A Novel Access Pattern-based Multi-core Memory Architecture

Author:
Tassadaq Hussain

Advisors:
Dr. Oscar Palomar, **Dr. Adrian Cristal**, **Dr. Osman S. Ünsal**,
Prof. Eduard Ayguadé and **Prof. Mateo Valero**

Submitted to the Departament d'Arquitectura de Computadors in Partial Fulfillment of the Requirements for

Doctor of Philosophy (PhD)

Barcelona - December 2014

بسم الله الرحمن الرحيم

To my father Aman Ullah Cheema, my sweet mother Iffat Tanveer, my lovely wife Amna Hussain and my charming little kid Umar Hussain

Acknowledgements

* All praises are due to Allah and may He send His salutations on our Master and Messenger of Allah Muhammad (P.B.U.H) and his brethren among the Prophets and Messengers and on their families and companions and whoever followed them in righteousness until the Day of Judgment.

* This work was not possible to complete without the help of my advisors. My cordial gratitude goes to my supervisors: Prof. Mateo Valero, Prof. Eduard Ayguadé, Dr. Adrian Cristal, Dr. Osman S. Ünsal and Dr. Oscar Palomar. I can never forget their outstanding manners, genius way of thinking and benevolence towards me. Because of their fruitful and everlasting efforts I feel very confident in myself in the field of science and technology. I often drew the inspiration from their incessant novel ideas. I consider myself fortunate to have the honor of being a Ph.D. student under their supervision. Dr. Oscar Palomar is the one who was absorbing and tolerating my strange ideas and helping me extraordinarily to bring them into good and meaningful shape before that these are presented to our seniors and as well at various research platforms. He has been conducting regular meetings throughout the years. I always found him ready to spare time for the research discussions.

* I would like to express my gratitude to Dr Miquel Pericas and Prof. Nacho Navarro. They helped me a lot to push this work ahead in the right direction. Later, my research group was changed but my learning with them helped me a lot to effectively pursue later studies.

* The acknowledgements will remain incomplete if I do not quote the organizations and the projects who financially supported this thesis. These include Barcelona Supercomputing Center (BSC), Technical University of Catalonia (UPC), The Ministry of Science and Innovation of Spain (Project: Computaciòn de Altas Prestaciones V, TIN2007-60625), The European Union Framework Program-7 (Project: Scalable Computer Architecture, IST-2006-27648), the HiPEAC-2 Network of Excellence (FP7/ICT 217068), the European Commission in the context of the SARC Project #27648 (FP6), the European Research Council under the European Unions 7th FP (FP/2007-2013) / ERC GA n. 321253, the Spanish Government (TIN2012-34557), and the Unal Center of Education Research and Development (UCERD). I am extremely thankful to all these organizations for their full support during the work.

Abstract

Increasingly High-Performance Computing (HPC) applications run on heterogeneous multi-core platforms. The basic reason of the growing popularity of these architectures is their low power consumption, and high throughput oriented nature. However, this throughput imposes a requirement on the data to be supplied in a high throughput manner for the multi-core system. This results in the necessity of an efficient management of on-chip and off-chip memory data transfers, which is a significant challenge. Complex regular and irregular memory data transfer patterns are becoming widely dominant for a range of application domains including the scientific, image and signal processing. Data accesses can be arranged in independent patterns that an efficient memory management can exploit. The software based approaches using general purpose caches and on-chip memories are beneficial to some extent. However, the task of efficient data management for the throughput oriented devices could be improved by providing hardware mechanisms that exploit the knowledge of access patterns in memory management and scheduling of accesses for a heterogeneous multi-core architecture.

The focus of this thesis is to present architectural explorations for a novel access pattern-based multi-core memory architecture. In general, the thesis covers four main aspects of memory system in this research. These aspects can be categorized as: i) Uni-core Memory System for Regular Data Pattern. ii) Multi-core Memory System for Regular Data Pattern. iii) Uni-core Memory System for Irregular Data Pattern. and iv) Multi-core Memory System for Irregular Data Pattern.

Contents

List of Figures

List of Tables

Part I

Introduction and Research Proposal

1

Introduction

A typical multi-core system uses two types of random-access memory (RAM): *on-chip memory* usually consisting of static SRAM, and *off-chip memory* usually consisting of dynamic DRAM. The on-chip and off-chip memories are also called *Local Memory* and *Main Memory* respectively (shown in Figure 1.1). The *on-chip memory* has low read/write access latencies compared to *off-chip memory*. The *Off-chip memory* is less expensive by a factor of 20 or more, denser and slower by a factor of 10 to 100 than *on-chip memory*. To combine their advantages, often a low-cost *off-chip memory* is used in the system that handles large data with high latency, and then a small low latency but expensive *on-chip memory* is added to manage the frequently used data in run-time.

The deviation between capacity and operating frequency is crucial in memory technology. In last few decades, the capacity of DRAM chips enlarged a thousand fold, quadrupling every three years while the memory cycle time improved by a factor of

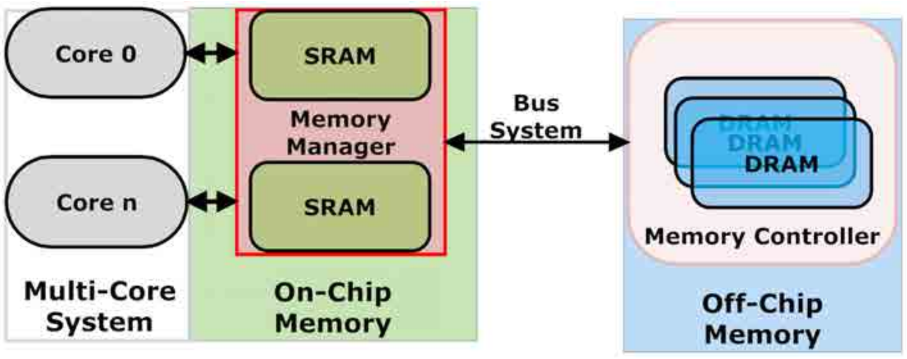


Figure 1.1: Conventional Multi-core Memory System

two only. In current multi-core architectures, the gap between processor cycle time and memory cycle time is substantially getting larger. The conventional DRAM architectures are using wider, and wider paths [1] into memory and greater interleaving of memory banks. A number of advanced DRAM [2] designs appear on the market which transfer a large number of bits per memory cycle within the chip, and pipeline the transfer of those bits at high frequency. These advanced designs improve the bandwidth, but latency issues remain there.

Efficient *Memory Management* is important to improve performance for many applications running on multi-core systems. The multi-core systems normally support efficient utilization of data in SRAMs by providing a fixed architecture of caches or scratchpad memories. The cache stores copies of the data from frequently used *Main Memory* locations. Transparent to the programmer, the cache memories [3] are very effective when the working set fits in the cache hierarchy, and there is locality. In many High Performance Computing (HPC) applications the data sets can be large and have unpredictable, irregular patterns, thus reducing the spatial locality. In cache memory systems, the mapping of program elements is done transparently, whereas in scratchpad memory systems this is done either by the user or automatically by the compiler using lookup table algorithms [4]. Scratchpads are, usually, not feasible for applications having irregular or dynamic data structures [5]. The generated scratchpad code is not portable across different sizes of scratchpads.

In the multi-core architecture, noncontiguous memory accesses take extra time while accessing and arranging data in *Local Memory* and generate delays while accessing data to/from *Main Memory*. Prefetching mechanism improves *Local Memory* usage efficiency by predicting future *Main Memory* data accesses and prefetching them before the data access request. Hardware prefetching [6] is effective in hiding the *Main Memory* delay when the accesses are known in advance, but it cannot be tailored for data-dependent accesses, e.g. lists, trees, etc. Adaptive software prefetching [7] can be utilized to change prefetch distances during runtime, but it is difficult to insert prefetch information for irregular access patterns at runtime. A number of intelligent and high-performance memory systems [8; 9] exist to manage the processor and *Main Memory* speed gap. Unfortunately, these memory controllers rely on a master processor or microcomputer and are typically limited to applications with regular memory access patterns thereby, prohibiting the acceleration of applications with irregular patterns.

Due to limited support of irregular access patterns in existing memory controllers, the multi-core systems are not effectively leveraging code with pointer-based data structures [10]. These memory systems are designed on the basis of few heuristics that are generic enough to provide different degree of performance enhancement for various applications. However, the performance for certain applications on the device can still be improved by providing more specialized memory layouts for those applications.

Over the past decade, system architectures changed towards reconfigurability, parallelism and heterogeneity that results in high performance, low power and easy design. Field-Programmable Gate Array (FPGA) provides high performance by reconfiguring parameterized general purpose soft cores and application-specific hardware accelerators. Due to the significant performance improvements, high density, low power and reconfigurability feature of FPGAs [11; 12] the combination of multi-cores and heterogeneity, referred to as multi-core heterogeneous systems for simplicity, is becoming more popular in areas ranging from low-power embedded systems to high-performance computing (HPC) systems. As the number of cores increases the demand of memory bandwidth increases as well. Efficient memory hierarchy is required that accelerates both regular and irregular access patterns manages memory accesses by applying intelligent scheduling policies.

As the number of cores in a chip increases the reuse of the communication infrastructure becomes difficult. Buses and point to point connections that have been the principal means to connect multiple cores on a chip today do not result in scalable platform architecture for many core architecture. Buses can cost-efficiently connect a few tens of processing cores. Point to point communication connections are practical for cores having regular streaming data transfers. As the applications having complex and irregular access patterns increase on multi-core system, latency of the long wire switching, effects the performance of the system. Therefore, the pattern (packet) based on-chip communication infrastructure can play an increasingly dominant role in multi-core design.

This thesis presents an overall view of the thesis work which centers upon the architectural explorations for heterogeneous multi-core systems. The chapter starts by giving a general introduction on the heterogeneous multi-core system architectures. This is followed by description of a conventional memory system. Before summarizing the chapter, we will briefly look at the contributions made during this thesis work as well the organization of this thesis document.

1.1 Target Heterogeneous Multi-core Systems

In the light of Moore's law, the number of transistors on silicon chips doubles approximately every 18 months. The processor clock speed increased 30% per year and hit the power wall when it reached 5 GHz due to power density [13]. The power wall then directed computer technology to carry Moore's law by adding multiple processing cores in a chip and keeping the clock frequency constant. In a comparison to the traditional parallel computing on general purpose cores, heterogeneous systems with components such as the application specific accelerators have exhibited significant performance advantages in application domains such as multi-media, graphics [14], digital signal processing [15] and some scientific applications. These hardware accelerators have low footprint and low power consumption and provide high performance computation. The application using data in the form of complex and irregular data structures could also utilize the potential of accelerators computing. This may require a general purpose Reduced Instruction Set Computing (RISC) core that reorders and manages data transfer tasks.

