inputs n_i to outputs n_j , where $i = 0, 1, \dots, N$ and $j = 0'', 1'', \dots, N''$:

$$\pi = \begin{pmatrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\ 0'' & 3'' & 2'' & 6'' & 4'' & 7'' & 5'' & X'' \end{pmatrix} \quad (5.1)$$

Each 2×2 crossbar switch (figure 5.8) can handle two inputs/outputs k , resulting in two sets of $N/2$ distinct nodes (switches) $SW_{I(O)nk}$; $n \in N/2$, $k \in 0, 1$ for inputs and outputs, respectively. Every $i, j \in N$ is uniquely mapped to a switch's k inputs. Hence equation (5.1) can be reinterpreted in terms of $SW_{I(O)nk}$:

$$\pi = \begin{pmatrix} SW_{I_{00}} & SW_{I_{01}} & SW_{I_{10}} & SW_{I_{11}} & SW_{I_{20}} & SW_{I_{21}} & SW_{I_{30}} & SW_{I_{31}} \\ SW_{O_{00}} & SW_{O_{11}} & SW_{O_{10}} & SW_{O_{30}} & SW_{O_{20}} & SW_{O_{31}} & SW_{O_{21}} & X'' \end{pmatrix} \quad (5.2)$$

If each switch $SW_{I(j)n_k} \in N/2 \leftarrow i(j), i+1(j+1) \in N$ and can only take the value $scb \in 0, 1$ and each input/output from a switch go/arrive through/from different subnetworks (figure 5.7.(b)), the connections between the two sets of switches form a bi-partite graph where every node n sharing a common node n_c is the neighbour n_a of n . The bi-partite graph can be solved by alternatively colouring the edges in the same *chain* (class in [22]) using 2 distinct colours (figure 5.9) representing the two subnetworks U and L .

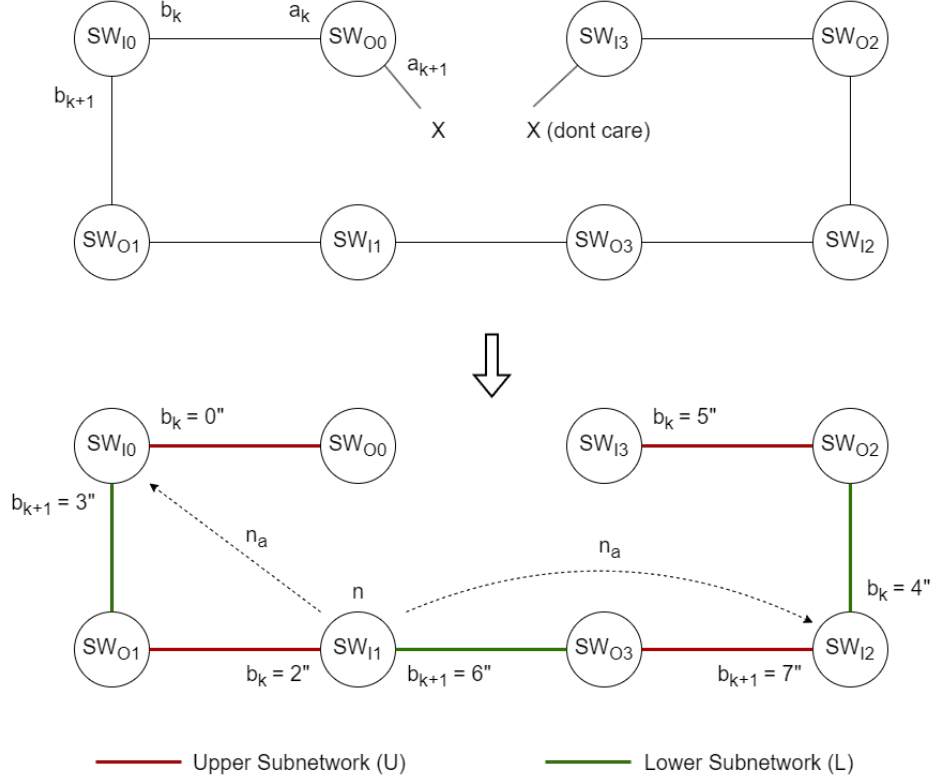


Figure 5.9: Bi-partite graph of equation (5.2) solved by 2 edge colours for upper and lower subnetworks. If node SW_{I_1} is n , then its neighbours n_a are SW_{I_0} and SW_{I_2} . Each b_k, b_{k+1} value corresponds to the destination addresses of a switch's inputs $k, k+1$.

If the graph is represented using a single set of switches SW_I/SW_O , then the adjacent switches can be related by their scb values using *equal* $=$ and *not-equal* \neq relationships (figure 5.10). If a switch has $scb = 0$, the switch is in *bar* mode resulting in b_k, b_{k+1} outputs connected to subnetwork U and L , respectively. The condition that two neighbouring nodes A and B cannot travel through the same subnetwork yields a scb relationship between them as follows:

$$A_{scb} = \begin{cases} B_{scb}, & \text{if Edge } A_{b_k} = B_{b_{k+1}} \\ \sim B_{scb}, & \text{otherwise} \end{cases}$$

Hence, the permutation (equation (5.2)) may be reduced to obtain set of boolean equations in terms of $SW_{I_n}; n \in N/2$:

$$SW_{I_0} = SW_{I_1} \neq SW_{I_2} \neq SW_{I_3} \quad (5.3)$$

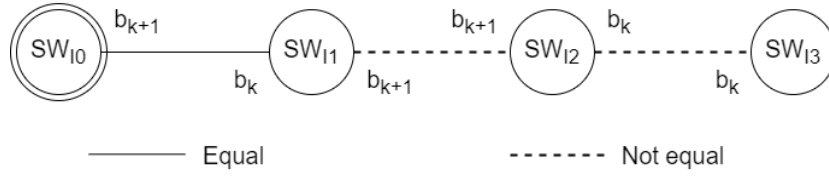


Figure 5.10: Graphical representation of node's *scb* relationship with its neighbours. The chain leader is the node with the smallest subscript, marked with double circles.

Each unique set of boolean equations as in equation (5.3) is considered a chain and a stage in Benes network with N inputs/outputs can have upto $N/2$ chains. If the switch with the smallest subscript is taken to be the chain leader (class representative in [22]), then setting the leader's *scb* will set all the other switches through their chain's relationship.

5.3.3 Control Architecture

For N inputs, $N/2$ BPEs are required, with each BPE handling two inputs. The number of chains can range from 0 to $N/2$, hence it is possible each BPE node could be the leader and *associated* with its own chain. Therefore, each chain is enumerated with respect to its leader BPE, i.e., chain 0's leader will be BPE 0 and so on. When two neighbouring BPE's have different chain associations, a collision/conflict will occur, in which the chain with the lower number will absorb the other. As a result, the losing chain's BPEs will change their association to the new chain and follow the new leader (BPE) as shown in figure 5.11.

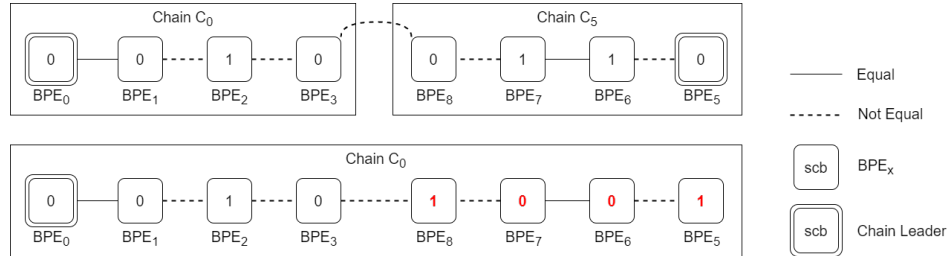


Figure 5.11: Chain C_5 with leader BPE_5 merges with C_0 upon discovery by BPE_8 . BPE_8 and BPE_3 are neighbours, whose relationship affects all the *scb* (shown in red) in (previously) C_5 's BPEs.