To manage power, memory wall effects and complexity of the system, the HPC industry supports multi- processor and accelerator cores for data intensive scientific applications. The *Application Specific Hardware Accelerators* (*ASHA*) and *Vector Processor* (*VP*) architectures lead to throughput oriented computing on devices with parallel streaming architectures. These throughputs oriented architectures execute parallel workloads while attempting to maximize total throughput. The *ASHA* and *VP* architectures consist of numerous types of components including the memory structures, compute units, control units, I/O channels and I/O controls, etc. The hardware accelerator cores, normally, offer a large number of compute resources but restrict the applications to arrange parallel and maximally independent data sets to feed the compute resources as streams. That is why the HPC accelerators use a *Scalar Soft Processor core* (*SSP*) *Master Core* to feed data. The performance for these applications on an arbitrary platform depends on how well the data is managed into streams before is forwarded to the computing cores. An example view of the data flow for HPC application [16] through

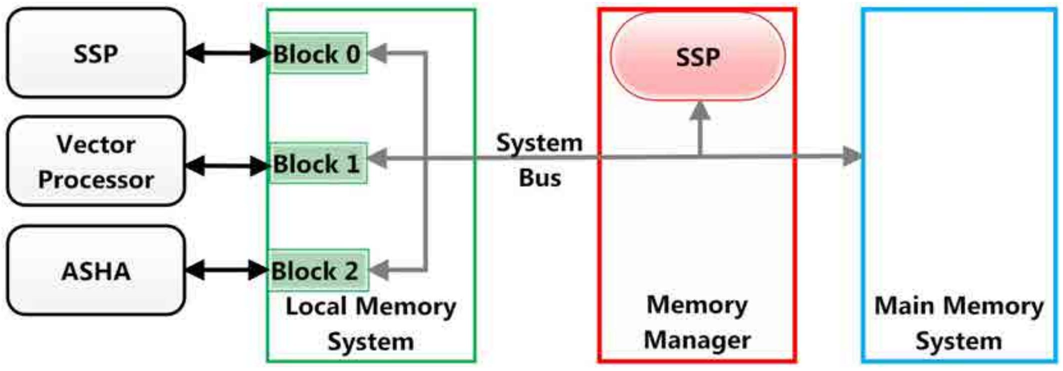


Figure 1.2: An example view of data flow of Heterogeneous System

a heterogeneous architecture is shown in the Figure 1.2. The figure shows that *SSP*, *ASHA* and *VP* are processing blocks of data ($Block_0$), ($Block_1$) and ($Block_2$) respectively. The heterogeneous multi-core system does not get optimum performance if accelerators are not tightly scheduled. The *SSP* schedules the memory requests and manages data transfers between *Accelerator Cores*, *Local Memory* and *Main Memory*. The *SSP* master core is capable of initiating data transfer on the bus and is also responsible for memory management and scheduling of heterogeneous multi-core system. It decides when a processing core start execution and provides the memory and I/O link. All cores can communicate with each other via *SSP*. Each *Accelerator Core* takes data from their *Local Memory* and performs computation in parallel. The section is further divided into a discussion on *FPGA* and a description of three types of cores that we consider in our heterogeneous architecture *Scalar Soft Processor core* (*SSP*), *Application Specific Hardware Accelerators* (*ASHA*) and *Vector Processor* (*VP*).

1.1.1 Field Programmable Gate Arrays (FPGA)

With the rapidly rising cost, time-to-market of designing state-of-the-art ASICs [17] and fast prototyping [18], an increasing number of HPC systems are being built using FPGA platforms [19]. FPGA technology now also mainly used in research and development sectors because it provides an environment where designer can build new architecture using a variety of languages and design flows. It allows designers to integrate multiple heterogeneous cores into one architecture and test it with real time debugging facility. With careful planning and partnering with an experienced Application Specific Integrated Circuit (ASIC) vendor, the FPGA based architecture can be fabricated in ASIC.

There are two possibilities when implementing general purpose codes in FPGA systems: hard-core processors, that are included in the chip besides the FPGA and offer excellent packaging and communication advantages, and soft-core processors, that are implemented configuring FPGA resources and offer the advantage of flexibility and lower part costs. While a soft core cannot easily match the performance/area/power consumption of a hard processor, processors have several compelling advantages. Many FPGA vendors are now offering such scalar soft processor/accelerator cores [20; 21; 22; 23] that designers can implement using a standard FPGA. The number of cores can be varied depending upon the required application and FPGA resources and FPGA Computer Aided Design (CAD) tools automatically place them within the design to ease routing. To address the computational requirements introduced by many demanding HPC applications, several research efforts have aimed at implementing HPC applications for different hardware architectures (Central Processing Unit (CPU), Graphical Processing Unit (GPU) and FPGA). Each architecture has its design methodology and specific optimizations for performance. A way to exploit parallelism is to build a heterogeneous multi-core system on FPGAs using parameterized general purpose soft-cores and application specific hardware accelerators.

1.1.2 Scalar Soft Processor core (SSP)

As the target design moves towards HPC, the design architecture requires high speed, programmable and low power architecture. Such architecture should have parameterizable processing, memory and communication units that can reconfigure a system according to the applications needs. The control unit is also required that schedules and manages the different components and their data movement. A soft parameterizable core is used in the design to support wide range of general purpose applications. A *SSP* is a Hardware Description Language (HDL) model of a general purpose processor that can be customized for a given application and synthesized for an ASIC or FPGA target. A parameterized soft processor is a processor designed at the Register Transfer

Level (RTL) allowing certain aspects of the architecture to be varied. The RTL design is expressed using a HDL such as Verilog or Very High Speed Integrated Circuit HDL (VHSICHDL or VHDL) which contains parameters that can be tuned to alter the processor architecture in a manner intended by the original designer. The functionality of the processor is guaranteed by the original designer for many combinations of parameter values.

1.1.3 Application Specific Hardware Accelerator (ASHA)

FPGAs have traditionally been programmed with languages and methodologies that stem from the Electronic Design Automation (EDA) community. While in HPC, high level languages such as C and Fortran are used. This design gap between EDA methodologies and HPC programmers has slowed the adoption of FPGAs by the HPC community. Translation of HPC algorithms into hardware is complex and time consuming. Large applications result in complex systems with a high probability of containing design errors, and it is challenging to reuse the hardware resources. To overcome these difficulties, FPGA designers use C-to-hardware High Level Synthesis (HLS) that performs design modeling and validation at a higher level of abstraction. However, this approach has some limitations. Every application kernel requires its own accelerator. A scalar processor core is needed to manage *Local Memory* data and *Main Memory* transfers for the accelerators. Every change in the code of the application introduces a new place-and-route iteration to generate the hardware accelerator. Hence, even simple changes can lead to dramatic shifts in area usage and clock frequency. All of these factors make design closure with custom data-path accelerators very difficult.

The HLS design flow consists of a sequence of steps (see Figure 1.3), with each step transforming the abstraction level of the algorithm into a lower level description. The HLS *Parallelism Extraction* step extracts the Control and Data Flow Graph (CDFG) from the application to be synthesized. The CDFGs describe the computational nodes and edges between the nodes. HLS provides different methods to explore the CDFG of the input algorithm and generates the data-path structure. The generated data-path structure may contains the user-defined number of computing resources for each computing node type and the number of storage resources (registers). The *Allocation and Scheduling* step maps the CDFG algorithm onto the computing data-path and

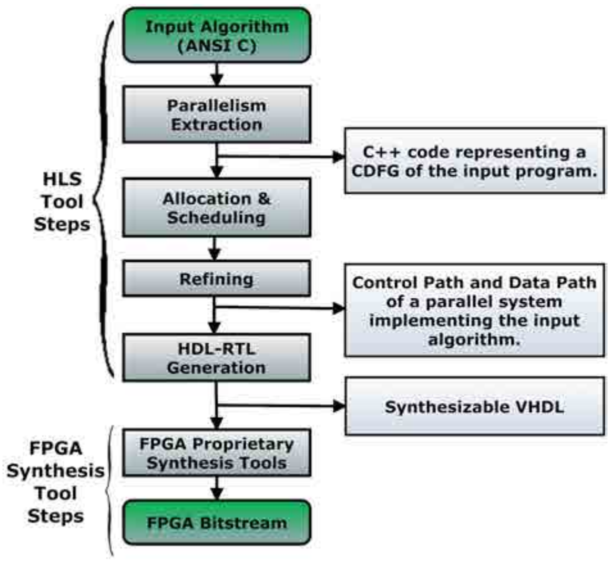

Input Algorithm (ANSI C)
Parallelism Extraction
C++ code representing a CDFG of the input program.
Allocation & Scheduling
HLS Tool Steps
Refining
Control Path and Data Path of a parallel system implementing the input algorithm.
HDL-RTL Generation
Synthesizable VHDL
FPGA Synthesis Tool Steps
FPGA Proprietary Synthesis Tools
FPGA Bitstream

Figure 1.3: Conventional High Level Synthesis Tool Flow

produces a Finite State Control Machine. The *Refining* step uses the appropriate board support package and performs synthesis for the *communication network*. Finally, the *HDL-RTL Generation* step produces the VHDL files to be supplied to the proprietary synthesis tools for the targeted FPGA.

Over the past few years, HLS tools have been developed that add the necessary technologies to become truly production-worthy. Initially limited to data path designs, HLS tools have now started to address complete systems, including control-logic and complex on-chip, off-chip interconnection. Xilinx Vivado HLS [24] (built on AutoESL tool technology [25]) accelerates design implementation. It takes C, C++, or SystemC as its input and produces device-specific RTL after exploring a multitude of micro-architectures based on design requirements. Impulse Accelerated Technologies develops the ImpulseC programming language [26]. The ImpulseC tools comprise a software-to-hardware compiler that translates individual Impulse C processes to hardware and generates the necessary process-to-process interface logic. Handel-C, developed by Celoxica [27], is based on the syntax of conventional C language. Programs written in Handel-C are sequential. To exploit benefits of parallelism from the target hardware, Handel-C provides parallel constructs such as pipelined communication and parallel sections. Catapult C, designed by Mentor Graphics [28], is a subset of

C++. The code that is compiled through Catapult C may be general purpose and result in much different hardware implementations with different timing and resource constraints. The Catapult C environment takes constraints and platform details in order to generate a set of optimal implementations. Ylichron (now PLDA Italia) developed a source-to-source C to VHDL compiler toolchain targeting system designers called HCE (Hardware Compiling Environment). The HCE toolchain [29] takes ANSI-C language as input, which describes the hardware architecture with some limitations and extensions. ROCCC 2.0 [23] is a free and open source tool that focuses on FPGA-based code acceleration from a subset of the C language. ROCCC tries to exploit parallelism within the constraints of the target device, optimize clock cycle time by pipelining, and minimize area. ROCCC is one of the few HLS tools that does memory access optimization.

1.1.4 Vector Processor (VP)

FPGA soft processor cores typically decrease system performance compared with hard-core processors. A way to provide substantial performance improvement, FPGA based soft vector processors have been proposed [30; 31]. A soft vector processor comprises a parameterized number of vector lanes, a vector memory bank and a crossbar network that shuffles vector operations. Soft vector architecture is very efficient for HPC applications with Data Level Parallelism (DLP), not only it can process multiple data elements based on a single vector instruction but also it can reconfigure itself depending upon the required performance.

A vector processor is also known as a “Single Instruction, Multiple Data” (SIMD) CPU [32], that can operate on an array of data in a pipelined fashion, one element at a time using a single instruction. For higher performance multiple Vector Lanes (VL) can be used to operate in lock-step on several elements of the vector in parallel. The number of vector lanes determines the number of Arithmetic Logic Units (ALUs) and elements that can be processed in parallel. The maximum vector length (MVL) determines the capacity of the vector register files (RF). Increasing the MVL allows a single vector instruction to encapsulate more parallel operations, but also increases the vector register file size. The vector processor uses a scalar core for all control flow, branches, stack, heap, and input/output ports.