During merging of two chains C_0 and C_5 , a change in *scb* value of the losing BPE in chain C_5 causes a cascade effect on all *scb* of chain C_5 . This suggests, all the *scb* in C_5 can be flipped in parallel, as long as all the BPEs in C_5 are notified of a merge. Hence, each BPE must be capable of notifying its chain leader about the discovery of a new chain, and receive information from its chain leader about *scb* flipping and switch of chain leaders.

An algorithm 2 is developed to find the neighbours of a BPE and establish their boolean relationship (equation (5.3)) in $O(1)$ complexity during the *initialization* step, followed by chain discovery (or) *searching* and chain *merging* simultaneously in $O(\log_2(N))$ time per stage of the Benes network. A supporting hardware architecture as shown in (fig mapper arch) is realized with a centralized memory space (register files) to share chain information between members of the chain and their leaders, while keeping the decision making distributed across all BPEs.

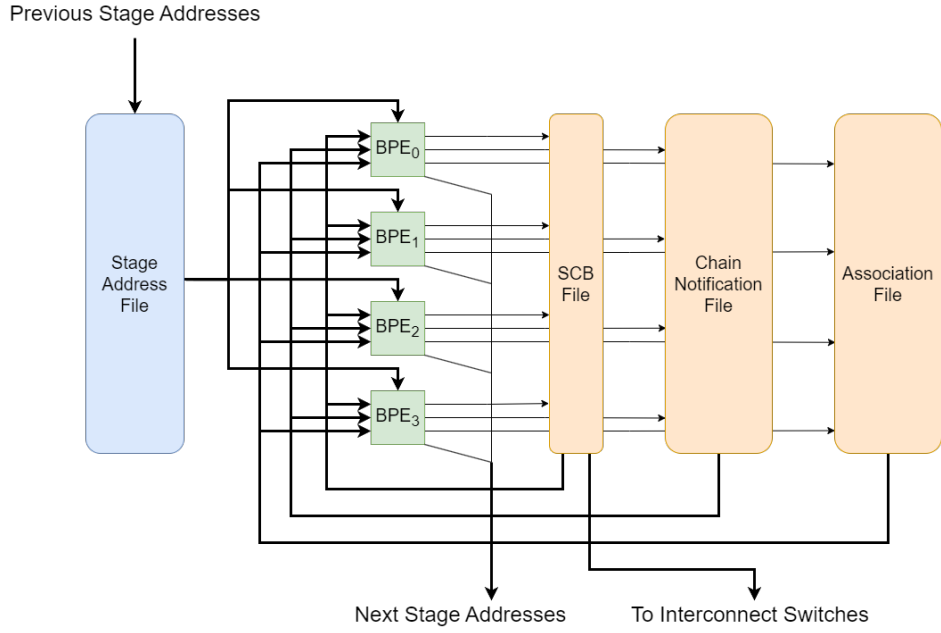


Figure 5.12: Architectural view of a single stage of Benes Mapper in phase A for $N = 8$. Thick lines indicate bundles/buses used to transfer stage information to all BPEs.

Shown in figure 5.12 is an arrangement of BPEs and register file, in a single stage of the mapper. The mapper mimics the Benes network, hence consists $\log_2(N) - 1$ stages in phase A (figure 5.7), with each BPE handling the controls of a single switch in the interconnect. Each BPE consists of three elements, *Collision Monitor*, *Chain Monitor* and *Association Manager*, namely.

- The **Collision Monitor** discovers BPEs through the shared stage addresses to establish *scb* relationships (figure 5.10) with its neighbours and obtain neighbouring node's association numbers ($n_{n_{a0(1)}}$) following (1) in algorithm 2. The *Collision Monitor* accepts inputs from the *Stage Address File* (to find neighbours) and *Association File* (to monitor neighbour's current chain). The neighbour relationship are stored local to every BPE while the *Association Number* (AN) is stored the *Association File* for easy distribution.
- The **Chain Monitor** is responsible for monitoring all notifications through the shared *Chain Notification File* and asserting a chain change if the notification is valid ((3) in algorithm 2). Each BPE has its own *Chain Monitor* and is independent of other BPEs during decision making.
- The **Association Manager** controls the generation of notifications based on the decisions of the *Collision Monitor* and *Chain Monitor* following (2) in algorithm 2.

Upon collision, the *Collision Monitor* will pass on the colliding chain's (AN and current chain *scb* flip) information to the *Association Manager*. Simultaneously, the *scb* of the base node will be updated using the colliding neighbour's node relationship. The *Association Manager* will generate a notification packet P , encasing *current AN*, *new AN* and *flip scb* flags and push the notification to the *Chain Notification File*. The *Chain Monitor* actively monitors the *Chain Notification File* for packets with *current AN* = n_{b_a} ; a match indicates the current chain the BPE belongs to (or) associated with (n_{b_a}) has encountered another chain and is being absorbed into the new chain. The *Chain Monitor* will pass on the *new AN* and *flip scb* flags to the respective

internal modules for updates. Doing so, the BPE will successfully switch chains without causing conflicts.

Each part *Collision Monitor*, *Chain Monitor* and *Association Manager* of the algorithm 2 runs in $O(1)$ time (single cycle) and the whole algorithm is repeated $\log_2(N_s)$ times to completely reduce all redundant chains, where N_s is the number of inputs/permutations per stage s . Each succeeding stage has the permutations cut in half $N_1 \leftarrow N_0/2$ and there are $\log_2(N_0) - 1$ stages in phase A. The computation of phase B is done in $O(1)$ complexity using DTR, hence the time complexity of the mapper is $O((\log_2(N_0))^2)$ and total time/cycles required to set all switches from input to outputs is:

$$\text{Time Phase A} + \text{Time Phase B} + \text{Stage Initialization} = \sum_{n=1}^{\log_2(N_0)} n + \log_2(N_0) \quad (5.4)$$

The major contributor towards time/cost complexity is *phase A* of the mapper due to its execution complexity. The internals of a BPE from *phase A*, encompassing *Collision Monitor*, *Chain Monitor* and *Association Manager* is shown in figure 5.13. In contrast, a BPE from *phase B* performs DTR and consists of a simple 2×2 switch with added auxiliary circuitry to extract stage address and store the *scb* value.

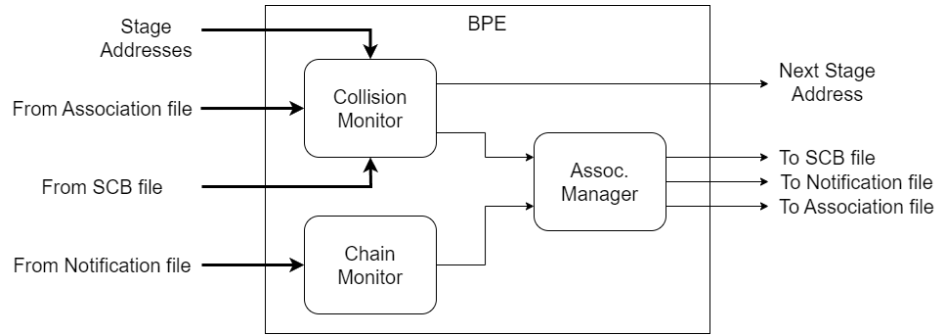


Figure 5.13: Internal view of a BPE. Thick lines indicate buses.

5.4 Heap and Translator

The *Heap* houses the collective of memory units (MAUs) that can be dynamically allocated by the *Heap Manager*. The MAU units in this instance are BRAMs instantiated using Vivado's 7series *BRAM_TDP_MACRO*, with each MAU providing True-Dual-Port (TDP) operation.

The *Translator* sits between the PL and interconnect, housing the APs. The translator is responsible for converting the incoming virtual address into physical address by using information from *Access Port Base Register* and *Page Table* (figure 5.1) in *Heap Manager*. The transformed virtual address (physical address) is asserted onto the *Mapper* inputs to establish a connection between the AP and BRAM port in the heap.

Each AP has a fixed window size of two to support dual channels for TDP operation. This is equivalent to assigning an individual BPE to an AP since each BPE can control/switch two channels simultaneously. Therefore, for N ports in the *Mapper*, there are $N/2$ APs in the

translator and $N/2$ TDP memory in the heap. Furthermore, an AP window can be operated as a sliding window (figure 5.14.(b)) to mask the port switching latency (equation (5.4)) introduced by the *Mapper*, enhancing the throughput of streaming or any application with consecutive memory access patterns.

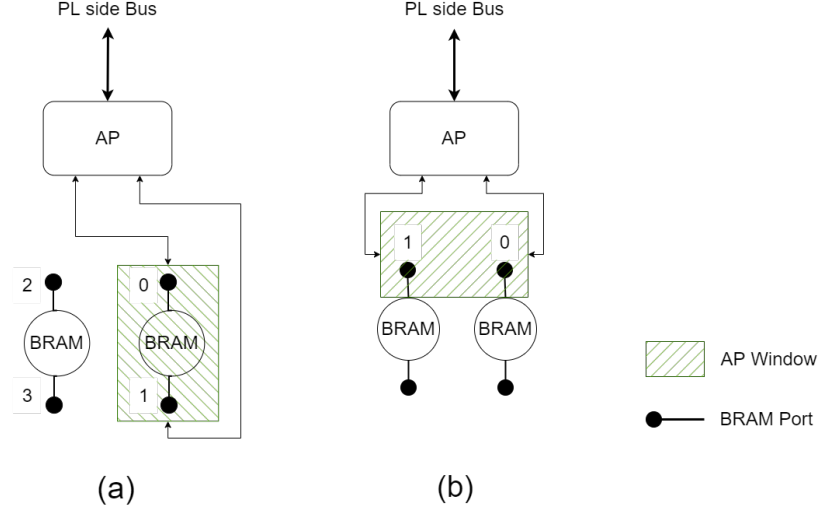


Figure 5.14: (a) AP window configured to handle a single BRAM in dual-port mode. (b) AP window configured as a sliding window to handle two BRAMs simultaneously. The numbers next to the BRAM ports indicate the physical address used by the *Mapper* to tag memory ports

5.5 Optimization

Scenarios where N APs require simultaneous connection to N memory ports is scarce. Most situations involve a few APs accessing a large set of BRAMs from heap. Hence the *Mapper* can be optimized to support upto $N/2$ ingress ports for N egress ports. The majority construction/time cost of the *Mapper* sprouts from *phase A* of the mapper (section 5.3.3), suggesting maximum impact of optimization can be realized by optimizing *phase A* of the mapper. Shown in table 5.1 are the resource consumption and switching/*Inter-BRAM Translation* latency of the mapper for different number of APs and dual-port (DP) BRAMs in heap, when implemented on the *PYNQ-Z2* FPGA board housing the *ZYNQ7020* SoC.

AP	$BRAM_{DP}$	LUT	FF	$f_{max} (MHz)$	$Switch Latency (min \sim max)$
4	4	299	142	140	9 ~ 12
8	8	1299	366	125	14 ~ 18
16	16	5201	898	115	20 ~ 25
32	32	19124	1911	115	27 ~ 33

Table 5.1: Resource consumption and Switching Latency for different AP-BRAM configurations.

The first optimization step reduces the number of APs supported by half to change the AP-BRAM configuration from $N - N$ to $N/2 - N$, where N is the number of dual-port BRAM in

heap. The second optimization stems from the observation of an AP's window. If the window is framed to two modes of operation, *Stream-Port* (SP) and *Dual-Port* (DP), as shown in figure 5.14. The window's channels will always take even and odd values, respectively. Hence, applying the *Even-Odd Tree* (section 5.1.1) concept to *phase A* of the mapper, the set of ingress inputs N to the mapper is split into two independent sets of size $N/2$. Therefore, the complexity and switching latency of *phase A* can be further reduced by half, resulting in the optimized metrics shown in table 5.2.

AP	$BRAM_{DP}$	LUT	FF	$f_{max} (MHz)$	$Switch Latency (min \sim max)$
4	8	166	98	130	3 ~ 5
8	16	813	422	130	6 ~ 9
16	32	4246	1447	116	10 ~ 14
32	64	19316	4328	100	15 ~ 20

Table 5.2: Optimized Resource consumption and Switching Latency for different AP-BRAM configurations.

The optimization comes at the cost of losing the number of APs supported by the *Translator*, i.e., a reduction in the number of parallel applications supported. But, when comparing the performance for connecting 32 BRAMs in dual-port mode, a reduction of 77% for LUT consumption and 57% for latency is observed, promoting the reason for optimization.

5.6 Operation Modes

The inherent disadvantage of introducing an interconnect such as the Benes network, is the additional latency (switching) introduced during the translation process. But, this is a necessary cost to improve the scalability and concurrency (parallel access) of the VDMMU system. The needs and memory access patterns of different applications (accelerators) may vary widely, demanding flexibility from the VDMMU. Therefore, the current design of the VDMMU can be implemented in one of two operation modes, *Stream-Port* (SP) and *Dual-Port* (DP) modes:

5.6.1 Stream-Port

Streaming applications that utilize BRAMs as FIFOs or PIPOs, applications with reuse of ordered (arrays) data from BRAMs, or applications with sequential memory access patterns, can be positively impacted by implementing the VDMMU in SP mode. In SP mode, an AP's window spans across two BRAM ports resulting in Single-Port configuration. An application can simultaneously *read and/or write* to the same address or, perform a *read or write* to different addresses. Assuming an application exhibits sequential memory access pattern for the majority of its dynamically allocated memory.

Then, the switching latency introduced by the Benes network can be completely masked by pre-switching one of the window channels to the next consecutive BRAM port, i.e., inter-BRAM translation latency (section 4.1) is completely eliminated and only inter-BRAM translation latency exists. The previous statement holds true only for sequential memory access patterns and does not hold when the application attempts to access non-consecutive BRAM ports.

Since the APs are only utilizing one of the BRAM's ports, the other port can be used to directly interface with PS, with secondary memory through *Direct-Memory-Access* or with other PL modules. The illustration in figure 5.1 shows the SP implementation of VDMMU, in which the PS can directly access the heap.

5.6.2 Dual-Port

If the need to support two independent channels, i.e., perform simultaneous *read and/or write* to the same BRAM, by an AP outweighs the switching latency cost presented by the Benes network. Then, the VDMMU may be implemented in DP operation mode. If the overhead of masking the switching latency is transferred to the PL application, DP mode can emulate SP mode's latency masking.

An application can mask the latency by ensuring its assigned AP's two channels are pointing to two virtually consecutive BRAMs, i.e., locations consecutive in memory access. By doing so, the application can simply switch channels to access different memory locations, eliminating inter-BRAM translation latency. Furthermore, the DP mode of operation has both BRAM ports tied to the Benes network.