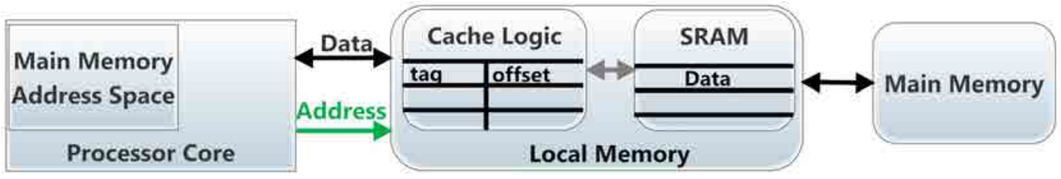


Figure 1.4: A Conventional Memory System Architecture

1.2 A Conventional Memory System

Figure 1.4 shows a conventional memory system having *Local Memory* and *Main Memory*. For each data access request, the *Memory System* accesses *Local Memory* in order to check if it holds the requested data. If it does, a *Local Memory* hit occurs, and the *Memory System* bypasses the *Main Memory* and provides the data to the processing core. Otherwise, the *Main Memory* transfers the aligned data block; the processing core receives the data and the *Local Memory* stores the whole block. The *Memory System* checks both address and data, and updates copy of data in *Local Memory*. When a read request is generated for an address location that is in *Local Memory*, the *Memory System* reuses its copy and avoid the *Main Memory* access. We first describe the *Local Memory System* and then the *Main Memory System*.

1.2.1 Local Memory System

The advantage of an SRAM based local memory (shown in Figure 1.5) is its speed but they suffer from a very large cell area that limits their density and make them ex-

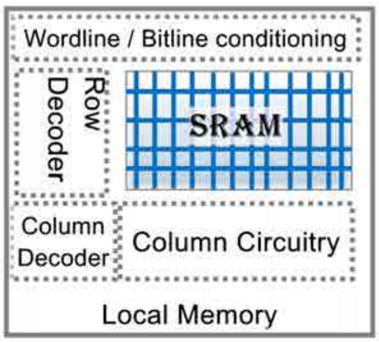


Figure 1.5: SRAM: Local Memory

pensive, therefore, small SRAM memory is used to balance the performance and cost. The performance of the embedded DRAM (eDRAM) technologies are improving have higher reliability and are expected to dissipate less power but have higher data access latency than SRAM based *Local Memory System*. SRAM-based *Local Memory System* are popular due to its read/write access time, but the power dissipation and reliability are primary issues. Therefore, the *Local Memory System* integrates a small SRAM memory close to the processing core and uses a number of techniques to improve on-chip data locality. Two types of *Local Memory System*s are *Cache* and *Scratchpad*.

1.2.1.1 Cache

Caches are present in most memory systems. The *Cache* dynamically stores a subset of the frequently used data. Thus, the timing of a load or store operation depends on the relationship between its effective address and the effective addresses of earlier operations. The processor shares an address bus and a data bus with the memory. When the processing core makes a memory access (read data, or write data), the *Memory System* first checks if the cache contains the data. The *Cache* monitors the access requests in order to check if it holds the requested data. If it does, a cache hit occurs and the cache bypasses the *Main Memory* and provides the data to the processing core. Otherwise, a cache miss occurs, and the *Main Memory* provides the data and its surrounding elements, the processing core receives the data and the cache stores the whole data elements as a cache line. The time cost of a cache miss is high due to the latency of the *Main Memory* accesses. A cache can be direct-mapped, fully-associative or n-way set associative.

A conventional cache uses byte addressable memory, i.e. each address refers to a byte. For example 2^b is the size of cache in bytes and 2^B is byte size of the *Main Memory* and `Addr` is the address of the *Main Memory*. A single data transfers to/from the memory hierarchy is called a cache line (`CL`) or block, `NCL` is the total number of `CLs` and `CLS` is the cache line size .

A Direct-mapped cache is one of the simplest placement policy for caches. Each memory location has only one possible location in the cache. The cache address $\mathtt{cache}_{address}$ is gathered by the LSB.

$$\mathtt{cache}_{\mathtt{address}} = (\mathtt{cache}_{\mathtt{address}}[\mathtt{b} - 1 : 0])$$

$$= \frac{(addr[\mathtt{B}-1:0])}{2^{\mathtt{b}}} = (addr[\mathtt{b}-1:0]) \tag{1.1}$$

This scheme is easy since there is just one location in the cache to lookup. The cache only checks if the tag of the cache line has `data`. The drawback of direct mapped caches is that they behave inefficiently in case of data request collisions. However, their access and replacement policy is considerably simpler than the fully associative caches.

In a fully associative cache, a block can be placed in any cache line. In this case, the *Main Memory* address `addr` is split into two parts: *tag* and *offset*. In the case of a cache miss, a whole `CL` is transferred from the *Main Memory* to the cache. The addresses of consecutive `data` share the same *tag*. While processing a request of a particular data, the most significant bits of its address are compared with the *tag* of each of the `NCL` cache lines. The comparisons have to take place at the same time, forcing the hardware to comprise `NCL` comparators, i.e. one per cache line.

The set associate cache tends to conciliate the advantages of direct-mapped and fully associative caches. It provides a set of possible locations for a cache line. Contrarily to fully associative caches, the set is limited to a small integer k, usually, chosen as a power of 2. The cache is then referred as k-way set associative. It can be seen as if the cache is divided into k smaller caches, each one of them being direct-mapped. One can pinpoint that when k = 1 the cache is direct mapped and when k = `NCL` the cache is fully associative. The set is given by `addr` [$\log_2$ (k) + `CLS` - 1 : `CLS`], so that tag can be compared with k locations at the same time. The hardware is simpler than in fully associative cache, and the risk of cache line conflicts is smaller than in direct-mapped caches, therefore, the set associative caches are often used as a compromise for processing core caches.

1.2.1.2 Scratchpad

The *Scratchpad* is a fast directly addressed software managed SRAM memory. The *Scratchpad* has better real-time guarantees than caches and by its significantly lower overheads it is better in access time, energy consumption and area. Recent advances [4] have made much progress in compiling static structures into scratchpad memory that

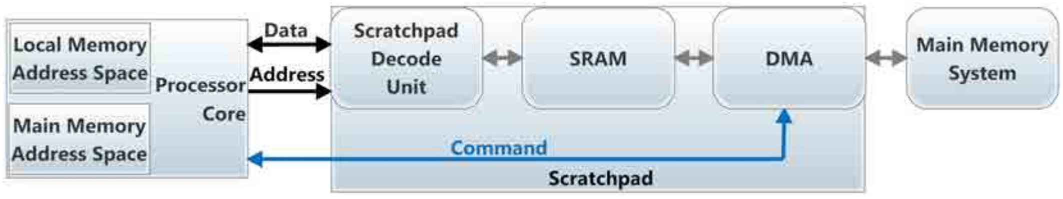


Figure 1.6: Scratchpad Memory Hierarchy

enable several performance enhancements. Instead of using traditional load/store instructions the scratchpad uses direct memory-memory operations using DMA. The *Scratchpad* memory access uses source and destination address registers, each of which holds a starting address of the memory.

The *Scratchpad* memory (shown in Figure 1.6) is designed to keep in view that the memory data is mapped to the *Local Memory*. The *Scratchpad* memory occupies one distinct part of the memory address space with the rest of the space occupied by *Main Memory*. Thus, the running application checks for the availability of the data/instruction in the *Scratchpad*. This eliminates the comparator and the signal miss/hit acknowledging circuitry and due to this, and energy and area consumption are reduced. On the contrary, the *Cache* uses an uniform address space. Whereas the *Scratchpad* uses the non-uniform address space. The accessed block of data can have data of non-contiguous memory locations and is placed in the *Scratchpad* memory known location and is managed by software. Therefore allocating irregular/complex data structures (e.g. pointers, linked-lists, etc.) to *Scratchpad* has proven far more difficult. Scratchpad lacks compile-time methods for irregular memory allocation [5], and a number of techniques are proposed to decide what to place in *Scratchpad* only at runtime; however mostly they have not been successful [33]. Accesses to conventional scratchpad are always as fast as data cache hits. The *Scratchpad*s offer a time-predictable replacement for a cache. Accesses outside the *Scratchpad*s are direct accesses to main memory, incur a large time penalty that is similar to a cache miss.

1.2.2 Main Memory System

A conventional *Main Memory System* uses an Synchronous DRAM (SDRAM) controller that receives the physical address to be read or written as well as maintaining the timing of other signals corresponding to the different phases of the SDRAM access protocol. The SDRAM controller implementations may have different levels of complexity; some may gather and reorder requests to increase access locality, and others use simple First In First Out (FIFO) processing of memory requests.

At the highest level, the SDRAM arrays are divided into ranks, typically with one or two ranks per SDRAM module. Within each rank, the memory is subdivided into banks and each bank consists of a 2 Dimensional array as shown in Figure 1.7. On an SDRAM read, bits from the physical address select the rank, bank and row; a set of sense amplifiers reads the contents of that row and then the results are latched in a row buffer. Any subsequent access to the same row can bypass the array-read and access the data directly from the row buffer. A few remaining bits from the physical address select the column to be read. The data are sent back to the SDRAM controller, which are later transferred back to the processor.

A conventional *Main Memory System* uses a Miss Status Handling Register (MSHR) to store bookkeeping information regarding the *Local Memory System* missed requests. The request is buffered in the MSHR and can be rearranged by a scheduler to use open banks and rows of SDRAM. The scheduler prioritizes the memory requests by keeping the fairness and maximize the offered net bandwidth. The *Main Memory* can have many SDRAM modules that can work independently of data processing.

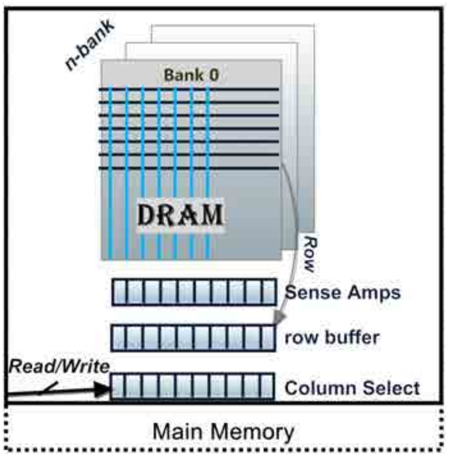


Figure 1.7: DRAM: Main Memory

]

1.3 Problem and Motivation

In a heterogeneous multi-core environment (see Figure 1.8), a master core (microprocessor) is used to manage the memory addresses and to schedule the data transfer requests of a multi-core system. Non optimized memory management and data transfer scheduling of application kernels could lead to significant performance degradation. In such a scenario, efficient management of memory accesses at compile-time as well as at run-time, across the set of multi-cores is critical to achieve high performance. This section presents three aspects of memory management in multi-cores: *Multi-core System Delays*, *Access Patterns* and *Scheduling*.

1.3.1 Multi-core System Delays

The factors that cause performance degradation are categorized into four types: *Memory Management Delay*, *Bus Delay*, *Scheduling Delay* and *Memory Delay*.

1.3.1.1 Memory Management Delay

The *Memory Management* incurs delays while handling data transfers – especially irregular and complex patterns – and performing address management to access data locations in memory. A conventional memory system uses a Load Store Unit (LSU) and Direct Memory Access (DMA) to handle data between multi-cores, *local* and *main memories*.