Hence in contrast to SP mode, DP mode cannot bypass the Benes network and connect to other peripherals such as the PS directly. But, DP mode provides the advantage of higher data bandwidth during read/write operations due to dual-port access to the BRAM, making it a suitable candidate for high-bandwidth applications.

5.7 Parameterization

The VDMMU is designed with flexibility in mind to support different applications. To this extent, a VDMMU implementation provides the following user configurable parameters:

1. **Heap Size:** Controls the number of BRAMs N_B that can be dynamically allocated from the heap.
2. **Translators:** Controls the number of APs VDMMU provides for simultaneous access of the heap. The number of translators (N_T) and heap size (N_B) are related by $N_T \leq N_B/2$.
3. **Data Width:** Defines the bus-width of the BRAM's data ports.
4. **Offset:** Defines the bus-width of the BRAM's address ports.
5. **BRAM Size:** Defines the type of BRAMs instantiated in the heap. Valid values for this parameter are "18Kb" or "36Kb".
6. **Buffer Write:** Registers all data/address lines going to the heap for better timing. Valid values are "1"/"0" to enable/disable the registers.
7. **Buffer Read:** Registers all data lines going to the PL from heap for better timing. Valid values are "1"/"0" to enable/disable the registers.
8. **Buffer Translator:** Register all signals between the *Translator* and *Mapper* to improve timing, but introduces additional switching latency.

Algorithm 2 Benes Processing Element

This node $n_b \leftarrow no_k; k \in N/2$
This node's Association (chain) $n_{b_a} \leftarrow b$
This node's scb $n_{b_{scb}} \leftarrow 0$
This node's inputs $n_{b_0}, n_{b_1} \leftarrow i, j; i, j \in N$
Relationship with Neighbours $n_{n_{r0}}, n_{n_{r1}} \leftarrow 0$
Neighbour's association $n_{n_{a0}}, n_{n_{a1}} \leftarrow 0$
Neighbour's scb $n_{n_{scb0}}, n_{n_{scb1}} \leftarrow 0$
Notification $P[3] : [current\ assoc., new\ assoc., flip\ scb]$

Require: $i \neq j$

(1) *Initializer/Collision Monitor*: \triangleright Initialize Base node, set neighbour relationships and assoc.

for Every $k \in N/2$ **do**

if $k \neq b$ **then**

\triangleright Set relationship for n_{b_0}

if $n_{b_0} \wedge no_{k_0} == 1$ **then**

$n_{n_{r0}} \leftarrow 1$

$n_{n_{a0}} \leftarrow k$

else if $n_{b_0} \wedge no_{k_1} == 1$ **then**

$n_{n_{r0}} \leftarrow 0$

$n_{n_{a0}} \leftarrow k$

end if

\triangleright Set relationship for n_{b_1}

if $n_{b_1} \wedge no_{k_0} == 1$ **then**

$n_{n_{r1}} \leftarrow 0$

$n_{n_{a1}} \leftarrow k$

else if $n_{b_1} \wedge no_{k_1} == 1$ **then**

$n_{n_{r1}} \leftarrow 1$

$n_{n_{a1}} \leftarrow k$

end if

end if

end for

(2) *Association Manager*:

\triangleright Monitor neighbouring nodes for chain collision

for Every $m \in 0, 1$ **do**

if $n_{n_{a0}} < n_{b_a}$ **then**

$P[0] \leftarrow n_{b_a}; P[1] \leftarrow n_{n_{a0}}; P[2] \leftarrow n_{b_{scb}} \wedge n_{n_{scb0}} \wedge n_{n_{r0}}$

$n_{b_a} \leftarrow n_{n_{a0}}$

$n_{b_{scb}} \leftarrow n_{n_{scb0}} \wedge n_{n_{r0}}$

else if $n_{n_{a1}} < n_{b_a}$ **then**

$P[0] \leftarrow n_{b_a}; P[1] \leftarrow n_{n_{a1}}; P[2] \leftarrow n_{b_{scb}} \wedge n_{n_{scb1}} \wedge n_{n_{r1}}$

$n_{b_a} \leftarrow n_{n_{a0}}$

$n_{b_{scb}} \leftarrow n_{n_{scb0}} \wedge n_{n_{r0}}$

end if

end for

(3) *Chain Monitor*:

\triangleright Monitor Chain for change notifications

for Every $k \in N/2$ **do**

if $P_k[0] == n_{b_a}$ **then**

$n_{b_a} \leftarrow P_k[1]$

$n_{b_{scb}} \leftarrow n_{b_{scb}} \wedge P_k[2]$

end if

end for

6 Results and Evaluation

The VDMMU uses paging to eliminate *External Fragmentation* (section 2.3.5) while providing low latency bounded (de)allocation time to requesting services (applications). Furthermore, VDMMU provides a scalable solution to concurrent heap access from multiple services. This chapter will present the obtained results, infer the pros/cons of using VDMMU and evaluate the functionality of the system using a software controlled test-structure on the PYNQ-Z2 dev board.

6.1 Heap Manager Cost-Performance

The graph in figure 6.1a shows the LUT consumption of the *Heap Manager*, which is responsible for performing (de)allocations and compacting the free/full list of MAUs to eliminate external fragmentation. All resource utilization metrics have been derived directly from Vivado 2023.2, implemented on the XC7Z020 SoC.

The major contributor to the LUT consumption in the Heap Manager arises from the *Reverse Butterfly* structure and, the number of MAUs the heap manager can handle indicates the total number of frame pointers in page-table (virtual memory) that can be compacted in $O(1)$ time complexity by the reverse butterfly structure. As seen in figure 6.1a, the cost of implementing Heap Manager doubles with the doubling of managed MAU.

In DMA architectures from [28, 10, 30, 40, 23], the heap is synonymous to BRAM and broken into chunks (MAU) for *fine-grained* dynamic allocation. In contrast, VDMMU (this work) and [9] focus on allocating a complete BRAM to a requesting application rather than chunks of memory from a BRAM. In other words, VDMMU and [9] perform *coarse-grained* allocation and target applications with large dynamic memory requirements spanning across multiple BRAMs.

Therefore, the cost-performance of coarse-grained *Heap Manager* in VDMMU cannot be quantified by directly comparing the '*managed MAU v. LUT*' count with other fine-grained works. Instead, we quantify the VDMMU's cost-performance in terms of the amount of BRAM resources managed (Heap Footprint) against the total LUT consumed as shown in figure 6.1a. At a heap memory footprint of 50%, the Heap Manager costs $\approx 10\%$ of LUTs. Hence at $\approx 20\%$ LUT consumption, all 100% BRAMs on the XC7Z020 SoC can be managed using 2 heaps.

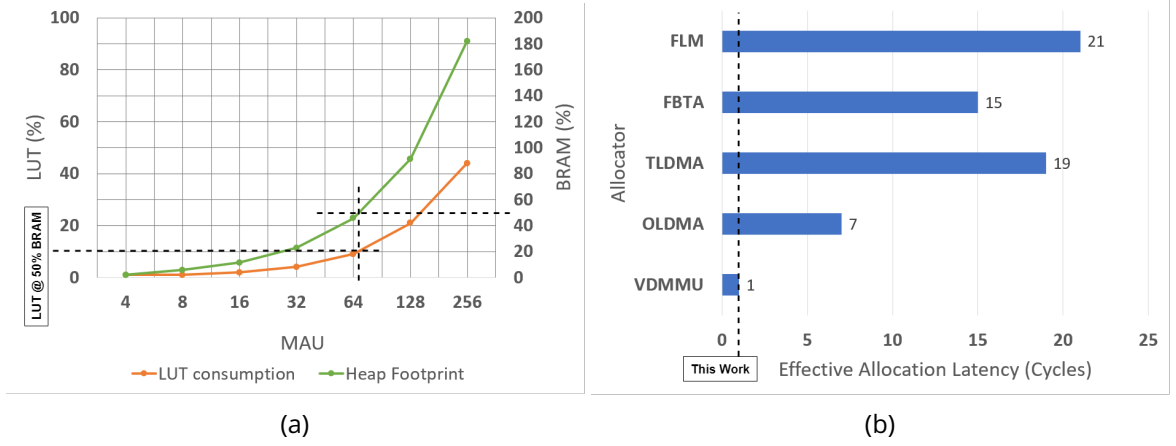


Figure 6.1: (a) LUT consumption (primary y-axis) and total BRAM memory included in heap (secondary y-axis) against the number of MAUs managed by Heap Manager. (b) Comparison of Effective allocation latency Eff_{alloc} @ $I_{req} = 2$, for this work's VDMMU allocator against OLDMA [30], TLDMA [30], FBTA [23] and FLM [28].

6.2 Allocation and Deallocation Latency

Normally, the *First-Fit* algorithm used in VDMMU would incur an operational latency with $O(N)$ complexity with respect to the number of MAUs (N) managed in the heap. But, due to the compaction and rearrangement of the free/full list by the reverse butterfly structure, the first fit algorithm can perform (de)allocations in $O(1)$ time complexity.

The VDMMU has a maximum bounded latency of 2 cycles for allocation, and 3 cycles for deallocation. The major advantage of using the reverse butterfly structure over a traditional barrel/log shifter is the ability of the reverse butterfly to shift multiple blocks simultaneously by differing shift values as long as the conditions stated in section 5.1.1 are satisfied. Hence, VDMMU exhibits support for *burst* (de)allocation with minimum latency.

Real-time applications require bounded latency for allocation process to meet operational deadlines [4], suggesting the system with a smaller allocation latency performs better and supports a wider range of applications/deadlines. To quantify allocation performance (latency) with respect to the allocation request frequency, [30] provides equation (6.1) to compute the *effective allocation latency* (Eff_{alloc}) of a system with respect to *allocation latency* T_{alloc} , *update latency* T_{update} and the *allocation request interval* I_{req} .

$$Eff_{alloc} = 2 \cdot T_{alloc} + T_{update} - I_{req} \quad (6.1)$$

The VDMMU requires a single cycle for T_{alloc}/T_{update} , hence at a request interval I_{req} of 2, the VDMMU achieves an Eff_{alloc} of 1. Shown in figure 6.1b is the effective allocation latency of VDMMU compared with contemporary fine-grained works. Unfortunately [9], a fine-grained paging based allocator, does not provide metrics on allocation latency, but judging by the method of allocation (free-list based) performed in [9], it can be assumed to exhibit an allocation latency in tens of cycles. Therefore, VDMMU exhibits the best performance in terms of allocation latency compared to related work.

6.3 Concurrent Heap Access Cost

The complete resource utilization of VDMMU is dependant on the number of MAU managed, the number of dual-channel APs supported and the data-width of the interconnect. The total number of independent heap access channels supported by VDMMU is twice the number of APs provided. For a data-width of 32 bits, the VDMMU implemented using Benes interconnect achieves $4\times$ reduction in LUT resource consumption compared to an implementation using CBT [9] (generic MUX tree), for 64 MAU. Shown in figure 6.2a is the normalized resource reduction (Benes over CBT) and AP roofline for various MAUs managed.

The resource reduction in implementing multiple parallel access channels to the heap using the Benes interconnect arrives at a cost of increased *Inter-BRAM Translation* latency (section 4.1). The CBT interconnect exhibits a constant translation latency of 1 cycle, while Benes interconnect has a $O((\log_2(N))^2)$ as discussed in section 5.3.3. Shown in figure 6.2b is a plot of inter-BRAM translation latency against the number of channels supported by VDMMU.

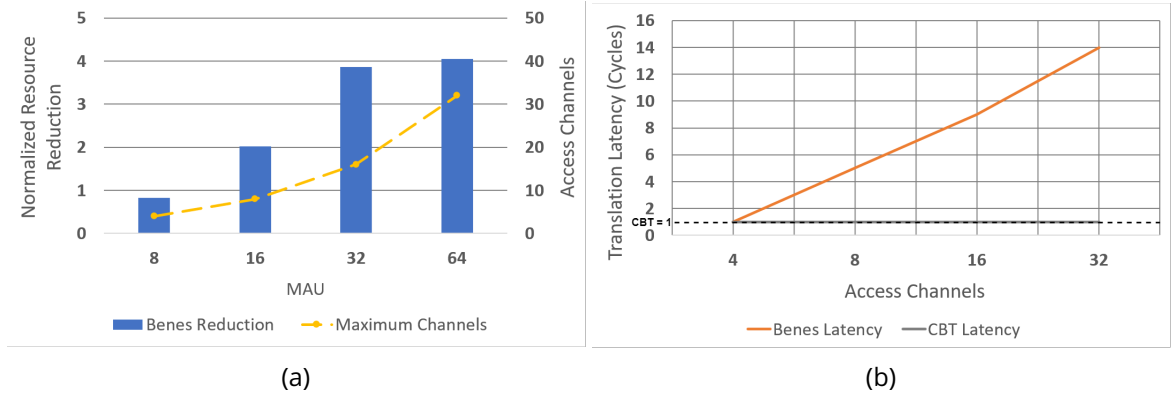


Figure 6.2: (a) Normalized resource reduction achieved by Benes interconnect over CBT; The dashed line represents the maximum number of channels VDMMU can support for a given number of MAU managed. (b) Inter-BRAM Translation latency of Benes interconnect against the number of MAU managed; Baseline latency marked at $CBT = 1$.

6.4 VDMMU Resource Utilization

The VDMMU reduces overall LUT consumption by 41.82% for a heap size of 32 MAU with 16 access channels (figure 6.3b), when compared with DOMMU [9]. Therefore, the Benes interconnect exhibits better scalability, at the expense of added latency.

Majority of the resource utilization is from interconnect/switch (figure 6.3a), followed by translator and heap manager. The interconnects resource utilization is a function of both phase A and phase B of the Benes structure, i.e., the growth is dependant on both the number of BRAMs managed and the number of APs supported.

In addition, the data-width of the switch is a significant factor influencing LUT consumption. Shown in figure 6.3c is the LUT utilization of two sets of VDMMU configuration (BRAM and

channels). One set is when VDMMU can handle 32 bit data transfer between heap and services, and the other is for a 8 bit databus. Clearly, the 8 bit databus can support higher parallel access (upto 32) channels at approximately half the resource consumption of the 32 bit databus version.

The total number of parallel channels with access to the heap is dependent on the total number of MAUs managed. The resource utilization of the heap manager grows significantly with increasing number of MAUs in the heap, suggesting better utilization of resources can be realized using small distributed heaps, each catering to a cluster of accelerators. Although VDMMU boasts scalability in terms of providing multiple access channels to heap, its flexibility is hindered by the fact that the total number of MAU managed and the total number of channels, must be a power of 2.