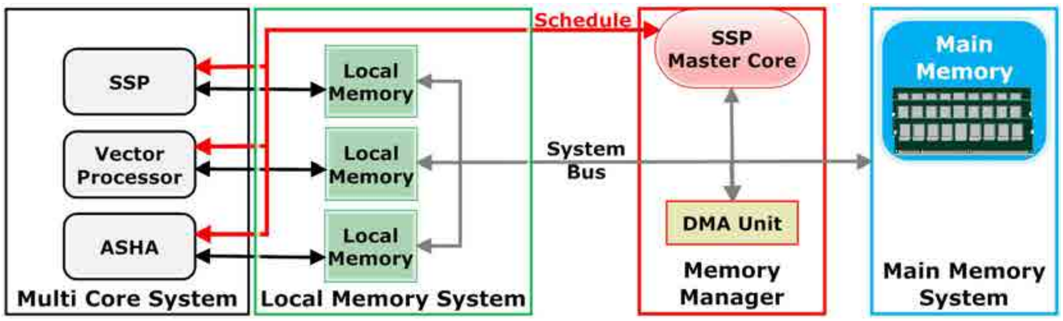


Figure 1.8: Generic Multi-Core Architecture

```
for ( start_value; inc=end_value; inc++)
//Addresses Management by Microprocessor
local_memory[inc]=main_memory[inc+offset];
main_memory[inc+offset]=local_memory[inc];
```

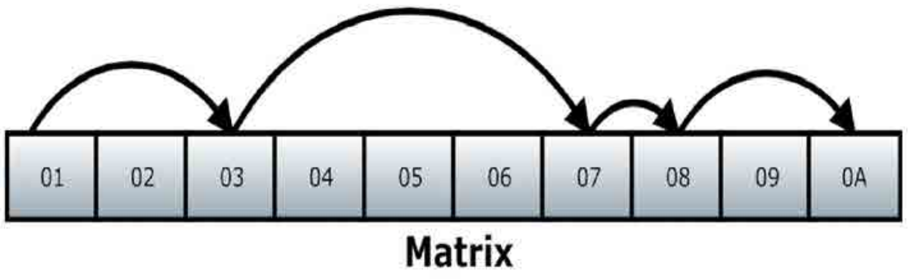


Figure 1.9: Conventional Load Store Access

1.3.1.1.1 Load Store Unit: The LSU is the conventional way to access data from the memory shown in Figure 1.9. The LSU uses a load queue and a store queue. The master core is responsible for managing and controlling main memory data movements. In a multi-core processor, all cores request to access main memory the master core takes memory requests and allow single core to transfer data. Such data transfers have fine granularity and can access complex access patterns. One load or one store operation can be issued per clock cycle. A single instruction transfers a single data element, as the processor running the application performs data management it takes less time for address generation of memory accesses. The LSU data access has low throughput each data requires multiple load store instructions.

```
Read_DMA(local_memory,main_memory,transfer_size);
Write_DMA(main_memory,local_memory,transfer_size);
```

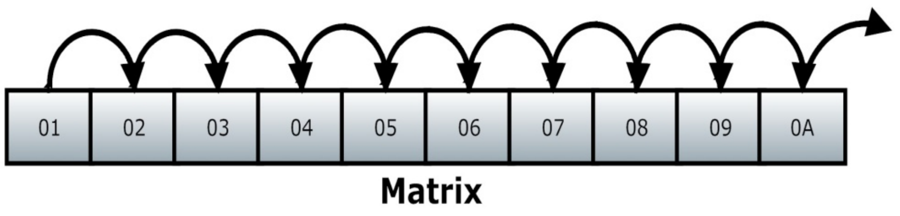


Figure 1.10: Direct Memory Access

1.3.1.1.2 Direct Memory Access: A Direct Memory Access (DMA) uses a separate hardware block that transfers blocks of data to/from *local memory* and *main memory*. The processor initiates the transfer by supplying the data transfer description to the DMA. A single DMA instruction manages one stream of contiguous data. DMA minimal descriptor contains *scratchpad* memory, *main memory* addresses and transfer size of the memory access with unit stride access that is not efficient to access complex memory patterns. This is shown in Figure 1.10. When accessing non-contiguous memory locations, the memory system requires multiple DMA requests which generate address management delays. In such cases, the task of DMA becomes significantly more complicated.

1.3.1.1.3 Scatter Gather DMA: A Scatter Gather DMA (SGDMA) is used to transfer complex data accesses. The SGDMA merges non-contiguous memory accesses to a continuous address space. The core reads a series of DMA instructions that specify the data to be transferred. In this way, the SGDMA stores the starting addresses of all the memory accesses. After the DMA transfer operation starts, the SGDMA core automatically takes the start address of the next data transfers after the previous transfer of memory is completed. Using SGDMA, the memory system can transfer efficiently non-contiguous blocks of memory.

1.3.1.2 Bus Delay

The shared communication buses are problematic both in latency and bandwidth. A shared bus has long electrical wires, and if there are several potential slave units, it make the bus even slower. Furthermore, the fact that several units share the bus fundamentally limits the bandwidth seen by each core. When the number of components attached to the bus increases, the physical capacitance on the bus wires grows, and, as a result, its wiring delay grows even further. The multi-core bus unit requires a master core which manages data movements, complex on-chip/off-chip bus matrix architecture and direct memory controllers. The bus faces delays such as master/slave arbitration, bus switching time and balancing workload. The *Memories* are mostly connected with analog high speed streaming interfaces. Numerous aspects that increase delay while accessing external memories include interface synchronization, signal quality, and interface timing.

1.3.1.3 Scheduling Delay

The *Scheduling Delay* is generated while executing data transfer requests. This delay includes the time to schedule transfer requests, scheduler wait time and execution time. Multiple cores may need to use the bus at the same time, so the scheduler decides which processor core controls the bus. A conventional multi-core scheduler executes data transfer requests based on the application priority and request arrival time. The applications having low priority wait until higher priority applications finish and the applications having same priorities are executed as First In First Out (FIFO) which affects the fairness. At run-time, the applications having multiple small memory requests generate scheduling delay and stall the system. Whereas an application request with large data transfer size occupies the whole memory bandwidth and stalls other applications.

1.3.1.4 Memory Delay

Memory Delay includes *Main Memory* access time due to address decoding, internal delays in driving long bit lines, selection and refresh logic of SDRAMs. The parameters *n-banks, WM, FME, DR*, and *BL* (shown in Figure 1.11) are used to describe the SDRAM memory architecture. Where `n-banks` stands for the number of banks, `WM` is the width of the data bus in bytes, `FME` represents the clock frequency of memory in MHz, `DR` defines the number of data words that can be transferred during a clock cycle, and `BL` is the word length of programmed burst. To control the SDRAM system, specific commands are sent to the memory port according to the protocol. The

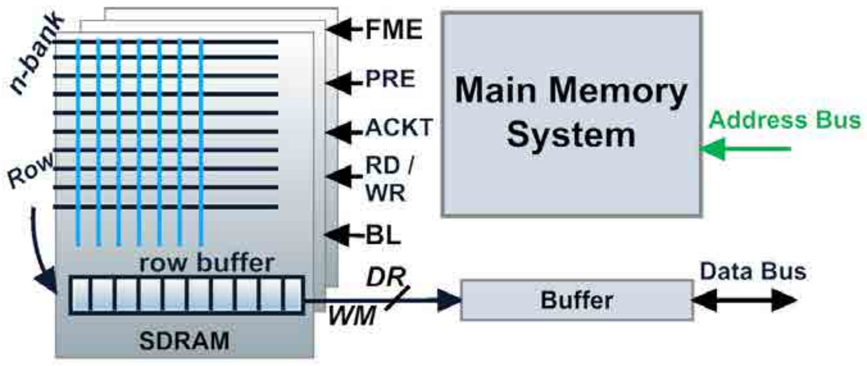


Figure 1.11: SDRAM Memory Architecture

SDRAM communications protocol includes six command signals which are activate (ACKT), read (RD), write (WR), precharge (PRE), refresh (REF), and no-operation (NOP). The activate command is supplied with a row and a bank as an argument, informs the selected bank to copy the requested row to its row buffer. Once the requested row is opened, column can be accessed by read and write bursts having different burst length (`BL`). The read and write commands involve a separate bank, row, and column address lines. The precharge command corresponds the activate command to place the contents of the row buffer back to its location in the memory. An auto-precharge flag with a read and write command is inserted to automatically precharge at early possible moment after the data transfer. This has the benefit that the next accessed row will be opened faster without causing contention on the shared system bus. The accesses to an SDRAM can be classified into three categories, each with different timing.

- Row hit: When the data is available in the *row buffer* of SDRAM the *Main Memory System* issues only a read (RD) or write (WR) command to the SDRAM bank.
- Bank hit: When a row is not opened in the *row-buffer* and *Main Memory System* needs first to issue an activate (ACKT) command to open the required row, followed by a read/write command.
- Bank conflict: When data access requires data from a bank that is not opened. This requires *Main Memory System* to first open a bank by issuing a precharge (PRE) command, then activate (ACKT) the required row, and then issue a read-/write (RD/WR) command.

1.3.2 Access Patterns

The memory access patterns are categorized into *regular* and *irregular* access patterns.

1.3.2.1 Regular Access Patterns

Regular access patterns have sequential access pattern, and each transfer has a constant stride between two consecutive memory addresses (an example is shown in Figure 1.12 (a)). The access pattern addresses are predictable at compile-time and therefore little

```
data[1024];
for(int x=y;x<100;x=x++)
{
read=data[x];
compute(read); }
```

(a)

```
data[1024];
for(int x=0;x<5;x=x++)
{
read=data[factorial(x)];
compute(read); }
```

(b)

```
data[1024];
addr=runtime_input();
for(int x=0;x<5;x++){
read=data[factorial(x)+addr];
compute(read); }
```

(c)

```
data[100];
for(int x=0;x<100;x=x++)
{
read=data[read+x];
compute(read); }
```

(d)

Figure 1.12: Examples of: (a) Regular access pattern (b) Irregular known access pattern (c) Independent unknown irregular access pattern (d) Dependent unknown irregular access pattern

processing is required to generate addresses at run-time. A multiple linked strided stream can be used to access a complex pattern (e.g. 3D stencil, 2D tile etc.).

1.3.2.2 Irregular Access Patterns

The *Irregular patterns* have no fixed strides and are further divided into three categories: *known*, *independent unknown* and *dependent unknown* access patterns. The *known* data patterns have irregular memory accesses – no fixed stride between consecutive accesses – but the address value of each memory access can be determined at compile-time (an example is shown in Figure 1.12 (b)). Computation and memory access are performed in parallel for *known* data patterns. The *independent unknown* and *dependent unknown* are exemplified in Figures 1.12 (c) and 1.12 (d) respectively. These memory patterns can not be predicted at compile-time and addresses can be generated at run-time. An address of an *independent unknown* pattern is not dependent on the other memory accesses of the pattern. Therefore, *independent unknown* pattern addresses are generated in parallel with computation. The *dependent unknown* pattern is dependent on a previous memory access from the same pattern. The memory address is generated by the compute unit. Therefore, the access patterns can not be processed in parallel with computation, and the memory controller has to wait for the address of the memory access.

1.3.3 Scheduling

The scheduling strategy of multi-core data transfers are categorized into three types, *Symmetric Scheduling*, *Asymmetric Scheduling* and *Run-time Scheduling*.

1.3.3.1 Symmetric Scheduling

Symmetric Scheduling strategy queues incoming data transfer requests in the queue buffer in the order that they arrive and execute the data transfer request in FIFO. The *Symmetric Scheduling* strategy treats all available data transfer requests as equal resources. Quality of service (QoS) management is one of the interesting features of *Run-time Scheduling* strategy, which relies on monitoring available hardware resources and prioritizing data transfer requests. The *Symmetric Scheduling* strategy affects the QoS since the applications having long data transfers occupy whole bandwidth.