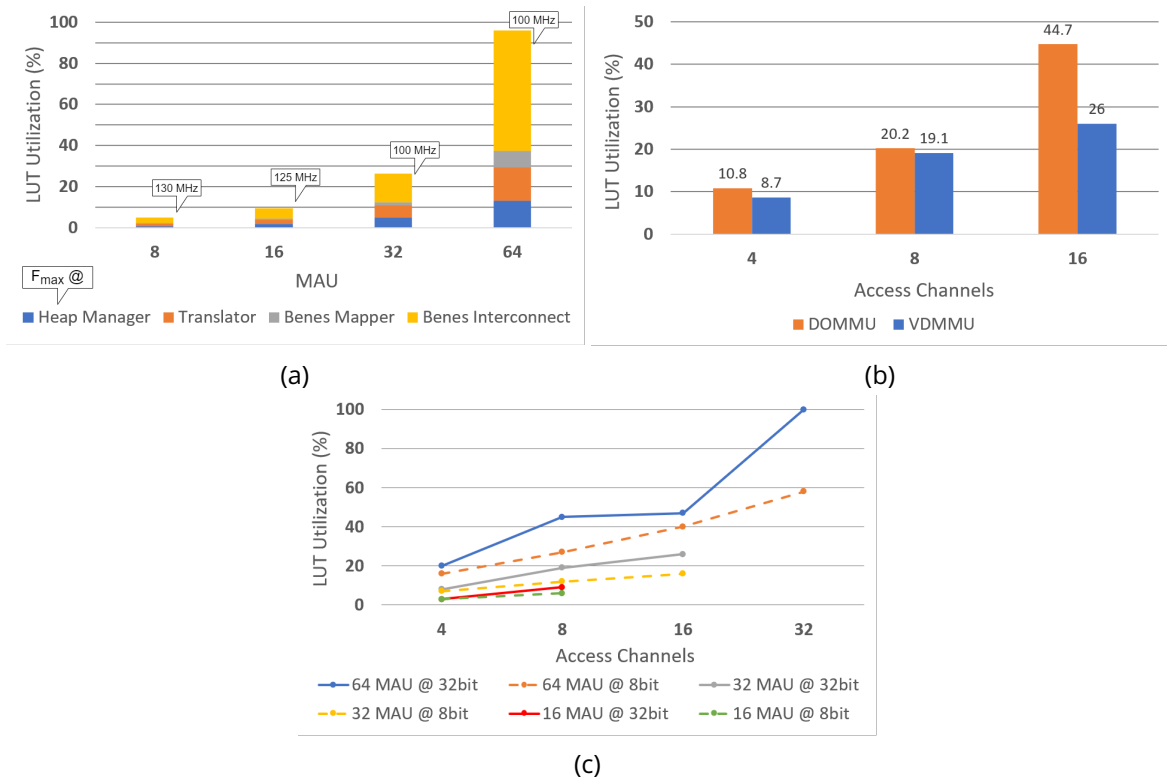


Figure 6.3: (a) LUT utilization breakdown of VDMMU for various number of managed MAU, with roof-line value of channels supported. Maximum operating frequency marked in callout boxes. (b) Comparison between VDMMU and DOMMU [9] LUT consumption against varying number of supported access channels, when managing 32 MAU. (c) Impact of datawidth (8bit/32bit) on VDMMU LUT utilization for different number of access channels and MAU.

6.5 Evaluation Testbench

A simple testbench including an *Integrated-Logic-Analyzer* (ILA) IP is constructed in Vivado 2023.2 as shown in figure 6.4, to test the functionality of a SP mode VDMMU implementation running at 100MHz configured with the parameters show in table 6.2.

DMA Unit	VDMMU	DOMMU* [#] [9]	FLM* [28]	FBTA [23]	SysAlloc [40]	OLDMA [30]	TLDMA [30]
Max LUT (%)	19	20	11	19	8	4	2
F_{max} (MHz)	100	89	175	100	100	100	100
Eff_{alloc} (Cycles)	1	Unknown	21	15	81 ~ 83	7	19
MAU Man-aged	64	40	32	512	32K	512	64K
Parallel Access	<i>YES</i>	<i>YES</i>	<i>NO</i>	<i>NO</i>	<i>NO</i>	<i>NO</i>	<i>NO</i>
External Fragmentation	<i>NO</i>	<i>NO</i>	<i>YES</i>	<i>YES</i>	<i>YES</i>	<i>YES</i>	<i>YES</i>
BRAM Utilized	<i>NO</i>	<i>NO</i>	<i>YES</i>	<i>NO</i>	<i>NO</i>	<i>YES</i>	<i>YES</i>

Table 6.1: Characteristics of various contemporary DMA architectures. Effective Allocation Latency (Eff_{alloc}) is measured at a request interval (I_{req}) of 2. * = LUT consumption from the perspective of XC7Z020 SoC; [#] = LUT and F_{max} adjusted to 32 MAU with 8 access channels.

The validation method involves PS (CPU) managing (de)allocation request to VDMMU and, PL peripherals simulating accelerator applications performing *read/ write* access to their respective allocated heap memory. The accelerator simulators (PL applications) connect to the PS, where the software (PS application) emulates read/write operations to VDMMU, through PL applications.

Two *AXI-Lite* interfaces are designed using *Vivado-HLS* for direct communication between PS and VDMMU. The PS is used to synthetically generate allocation and deallocation requests to the VDMMU. In-addition, the PS also directly writes to the heap using the free port of the heap provided in SP mode, enabling direct data transfer from On-Chip-Memory (OCM) or DDR RAM through PS to heap.

The *DMEM-AXIL* (green bounding box in figure 6.4) provides the AXI-Lite interface for PS (black box) to communicate (de)allocation requests to the VDMMU (red box) using a simple request/reply protocol. Once a valid request is sent, the PS must wait for an *ACK* before

Heap Size	= 16	BRAM Size	= 36Kb
Translators	= 4	Buffer Write	= 0
Data Width	= 8	Buffer Read	= 1
Offset	= 12	Buffer Translator	= 1

Table 6.2: Parameter settings for the evaluated SP mode VDMMU.



Figure 6.5: (a) Allocation requests from PS to VDMMU for PLA_3 , PLA_0 and PLA_1 , respectively. (b) PS writing 4 bytes of `0xdeadd` to BRAM 8 in heap; BRAMs 8 and 9 are allocated to PLA_2 in physical memory (heap) in order to hold 8000 bytes of data since, each BRAM is configured to hold 4096 bytes. (c) PLA_2 reading 4 bytes from BRAM 8 in the heap. (d) Deallocation of PLA_3 from heap and subsequent deactivation of AP_3 in translator (blue bounding box).

7 Conclusion and Future Work

7.1 Conclusion

Many software algorithms depend on dynamic memory management to support variable workloads and efficiently use common memory resources. The realization of accelerators for software kernels using HLS has expedited the process of design and verification significantly, but HLS suffers from pessimistic static allocation of memory resources causing an under-utilization of allocated resources when the average workload is smaller than HLS's pessimistic assumption. Thus, a bottle-neck is presented to the maximum achievable throughput of a multi-accelerator system, due to the quick exhaustion of on-chip memory resources such as BRAMs.

Dynamic memory management in FPGAs is employed to better utilize BRAMs over various workloads, improving the overall memory resource utilization while boosting application throughput by packing more accelerators. This work proposes VDMMU, a DMA architecture with no heap fragmentation and parallel heap access support for multi-accelerator systems. VDMMU achieves $2\times$ and $4\times$ lower LUT consumption using the Benes interconnect for 16 and 64 independent access channels, respectively. However, the improved resource efficiency of VDMMU comes at the cost of added switching (inter-BRAM translation) latency.

On paper, the translation latency introduced due to the mapper may seem very significant when compared to CBT interconnect implementation, but the true impact of the heap access latency introduced by VDMMU can only be determined by running accelerator applications exhibiting various memory access patterns as described in section 7.2. Another drawback of the VDMMU is, it can only support MAU/AP configurations which are a power of 2.