1.3.3.2 Asymmetric Scheduling

Unlike *Symmetric Scheduling* strategy, *Asymmetric Scheduling* emphasizes partitioning and role specialization for available processor cores. *Asymmetric Scheduling* deals with heterogeneous multi-cores with different capabilities, e.g. different instruction set architectures or different execution speeds. The *Asymmetric Scheduling* strategy emphasizes on priority and incoming requests of the data transfer request. The priority value sets the policy for how to share the system resources among available requesting core. The scheduling policies are configured statically at program-time and are executed by hardware at run-time. Asymmetric scheduler balances the waiting time of multi-core requests and provides controlled data communication between processors and memories.

1.3.3.3 Run-time Scheduling

Multi-core system share multiple hardware resources in the memory subsystem. If priorities of multi-core system are not appropriate, some applications can be delayed significantly while others are unfairly prioritized. *Asymmetric Scheduling* proposed mechanisms for each resource at compile-time. Such scheduling policies make suboptimal decisions, leading to low fairness and loss of performance. The application with multiple irregular requests generates scheduling delay and stalls the system. Whereas an application request with large data transfer size occupies the whole memory bandwidth and stalls other applications. The *Run-time Scheduling* strategy reorganizes data transfer requests with respect to run-time specification and available resource and specification. Like *Symmetric and Asymmetric Scheduling* strategies *Run-time Scheduling* strategy prioritize memory requests at run-time, unlike *Symmetric and Asymmetric Scheduling* the prioritizes are assigned at run-time based on system specification (e.g. transfer size and number of requests).

1.4 Thesis Contributions

In this section the main contributions of this thesis are categorized into four parts with the list of papers associated with PMC. The first part lists single-core designs with regular data access patterns. The second part of the work studies the multi-core based design that supports regular data transfer patterns. The third part describes the single-core design with irregular and regular data transfer patterns. The last part is based on the multi-core system with regular and irregular memory pattern support.

1.4.1 Regular Data Access Patterns For Single Core

1.4.1.0.1 —(1)— We propose a Pattern-based Memory Controller (PMC), which can work with any SoC architecture and stand-alone HPC kernel without modifications to the microprocessor system. The PMC improves the memory-processor data access bottlenecks by allowing to fetch *complex regular patterns* without processor intervention. Furthermore, to improve the on-chip bandwidth, this work proposes hardware tiling based on programmable domain decomposition. The proposed controller can be programmed by a microprocessor using High Level Language or directly from an accelerator using a special command interface. This contribution was published in WRC11 [34] and ARC12 [35].

1.4.1.0.2 —(2)— We introduce PMC based bus controller for low cost and low power graphics system. The system takes high resolution images and supports video at higher frame rate without the support of a processor. The PMC graphics system provides strided, scatter/gather and tiled access pattern that eliminates the overhead of arranging and gathering address/data. This contribution was initially published in ARC14 [36]. An extended version of the same work has appeared in IJCAD14 [37].

1.4.1.0.3 —(3)— We have suggested PMC for vector processor architectures that manages memory accesses without the support of a scalar processor. Furthermore, to improve the on-chip data access a specialized memory manager are integrated that efficiently access, reuse, align and feed data to the vector processor. A *Multi DRAM Access Unit* is used to improve the main memory bandwidth which manages the memory accesses of multiple *SDRAM*s. This work is presented in ASAP14 [38] and [39]

1.4.2 Multi Core System with Regular Data Patterns

1.4.2.0.4 —(1)— We have proposed PMC for multi-ASHA environment. The PMC improves the HLS based multi-ASHA system performance by reducing accelerator/processor and memory speed gap, and schedules/manages complex memory patterns without processor intervention. This work is presented in FPL12 [9]. An extended version of the same work has appeared in JPDC14 [40].

1.4.2.0.5 —(2)— We have integrated PMC in HLS based multi-ASHA environment, that handles regular and complex pattern requests of multiple *ASHA*s. The proposed environment can be programmed by a microprocessor using High Level Language (HLL) API or directly from an accelerator using a specific command interface. The AMC HLS based multi-ASHA system is evaluated with memory intensive accelerators. The work is presented in FPT12 [16] and ParCo14 [41].

1.4.3 Single Core System with Irregular Data Transfer Support

1.4.3.0.6 —(1)— We have suggested support in hardware for both regular and irregular memory access patterns. This is achieved with an on-chip (1D/2D/3D) memory unit, efficient memory management and a memory access policy to access DRAM for FPGA architectures. The technique enhances the system performance by reducing the processor/memory speed gap as well as accessing irregular access patterns and allocating them in local memory without processor intervention. A high-level language API is provided to program the system. Memory patterns are arranged in descriptors at program-time, at run-time PMC accesses them without adding memory request and address generation delay and places them in on-chip *Specialized Scratchpad Memory*. This reduces the processor/memory communication cost, improves utilization of memory hierarchies and efficiently prefetches complex/irregular access patterns. This contribution is from our paper accepted in HPCS14 [42] and FPGA14 [43]

1.4.4 Multi Core System with Irregular Data Transfer Support

1.4.4.0.7 —(1)— We propose an efficient scheduler and intelligent memory manager in PMC, which proficiently handles data movement and computational tasks. The proposed PMC system improves performance by managing complex data transfers at run-time and scheduling multi-cores without the intervention of a control processor nor an operating system. PMC has been coupled with a heterogeneous system that provides both general-purpose cores and application specific accelerators. This contribution is accepted in FPT14 [44].

1.4.4.0.8 —(2)— We propose memory access pattern based scheduling policies in PMC. The enhanced PMC organizes data accesses in descriptors, prioritizes them with respect to the number and size of transfer requests and manages DRAM banks. This contribution is presented in FPL14 [45] and ReConFig14 [46].

1.5 Thesis Organization

This thesis document consists of twelve chapters in total and are subdivided into 6 parts. Part 1 contains chapters 1 and 2. This – first – chapter gives a general introduction to the work, discusses the problem and motivation and the chapter 2 explains proposed architecture. Part 2 includes the chapters 3, 4 and 5 and covers the management and generation of regular memory access pattern for single core system. Part 3 covers the details on the regular memory access for heterogeneous multi-core system and can be found in chapters, 6 and 7. Part 4 includes chapter 8 which contains information on the irregular memory patterns for single core system. Part 5 has chapters 9 and 10 and presents irregular memory patterns for multi-core system. Part 6 contains chapter 11 and 12 which covers the related work, conclusions and the future work respectively.

2

Proposed Heterogeneous Multi-Core Memory System

In this work, we present a new high performance intelligent memory controller which helps the Heterogeneous Multi-Core architecture to make best use of the memory. Our design is a Pattern based Memory Controller (PMC) that aims at improving the performance of compute intensive applications. The PMC supports regular and irregular memory access patterns using the memory pattern descriptors. Irregular access patterns are supported by using a specialized data structure that describes irregular patterns and implements a hardware communication protocol between the cores and the memory system. The design describes regular and irregular access patterns of each processing core at program-time using a separate *Descriptor Memory*, which reduces the on-chip communication time, run-time address generation overhead and efficiently accesses data from the main memory. This controller uses complex irregular, strided 1D, 2D, 3D patterns and automated blocking for data accesses. The PMC provides a specialized on-chip memory to each processing core that improves system performance by prefetching complete patterns and reduces the on-chip bus switching and network/bus latencies. PMC can prefetch complete patterns, having multiple scatter/gather operations, into its scratchpad memories that can be accessed either by a microprocessor or an accelerator. Our proposed system provides features of parallelism (independent read/write operation), flexibility (support for irregular/complex access patterns) and programmability (high level programming language support). The salient features of the proposed PMC architecture are:

- The memory system can operate as a stand-alone system, without support of a master processor or the operating system.
- PMC has support for both regular and irregular memory access patterns using the memory pattern descriptors thus reducing the impact of memory latency.
- PMC is capable of handling both general purpose processors and application specific accelerator cores using event driven handshaking methodology.
- PMC uses hardware scheduler that handles memory requests of a heterogeneous multi-core system provides precise timing and allows scheduling mode to be changed at run-time.
- The controller intelligently schedules multiple data access requests at run-time with respect to their data transfer size and memory requests.
- The memory system uses a *Specialized Local Memory* that tailors local memory organization and maps complex access patterns.
- PMC memory manager efficiently accesses, reuses and feeds data to the computing unit and handles complex memory accesses at run-time.
- The main memory unit improves the performance and reduces system and external bus latencies by efficiently prefetching complex/irregular patterns.

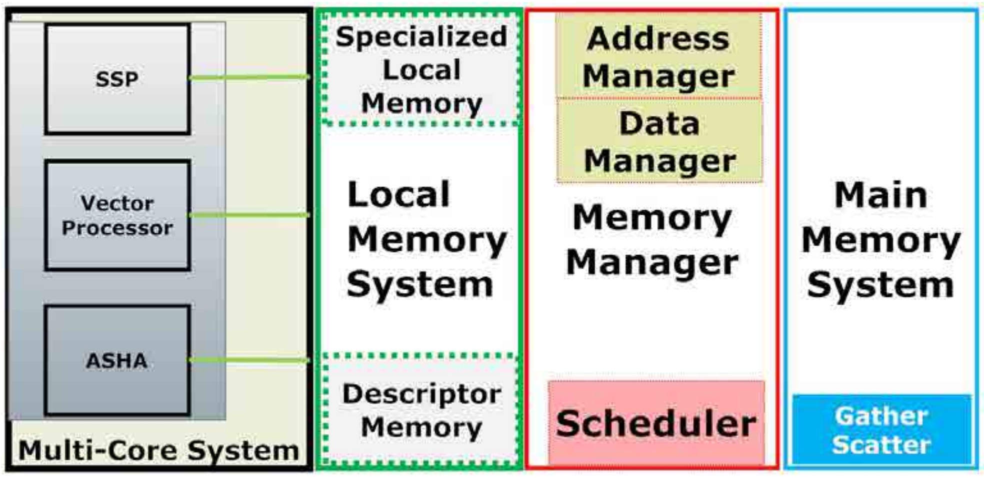


Figure 2.1: PMC based Multi-core Heterogeneous Architecture

- The concurrent memory support for multi-core system allows to integrate PMC in a parallel programming model.

A simplified form of our target architectural model is shown in Figure 2.1, including the memory system as well as the Heterogeneous *Multi-core System*, which executes the applications. The *Multi-core System* can have general purpose processors, application specific accelerator cores, a programmable accelerator such as vector processors or a combination of all types. The system may include more than one core of each type. The *Bus System* provides a link between the *Main Memory System* and the *Multi-core System*. Support for irregular access patterns is provided with a specialized data structure that describes irregular patterns, and a hardware communication protocol is implemented between the multi-core system and PMC. The PMC *Local Memory System* stores both data and the access pattern descriptors in *SSM* and *Descriptor Memory* respectively. Each processing core has separate *Specialized Local Memory* and *Descriptor Memory* blocks. Each *Specialized Local Memory* rearranges the data-set as required by the corresponding application. The descriptors are programmed at compile-time with the specialized descriptions of memory access pattern and the priority of the patterns. Memory accesses are arranged in the pattern descriptors at program- and run-time to reduce *Main Memory* data access latency. At run-time, the PMC *Scheduler* receives multiple memory read/write requests from the *Multi-core System* and selects a processing core, depending upon its priority level and scheduling policy. The *Scheduler* also performs run-time automated scheduling depending upon the core data requests and data transfer size. The *Scheduler* forwards the memory request to the *Memory Manager*. The *Memory Manager* takes a single or multiple descriptors and reads/writes the data pattern. It is divided into the Address Manager and the Data Manager. The *Memory Manager* reorganizes data for two reasons: (i) Data arrangement for distribution in parallel of computation and (ii) Transformation of data from a memory default arrangement to an application required arrangement. The *Main Memory System* is responsible for transferring data between SDRAM and the specialized memory. It gathers multiple memory read/write requests and maximizes the reuse of opened SDRAM banks to decrease the overhead of opening and closing rows. This chapter is subdivided into two sections: *PMC Architecture* and *Programming Model*.

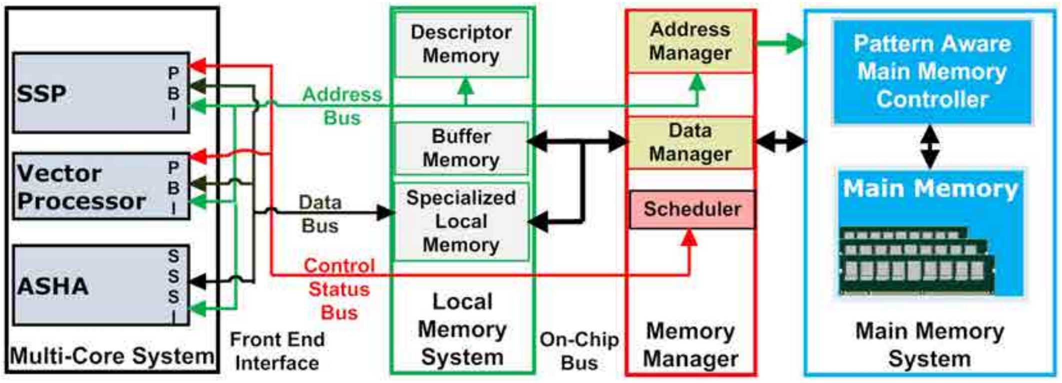


Figure 2.2: PMC: Front End Interface

2.1 Programmable Memory Controller Architecture

In this section, we explain the PMC architecture. The PMC can have reduced or limited features for different types of evaluation structure e.g. a single core system with a regular pattern uses bus arbiter as scheduler (see Section 2.1.3.3). The block diagram of PMC having all units and connections is shown in Figure 2.3. The PMC reduces the multi-core system delays and aims at improving the system performance. The section is further subdivided into sections: *Bus System*, *Local Memory System*, *Memory Manager*, *Main Memory System*. and *PMC System*.

2.1.1 Bus System

The PMC *Bus System* communicates the cores with main memory. It interfaces general purpose (RISC), vector and *ASHA* cores It provides the pattern based switching for regular and known irregular access patterns and circuit switching for unknown irregular access pattern. The *Bus System* is further divided into two sections: *Front-End Interface* and *On-chip Bus*.

2.1.1.1 Front-End Interface

The Front-End Interface demonstrates support of the PMC for different types of interfaces.

- Source Synchronous Streaming Interface

- Processor Bus Interface

2.1.1.1.1 Source Synchronous Streaming Interface (SSSI) The Source Synchronous Streaming Interface (SSSI) shown in Figure 2.2 is used to supply high-speed data to application specific cores such as hardware accelerators. SSSI uses only address and data buses. The synchronous handshaking protocol is applied to *request* and *grant* Data. Transfer of *Data* is performed according to the *Main Memory* clock.

2.1.1.1.2 Processor Bus Interface (PBI) As the number of processing cores and the capacity of memory components increase the system requires a high speed bus interconnection network that connects the general purpose processors and memory modules. The *Processor Bus Interface* (*PBI*) as shown in the Figure 2.2 contains *Data Bus*, *Address Bus*, *Control* and *Status Bus*. The *Data* and *Address* buses are used to read/write high-speed data to/from the PMC and the *Multi-core System*. The PMC *PBI* single *Data* bus has a peak bandwidth of 1.6 GB/s as it can operate at maximum 200 MHz clock having data bus width of 64-bit. The *Address Bus* is used to identify the locations to read or write data from memory components or processing cores. The *Address Bus* of the functional units is decoded and arbitrated by memory manager. The *Control Bus* controls the data movement and carries information of data transfers. It holds the request/grant & select/ready signals that are used to communicate with scheduler and memory manager. The bus is also used to move data between PMC descriptors and the *Multi-core System* registers. The *Status Bus* holds signals of multiple sources that indicate data transfer requests, acknowledgement, wait/ready and error/ok messages.

2.1.1.2 On-Chip Bus System

The *On-Chip Bus System* (shown in Figure 2.3 provides connection between data ports of *Local Memory* and *Main Memory*. The data buses are unidirectional and address buses are decoded and arbitrated by *Address Manager* and *Scheduler* respectively, both part of the *Memory Manager* (see Section 2.1.3). The *On-Chip Bus System* relies on the PMC pattern descriptors to connect the data port between different components. Depending upon the address space of a data access pattern that defines the requesting core, the *Scheduler* creates a link between related processing core and memory

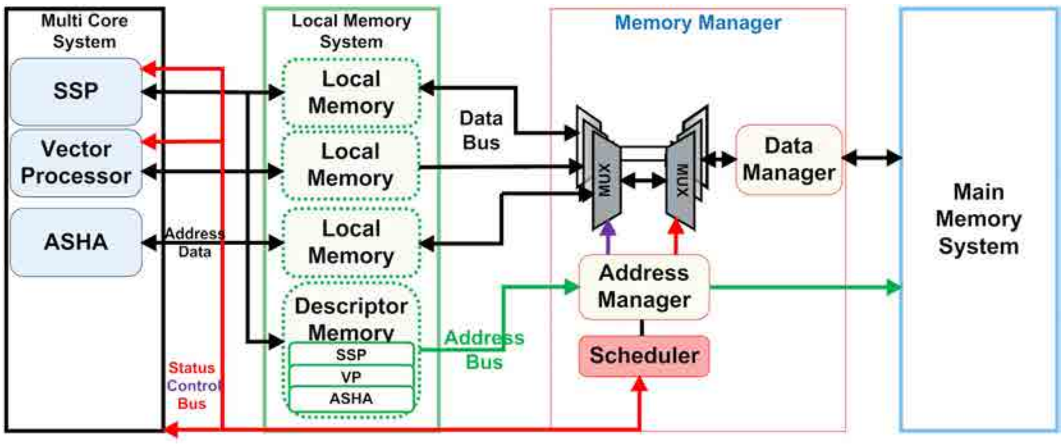


Figure 2.3: On Chip Bus System

system and store requests of other processing cores units from interrupting the data transactions.

The shared buses cause inefficient utilization of multi-core system, whereas the multi-layer bus architecture requires complex interconnections including multi-port arbiters with long and wide global metal wires, leading to high power consumption. To manage the performance and power, the *On-Chip Bus System* can be programmed for different number of layers and transfer data in the form of patterns. Transferring data in the form of patterns reduce bus switching and improve on-chip bus throughput. The *On-Chip Bus System* can be reconfigured into shared or multi-layer bus system based on access pattern descriptions. Depending upon the available on-chip data buses the PMC executes multiple data access pattern requests in parallel and queue the rest of the requests. For example, if there are two data buses then the PMC executes two data patterns in parallel.

2.1.2 Local Memory System

The PMC *Local Memory* consists of *Descriptor Memory*, *Buffer Memory* and *Specialized Local Memory*.

2.1.2.1 Descriptor Memory

The PMC uses pattern descriptors to organize the *Regular* and *Irregular* access patterns. These pattern descriptors reduce the overhead of run-time address generation and data management. The PMC *regular/Irregular Descriptor Memory* contains fixed length data words with a number of attribute fields that describe the access patterns. The set of parameters for a *regular memory* descriptor block (shown in Figure 2.4) includes *Local Address*, *Main Address*, *Priority*, *Size* and *Stride*. The address parameters specify the starting addresses of the source and destination locations. The *Priority*, when combined with other priorities, determines the order in which memory access is entitled to be process. *Size* defines the number of data elements to be transferred. In each stream, the first data transfer uses addresses taken by the descriptor unit and for the rest of the transfer the address is equal to the address of the previous transfer plus size of strides. *Stride* indicates the jump between two consecutive *Main Memory* addresses of a stream. Strides between two consecutive accesses are handled by the PMC without generating delay.

The parameters for *Irregular Descriptor Memory* accesses are also presented in Figure 2.4. The *Type* and *Offset* register fields define the category of irregular pattern and the location of the next element respectively. Thus, irregular patterns are represented by a chain of multiple descriptors. To reduce the size of the *Irregular Descriptor Memory* for same data pattern, following descriptors hold only *Size*, *Stride* and *Offset* registers. The *Offset* is used to point the first element of the access pattern by adding it to the *Main Address*. Depending on the data structure to access, there can be multiple *Offset* registers per entry in the *Descriptor Memory*. For example, a single *Offset* register describes a 1D linked list structure and a binary-tree access pattern uses two *Offset* registers. The *Type* register can hold three values which are: `NULL`, `known` and `unknown`. The `NULL` type indicates that this is the end of a particular

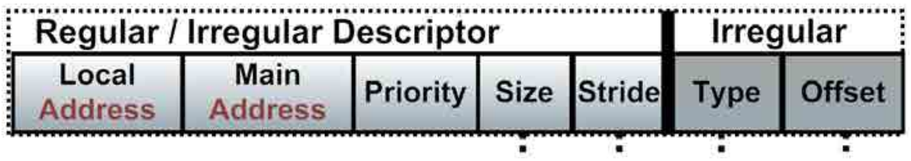


Figure 2.4: Descriptor Memory Structure: Regular and Irregular Access Patterns

pattern and signifies that PMC should stop accessing data for this pattern. The `known` type indicates that the *Offset* register holds the address for the next access. These addresses are generated and placed in the *Descriptor Memory* at compile-time therefore it removes the overhead of address generation at run-time. The `unknown` type tells PMC to gather the next address at run-time from the processing core. The *dependent unknown* patterns are not programmed at compile-time, PMC control bus uses separate (*request* and *grant*) signals (shown in Figure 2.3) to read addresses from the compute unit. C/C++ calls such as send() and send_irregular() are used to program the descriptor blocks (see Section 2.2).

2.1.2.2 Buffer Memory

The *Buffer Memory* architecture implements the following features:

- Data realignment to match different access patterns. It aligns data when input and output data access elements are not the same.
- Load/reuse/update to avoid accessing the same data multiple times (uses the realignment feature).
- In-order data delivery. In cooperation with the Memory Manager (see Section 2.1.3) that prefetches data, it ensures that the data of one pattern is sent in-order to the processing core.

The *Buffer Memory* is managed by the *Memory Manager*. The *Buffer Memory* holds three buffers which are the *load buffer*, the *update buffer* and the *reuse buffer*. The *Buffer Memory* transfers data to the *Multi-core System* using the *update buffer*. The *load* and *reuse* buffers are used to manage the *Specialized Scratchpad Memory* (SSM) data (see Section 2.1.2.3). For example, if the core of *Multi-core System* requests data that has been written recently then the *Buffer Memory* performs on-chip data management and arrangement.