The fine-grained DMA architectures have established, *Buddy System* based allocators exhibit the best scalability at a reasonable allocation latency, in terms of number of MAUs managed by the heap. But, buddy system suffers from heap fragmentation which can be a major concern when the heap size is small. Thus, VDMMU trades increased resource consumption in exchange for eliminating heap fragmentation, while excelling in providing very low effective (de)allocation latency of 1 cycle at a allocation request interval of 2. The absolute allocation and deallocation latency of VDMMU are 2 and 3 cycles, respectively.

Furthermore, VDMMU introduces a certain level of memory abstraction to designers when using HLS. The designer need not fret about managing memory resources during construction in C++, rather the designer may focus on efficient implementation of an accelerator/kernels functionality, followed by integration of the accelerator with the heap managed by VDMMU

using one of its Access Points. In conclusion DMA for BRAMs reduces the design complexity by removing memory management from the equation.

7.2 Future Work

The impact of VDMMU on throughput of multi-accelerator systems must be explored to determine its efficacy with relation to other multi-accelerator DMA implementations [10, 9]. Hence, as part of future work, a SpGEMM accelerator with multiple (4) PEs is interfaced to a collection of BRAMs (heap) using the VDMMU for further investigation of the impact of using VDMMU, on application throughput. Shown in figure 7.1 is a work in progress integration of 4 PEs, each performing *Row-Stationary* multiplication of sparse matrices. Each PE implements a streaming interface using FIFOs to perform read/write from the heap. Since the PEs are stream based, the VDMMU is implemented in SP mode of operation.

Furthermore, the paging mechanism used by the heap manager may be improved by replacing the first-fit algorithm with the buddy system. Other contemporary work have shown that buddy-system exhibits the highest scalability in terms of the number of MAUs managed by the heap. In order to realize paging with buddy-system, a scalable mechanism to dynamically compute the shift distance (count holes) must be developed. This would allow a smaller version of the reverse butterfly network to sequentially reduce heap fragmentation introduced by the buddy system. Thus striking a balance between resource consumption and performance.

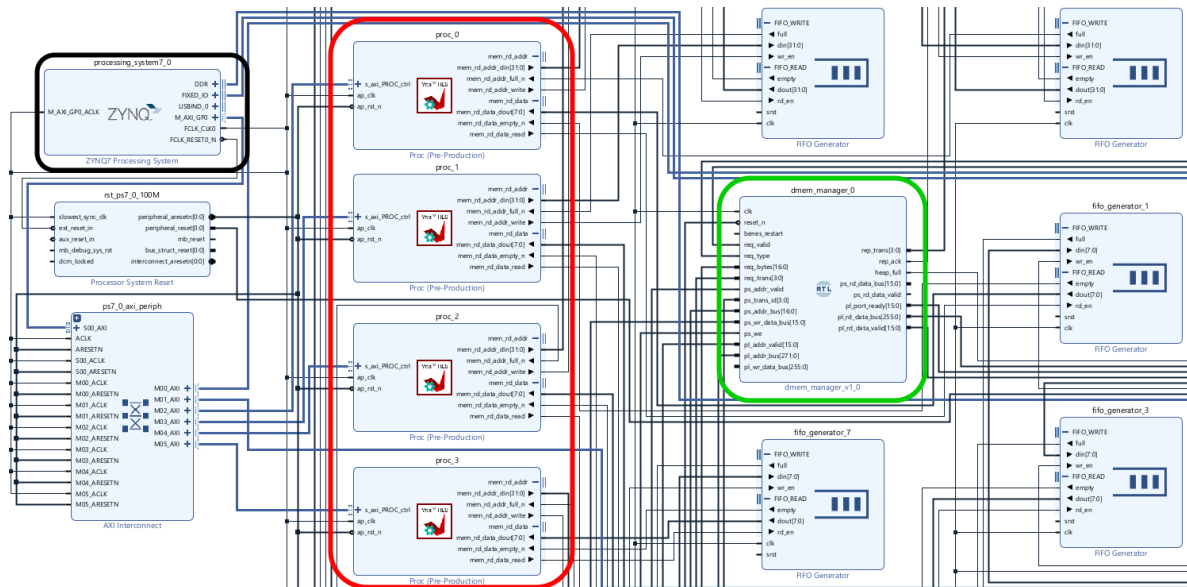


Figure 7.1: Four Sparse Matrix Multiplication accelerators (red box) attached to VDMMU (green box) in streaming mode, controlled by PS (black box).

Bibliography

- [1] Xilinx (AMD). "7 Series FPGAs Memory Resources User Guide (UG473)". In: online: AMD, 2019.
- [2] Xilinx (AMD). "Vitis High-Level Synthesis User Guide (UG1399)". In: online: AMD, 2024.
- [3] Xilinx (AMD). "Vivado Design Suite 7 Series FPGA and Zynq 7000 SoC Libraries Guide (UG953)". In: online: AMD, 2024.
- [4] Doug Abbott. "Chapter 17 - Linux and real-time". In: *Linux for Embedded and Real-Time Applications (Fourth Edition)*. Ed. by Doug Abbott. Fourth Edition. Newnes, 2018, pp. 257–270. ISBN: 978-0-12-811277-9. DOI: <https://doi.org/10.1016/B978-0-12-811277-9.00017-1>.
- [5] Andrew Boutros and Vaughn Betz. "FPGA Architecture: Principles and Progression". In: *IEEE Circuits and Systems Magazine* 21.2 (2021), pp. 4–29. DOI: 10.1109/MCAS.2021.3071607.
- [6] H. Cam, M. Abd-El-Barr, and S.M. Sait. "A high-performance hardware-efficient memory allocation technique and design". In: *Proceedings 1999 IEEE International Conference on Computer Design: VLSI in Computers and Processors (Cat. No.99CB37040)*. 1999, pp. 274–276. DOI: 10.1109/ICCD.1999.808436.
- [7] J.M. Chang and E.F. Gehringer. "A high performance memory allocator for object-oriented systems". In: *IEEE Transactions on Computers* 45.3 (1996), pp. 357–366. DOI: 10.1109/12.485574.
- [8] Yu-Ting Chen et al. "Accelerator-rich CMPs: From concept to real hardware". In: *2013 IEEE 31st International Conference on Computer Design (ICCD)*. 2013, pp. 169–176. DOI: 10.1109/ICCD.2013.6657039.
- [9] Ghada Dessouky et al. "Adaptive Dynamic On-chip Memory Management for FPGA-based reconfigurable architectures". In: *2014 24th International Conference on Field Programmable Logic and Applications (FPL)*. 2014, pp. 1–8. DOI: 10.1109/FPL.2014.6927471.
- [10] Dionysios Diamantopoulos et al. "Dynamic Memory Management in Vivado-HLS for Scalable Many-Accelerator Architectures". In: *Applied Reconfigurable Computing*. Ed. by Kentaro Sano et al. Cham: Springer International Publishing, 2015, pp. 117–128. ISBN: 978-3-319-16214-0.
- [11] Ru Ding et al. "A FPGA-based Accelerator of Convolutional Neural Network for Face Feature Extraction". In: *2019 IEEE International Conference on Electron Devices and Solid-State Circuits (EDSSC)*. 2019, pp. 1–3. DOI: 10.1109/EDSSC.2019.8754067.