2.1.2.3 Specialized Scratchpad Memory

The system provides a *Specialized* on-chip memory that improves system performance by prefetching complete patterns by using multiple scatter/gather operations. As a conventional *Local Memory* system, the *Specialized Scratchpad Memory* (*SSM*) accesses the whole data pattern as a cache line and temporarily holds data to speedup later accesses. Unlike the *Cache Memory* system, the accessed block can have data of noncontiguous memory locations and is deliberately placed in the *SSM* at a known location, rather than automatically cached according to a fixed hardware policy. At run-time, the PMC allocates a single *Descriptor* block for each processing core. Unlike the *Local Memory* system which transfers an aligned block of data for each data miss, the PMC transfers only the missing data by scattering/gathering operation at run-time and transfers irregular blocks of data.

The PMC *SSM* is directly connected to the compute unit and provides single cycle data access. Like a cache, the *SSM* temporarily holds data to speedup later accesses. Unlike a cache, data is deliberately placed in the *SSM* at a specified location, rather than automatically cached according to a fixed hardware policy. Depending upon the available block RAMs, the *SSM* can be organized into multiple banks. Each bank has two ports (PortA & PortB), that allows the compute unit to perform parallel reads and writes. In order to exploit parallelism better, the banks of *SSM* are organized into a multi dimensional (1D/2D/3D) architecture to map the kernel access pattern on the *SSM*.

For example, a generic 3D-stencil structure is shown in Figure 2.5. When n=4, 25 points are required to compute one central point. This means that to compute a single element 8 points are required in each of the x, y and z directions, as well as

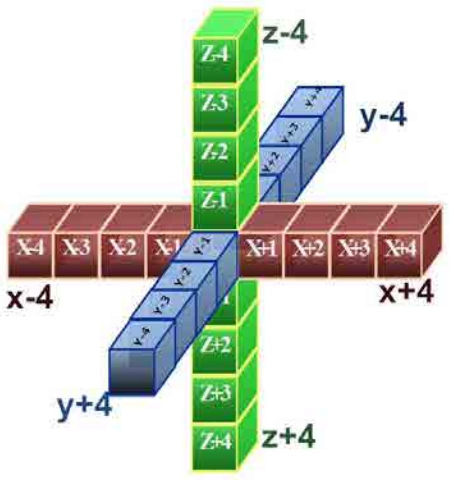


Figure 2.5: 3D-Stencil

one central point. A 3D *SSM* architecture[1] that accommodates the stencil is shown in Figure 2.6. The 3D-stencil occupies 9 different banks; the central, x and z points are placed on the same bank (y), and each y-point needs a separate bank (y+4 to y-4). The *SSM* takes 9 cycles (2 points per bank per cycle) to transfer the 17 points of bank (y), and the points from banks (y+4) to (y-4) are transferred in a single cycle. Each bank read/write operation is independent of each other and can be performed in parallel. Therefore, the *SSM* takes 9 cycles to transfer the 25 points of a single stencil. Whereas with a regular memory structure, PMC would take a minimum of 25 cycles to transfer the 25 points of a single stencil. To fully utilize all banks and to reduce access time, multiple 3D-Stencils are placed on the 3D *SSM*.

In our current evaluation on Xilinx Virtex-7, the *SSM* has 64 banks and each bank holds 128 $\times$ 128 Bytes (32 $\times$ 32 words) (row $\times$ column). Each bank uses a single BRAM (`1x 36 Kb`), which is controlled by a separate BRAM controller and has a different base address. In the current architecture, number of banks is fixed but row

[1]Please note that we do not refer to 3D-stacked memories but a memory with 3D from the point of view of access.

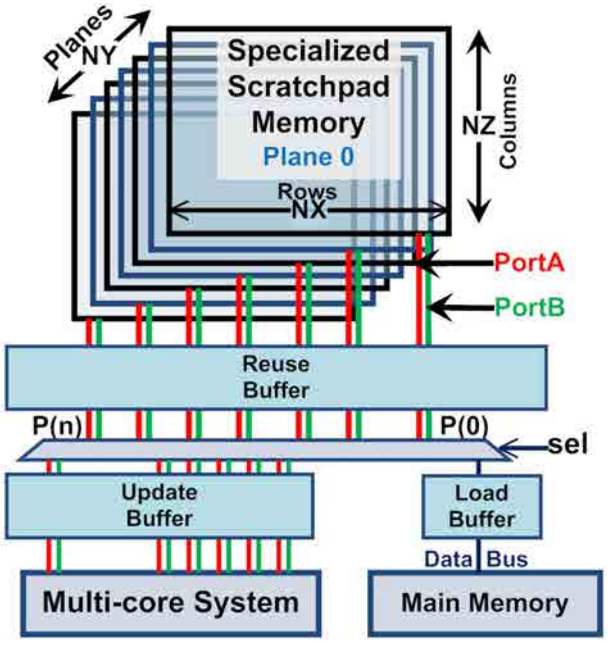


Figure 2.6: Buffer Memory Organization and Specialized Memory Structure

and column size can be changed depending upon the third dimensions of the data set. A single or two dimension data sets are placed in a single bank and can use single or multiple BRAM/s. For a 3D data set, depending upon the 3rd dimension, up to 64 banks can be used for a single core. Depending upon the dimensions of data set, the *SSM* can be arranged by re-programming the PMC *Descriptor Memory*. The PMC accesses and places data in tiles if the data set is larger than the *SSM* structure.

2.1.3 Memory Manager

The PMC *Memory Manager* organizes and rearranges multiple noncontiguous memory accesses simultaneously which reduces read/write delay due to the control selection of SDRAM memory. The PMC *Memory Manager* applies protection at the access pattern level e.g. a pattern can be read/written by a core for which it is allocated and not by the other cores. PMC keeps the knowledge of memory as to whether or not a certain memory area is in the *SSM*. This knowledge allows the PMC to manage the placement of memory as well as reuse and share already accessed memory. The *Memory Manager* controls the local memory system (descriptor and scratchpad memory). It

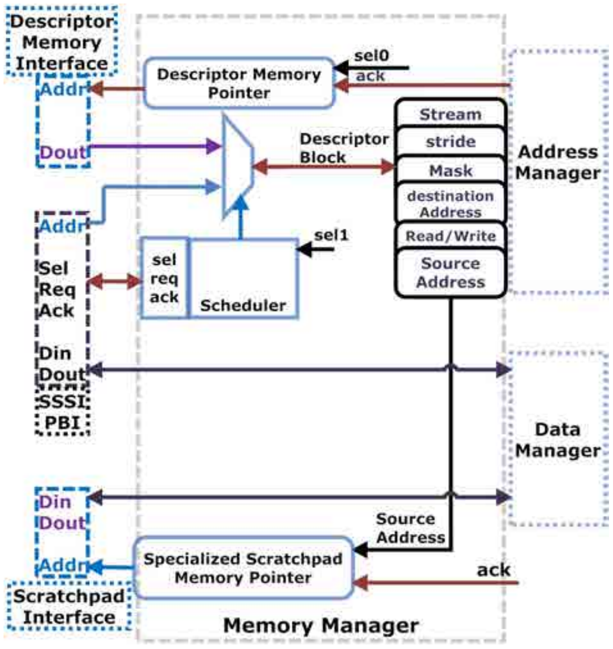


Figure 2.7: PMC : Memory Manager

handles a *Descriptor Memory* Pointer (DMP) and a *SSM* Pointer (SSMP) as shown in Figure 2.7. The DMP holds the address for the next descriptor block and provides this address to the *Descriptor Memory* to fetch a descriptor block. After completion of target data access, the address manager sends an `ack` signal to the DMP that requests the next descriptor block. The SSMP is responsible for generating addresses for the scratchpad memory. Depending on the application (multi-threaded) or hardware architecture (multi-ASHA), the scratchpad memory is divided into multiple buffers. The SSMP takes the source address (*Base Address*) from the descriptor controller and starts incrementing in it. The SSMP stops incrementing addresses, when the `ack` signal is granted by the *Pattern Aware Main Memory Controller* (see Section 2.1.4). The *Memory Manager* is further divided into three sections which are: the *Address Manager*, the *Data Manager* and the *Scheduler*.

2.1.3.1 Address Manager

The *Address Manager* fetches a single or multiple descriptors depending on the access pattern, translates/reorders in hardware, in parallel with PMC Read/Write operations and access data patterns of a *Processor Core*. Strides between two consecutive accesses are handled by the *Address Manager* without generating delay. The *Address Manager* uses one or multiple descriptors at run-time to describe the data access. Unlike the cache which always transfers an aligned block of data for each data miss, the *PMC Address Manager* accesses only the missed data for scratchpad by gathering address

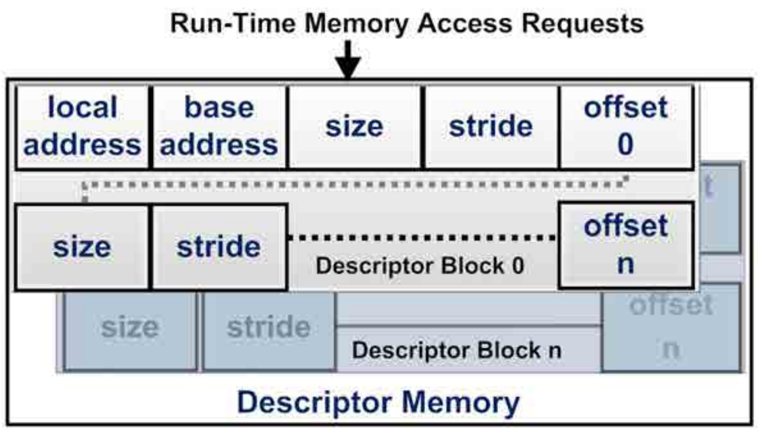


Figure 2.8: Descriptor Memory: Run-Time Managed Access Patterns

requests at run-time and can transfer irregular blocks of data. The *Address Manager* reads compile time generated addresses from *Descriptor Memory* and accesses complex data pattern. The *Address Manager* also manages run-time unpredictable memory accesses and places them in *Descriptor Memory* (Figure 2.8). The *Address Manager* takes memory address requests from a *Processor Core*, buffers them and compares the consecutive requesting addresses with the previous one. If the addresses of consecutive memory requests have constant strides, the *Address Manager* allocates a *descriptor* block by defining *Stride* and *Size* parameters. If the request has variable strides, then the *Address Manager* uses the *Offset* parameter of the *descriptor* that points at the random location of the *Main Memory*.