-
- [12] Yedidya Hilewitz and Ruby B. Lee. "A New Basis for Shifters in General-Purpose Processors for Existing and Advanced Bit Manipulations". In: *IEEE Transactions on Computers* 58.8 (2009), pp. 1035–1048. DOI: 10.1109/TC.2008.219.
 - [13] Chin Hau Hoo and Akash Kumar. "An area-efficient partially reconfigurable crossbar switch with low reconfiguration delay". In: *22nd International Conference on Field Programmable Logic and Applications (FPL)*. 2012, pp. 400–406. DOI: 10.1109/FPL.2012.6339136.
 - [14] Longbo Huang and Jean Walrand. *A Benes Packet Network*. 2012. arXiv: 1208.0561 [math.OC].
 - [15] Xu Ji et al. "Understanding object-level memory access patterns across the spectrum". In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis. SC '17*. Denver, Colorado: Association for Computing Machinery, 2017. ISBN: 9781450351140. DOI: 10.1145/3126908.3126917.
 - [16] Yikun Jiang and Mei Yang. "Hardware implementation of parallel algorithm for setting up Benes networks". In: 2016.
 - [17] Yikun Jiang and Mei Yang. "Hardware implementation of parallel algorithm for setting up Benes networks". In: 2016.
 - [18] Y. Kai et al. "Design of partially-asynchronous parallel processing elements for setting up Benes networks in $O(\log^2 N)$ time". In: *2009 International Conference on Photonics in Switching*. 2009, pp. 1–2. DOI: 10.1109/PS.2009.5307812.
 - [19] Labson Koloko, Takahiro Matsumoto, and Hitoshi Obara. "Design and implementation of fast and hardware-efficient parallel processing elements to set full and partial permutations in Beneš networks". In: *The Journal of Engineering* 2021.6 (2021), pp. 312–320. DOI: <https://doi.org/10.1049/tje2.12037>. eprint: <https://ietresearch.onlinelibrary.wiley.com/doi/pdf/10.1049/tje2.12037>.
 - [20] Marcin Kowalczyk et al. "Real-Time FPGA Implementation of Parallel Connected Component Labelling for a 4K Video Stream". In: *Journal of Signal Processing Systems* 93.5 (May 2021), pp. 481–498. ISSN: 1939-8115. DOI: 10.1007/s11265-021-01636-4.
 - [21] Onur Küçüktunç et al. "Fast recommendation on bibliographic networks with sparse-matrix ordering and partitioning". In: *Social Network Analysis and Mining* 3.4 (Dec. 2013), pp. 1097–1111. ISSN: 1869-5469. DOI: 10.1007/s13278-013-0106-z.
 - [22] T.T. Lee and S.Y. Liew. "Parallel routing algorithms in Benes-Clos networks". In: *IEEE Transactions on Communications* 50.11 (2002), pp. 1841–1847. DOI: 10.1109/TCOMM.2002.805258.
 - [23] Tingyuan Liang et al. "Hi-DMM: High-Performance Dynamic Memory Management in High-Level Synthesis". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 37.11 (2018), pp. 2555–2566. DOI: 10.1109/TCAD.2018.2857040.
 - [24] Shaoteng Liu, Axel Jantsch, and Zhonghai Lu. "Analysis and Evaluation of Circuit Switched NoC and Packet Switched NoC". In: *2013 Euromicro Conference on Digital System Design*. 2013, pp. 21–28. DOI: 10.1109/DSD.2013.13.
 - [25] Nassimi and Sahni. "A Self-Routing Benes Network and Parallel Permutation Algorithms". In: *IEEE Transactions on Computers* C-30.5 (1981), pp. 332–340. DOI: 10.1109/TC.1981.1675791.

-
- [26] Tuan D. A. Nguyen and Akash Kumar. "XNoC: A non-intrusive TDM circuit-switched Network-on-Chip". In: *2016 26th International Conference on Field Programmable Logic and Applications (FPL)*. 2016, pp. 1–11. DOI: 10.1109/FPL.2016.7577378.
- [27] D. C. Opferman and N. T. Tsao-wu. "On a class of rearrangeable switching networks part I: Control algorithm". In: *The Bell System Technical Journal* 50.5 (1971), pp. 1579–1600. DOI: 10.1002/j.1538-7305.1971.tb02569.x.
- [28] Cenk Özer. "A Dynamic Memory Manager for FPGA Applications". THE GRADUATE SCHOOL OF NATURAL and APPLIED SCIENCES OF MIDDLE EAST TECHNICAL UNIVERSITY, 2014.
- [29] Ashley Walker Robert Fisher Simon Perkins and Erik Wolfart. *Image Analysis - Connected Components Labeling*. [Accessed 20-07-2024]. 2003.
- [30] Mohamad Mehdi Sadeghi, Somayeh Timarchi, and Mahmood Fazlali. "High-Performance Memory Allocation on FPGA With Reduced Internal Fragmentation". In: *IEEE Access* 11 (2023), pp. 66672–66681. DOI: 10.1109/ACCESS.2023.3290100.
- [31] Fazle Sadi et al. "PageRank Acceleration for Large Graphs with Scalable Hardware and Two-Step SpMV". In: *2018 IEEE High Performance extreme Computing Conference (HPEC)*. 2018, pp. 1–7. DOI: 10.1109/HPEC.2018.8547561.
- [32] Tsutomu Sasao and Alan Mishchenko. "LUTMIN : FPGA Logic Synthesis with MUX-Based and Cascade Realizations". In: 2009.
- [33] Dimitrios Serpanos and Tilman Wolf. "Chapter 4 - Interconnects and switching fabrics". In: *Architecture of Network Systems*. Ed. by Dimitrios Serpanos and Tilman Wolf. The Morgan Kaufmann Series in Computer Architecture and Design. Boston: Morgan Kaufmann, 2011, pp. 35–61. DOI: <https://doi.org/10.1016/B978-0-12-374494-4.00004-9>.
- [34] Nitish Srivastava et al. "MatRaptor: A Sparse-Sparse Matrix Multiplication Accelerator Based on Row-Wise Product". In: *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 2020, pp. 766–780. DOI: 10.1109/MICRO50266.2020.00068.
- [35] Tsung-Han Tsai, Yuan-Chen Ho, and Chi-En Tsai. "Implementation of Real-Time Connected Component Labeling Using FPGA". In: *2018 IEEE International Conference on Consumer Electronics-Taiwan (ICCE-TW)*. 2018, pp. 1–2. DOI: 10.1109/ICCE-China.2018.8448784.
- [36] George Varghese. "Chapter 13 - Switching". In: *Network Algorithmics*. Ed. by George Varghese. The Morgan Kaufmann Series in Networking. San Francisco: Morgan Kaufmann, 2005, pp. 302–338. DOI: <https://doi.org/10.1016/B978-012088477-3/50016-3>.
- [37] Ganesh Venkatesh et al. "Conservation cores: reducing the energy of mature computations". In: *SIGARCH Comput. Archit. News* 38.1 (Mar. 2010), pp. 205–218. ISSN: 0163-5964. DOI: 10.1145/1735970.1736044.
- [38] E. Von Puttkamer. "A Simple Hardware Buddy System Memory Allocator". In: *IEEE Transactions on Computers* C-24.10 (1975), pp. 953–957. DOI: 10.1109/T-C.1975.224100.
- [39] Paul R. Wilson et al. "Dynamic storage allocation: A survey and critical review". In: *Memory Management*. Ed. by Henry G. Baler. Berlin, Heidelberg: Springer Berlin Heidelberg, 1995, pp. 1–116. ISBN: 978-3-540-45511-0.
- [40] Zeping Xue and David B. Thomas. "SysAlloc: A hardware manager for dynamic memory allocation in heterogeneous systems". In: *2015 25th International Conference on Field Programmable Logic and Applications (FPL)*. 2015, pp. 1–7. DOI: 10.1109/FPL.2015.7293959.