The structure of run-time *Address Manager* is shown in Figure 2.9. The *Address Manager Stride Detector* takes an address from the address bus and gives it to *reg 0* and *comparator 0*. The *comparator 0* compares current address value ($Address_t$) with the previous one ($Address_{t-1}$) and generates the stride which is given to *Pattern Controller*. The *Pattern Controller* compares two strides ($Stride_t$ and $Stride_{t-1}$) and checks if they are same. If they are same then it increment in *Size* register of *Descriptor Memory*

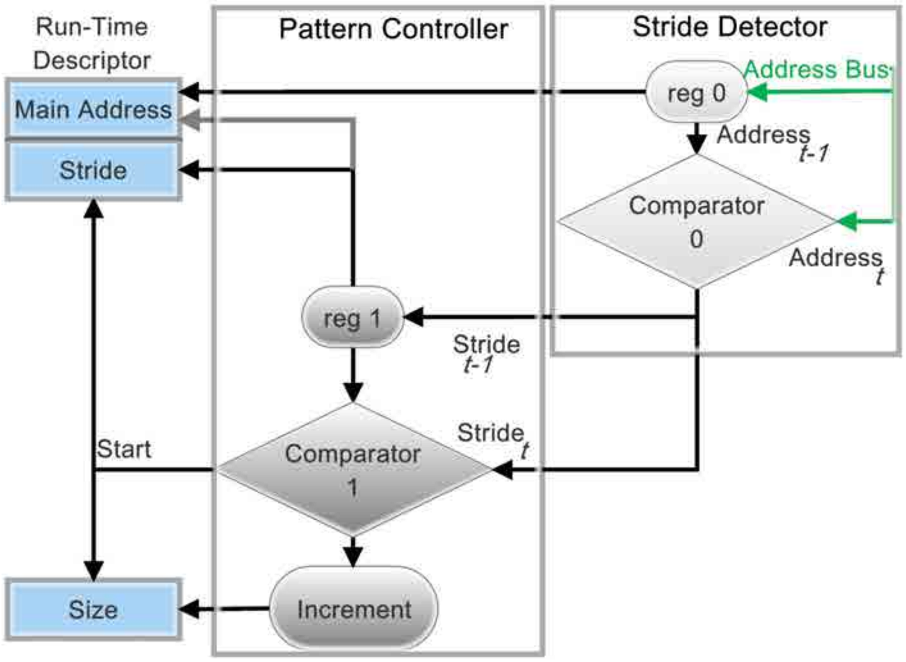


Figure 2.9: Address Manager: Run Time Managed Descriptor Block

and if strides are not same then it generates a `start` signal. The *Descriptor Memory* stores first value of *reg 0* and *reg 1* in *Main Address* and *Stride* respectively. Once a `start` signal is generated a new *Descriptor Memory* block or *Offset* is allocated for the requesting source.

2.1.3.2 Data Manager

The PMC data management plays a key role in managing and allocating data for the application kernel. The *Data Manager* is used to access non-contiguous data in a contiguous pattern without generating extra delay. It takes one or multiple descriptor blocks and transfers the appropriate data between the *SSM* and main memory. At run-time the *Data Manager* executes these descriptor blocks in hardware, typically overlapping address generation and memory requests with computation in the processing unit. For *dependent unknown irregular* memory accesses the *Data Manager* uses the *request* and *grant* signals to communicate with the processing unit. The communication process does not affect the other PMC units, and they keep working in parallel.

In general, the efficient handling of irregular access patterns implies a high overhead for data and address management, because each data transfer uses a different data arrangement. The *Data Manager* manages the access patterns of application kernels with complex data layouts. The *Data Manager* reads/writes data from/to the main memory and keeps information of the data currently stored in the data memory and reuses data when possible. It increases the *c/a* ratio ($computed_{elements}/accessed_{elements}$) by organizing and managing the complex memory patterns. The *Data Manager* reads a single descriptor and accesses its elements using the *load buffer* (shown in Figure 2.6). The *reuse buffers* temporarily holds loaded elements for further reuse. For each

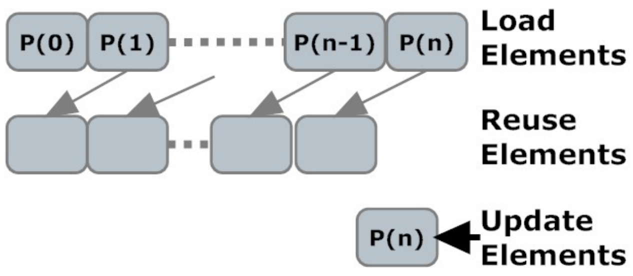


Figure 2.10: Load, Update and Reuse of FIR Access Pattern

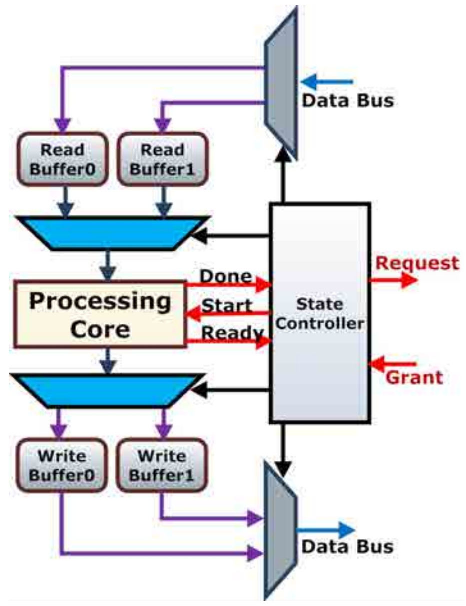
memory access descriptor, the *Data Manager* compares the requested elements with the elements placed in the *reuse unit*. If the elements are found in the *reuse unit*, the *Data Manager* uses them again and requests the rest. The *update buffer* transfers the elements addresses which are not present in the *reuse buffer* to the *Main Memory System* (see 2.1.4). The *update buffer* of the *Data Manager* rearranges available elements in the *SSM* and updates new loaded elements. For example, the *Data Manager* for an n-tap FIR filter (shown in Figure 2.10) loads all elements of an access pattern, which are required for a single $Computed_{elements}$. After transferring the first memory access (*elements*), the *Data Manager* performs reuse and update operations. The *Data Manager* keeps reusing previous accessed *elements* and updates the remaining *elements* required. For an FIR filter, if n=16, then 16 elements are required to compute one point, and without the *Data Manager c/a* ratio is *1/16* = 0.062. The *Data Manager* improves *c/a* ratio to 1 by reusing accessed points.

2.1.3.3 Scheduler

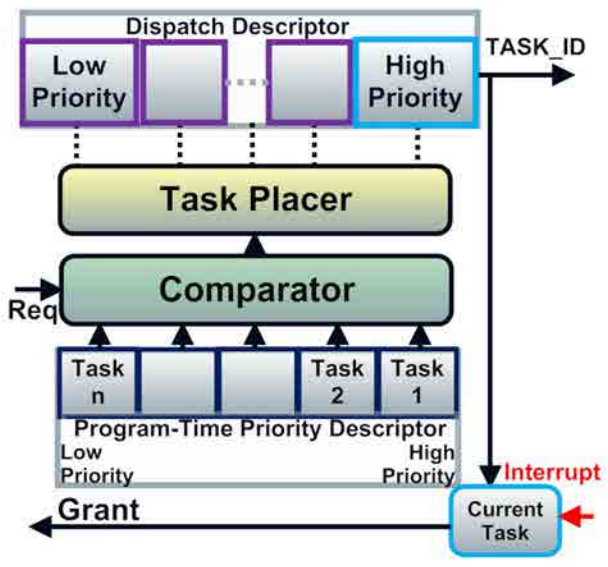
The PMC *Scheduler* handles heterogeneous multi-cores and multiple input/output interfaces such as source synchronous streaming interface and bus interface. The design describes access patterns of each processing core using the *Descriptor Memory*, which reduces the input/output interfaces and multi-cores communication time. At run-time, the PMC *Scheduler* receives multiple memory read/write requests from the *Multi-core System*, selects a core and create a link depending upon its priority level and scheduling policy. The *Scheduler* also perform run-time automated scheduling depending upon the core data requests and data transfer size. As conventional multi-core system scheduling, *Scheduler* supports the *Symmetric* and *Asymmetric* scheduling policies. The *Scheduler* also applies a **Request, Hold and Grant** (*RHG*) policy to process the memory requests. The *Scheduler* can apply *RHG* policy together with *Symmetric* and *Asymmetric* scheduling policies. This implies that, at run-time the *Scheduler* improves the QoS by reorganizing and prioritizing the memory request with respects to number of requests and transfer size. In this section, we further explain the PMC scheduling policies.

The PMC *Scheduler* manages and controls the run-time requests and programmed priorities of processing cores. Each processing core's *request* includes a read and write memory operation. At program-time, each processing core is assigned a priority value along with the *Local Address* (*Task ID*), which are placed in the *Program-Time Priority Descriptor* (shown in Figure 2.11 (b)). The processing cores are categorized into three states, *busy* (core is processing on local buffer), *requesting* (core is idle), and *request & busy*. In the *request & busy* state the core is assumed to have double or multi buffers. During this state, the core is processing the data of one buffer while making requests to fill another buffer.

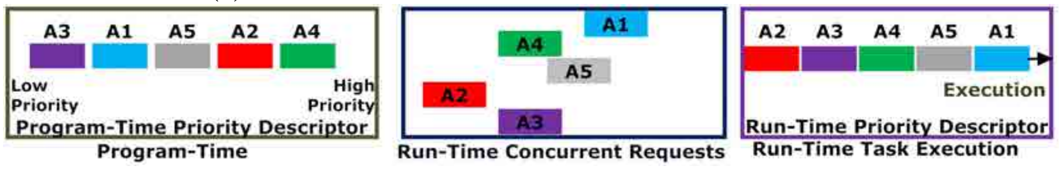
To provide multi-buffer support in the current developed platform, a *state controller* (Figure 2.11 (a)) is instantiated with each core that handles the states of the core using a double/multi buffering technique. The *state controller* manages the processing core's *request & grant* signals and communicates with the scheduler. The *state controller* can manages multiple buffers. For example, if there are two buffers (*Buff0* & *Buff1*)



(a)



(b)



(c)

Figure 2.11: PMC: (a) State Controller (b) Scheduler (c) Scheduling Example

attached with the state controller, it provides a link to transfer *Buff0* data and allows processing core to process *Buff1*. Once the core finishes processing *Buff1* the *state controller* swaps *Buff1* with *Buff0*. The *state controller* gives access of *Buff0* to the processing core and requests PMC to transfer *Buff1* data. Once the request is accepted, the PMC transfers the *state controller* buffer data to/from main memory.

PMC *Scheduler* supports three scheduling policies of the requests of the Multi core System. Two are static, i.e. provided by the programmer, symmetric and asymmetric and the third is an adaptive policy, Request, Hold and Grant (RHG).

The **Symmetric** and **Asymmetric** are programmed scheduling policies that are defined at compile-time and are executed by hardware at run-time. In **Symmetric** multi-core strategy, the PMC *Task Placer* manipulates the incoming requests in FIFO (First in First out) and places them in the *Dispatch Descriptor* (Figure 2.11 (b)). The **Asymmetric** strategy emphasizes on priority and incoming requests of the processing cores. Each core is assigned a fixed priority at program-time which is placed in the *Program-Time Priority Descriptor* (shown in Figure 2.11 (b)). At run-time, the *Scheduler* accumulates requests from the *Multi-core System*. The *Comparator* and *Task Placer* maintain them in the *Dispatch Descriptor*. The *Comparator* takes requests from multiple processing cores, compares them with programmed priorities and forwards results to the *Task Placer*. The *Task Placer* places the requests in the *Dispatch Descriptor* and executes requests only the are ready to run, and there are no higher priority cores that are in a ready state. The *Dispatch Descriptor* executes processing core requests one by one. For example, different requests (A1 to A5) are generated in concurrent order from multiple sources (shown in Figure 2.11 (c)). The *Scheduler* takes the first request (`A1`) as it is from *Dispatch Descriptor*, remaining requests are executed according to the priority level defined in *Program-Time Priority Descriptor*. The requests `A4` and `A3` are generated while `A5` is not finished. The PMC selects `A4` due to its highest priority level defined in *Program-Time Priority Descriptor*. Depending upon the scheduling policies and *Program-Time Priority Descriptor*, the PMC selects the next core from *Dispatch Descriptor*. If same priorities are assigned for more than one processing core, PMC scheduler executes them using the FIFO method.

The *Address Manager* of PMC has particular *Descriptor Memory* (register set) for each core. These descriptors are masked with interrupt and request signals. Once a request is generated, the *Address Manager* starts memory operation for the requested

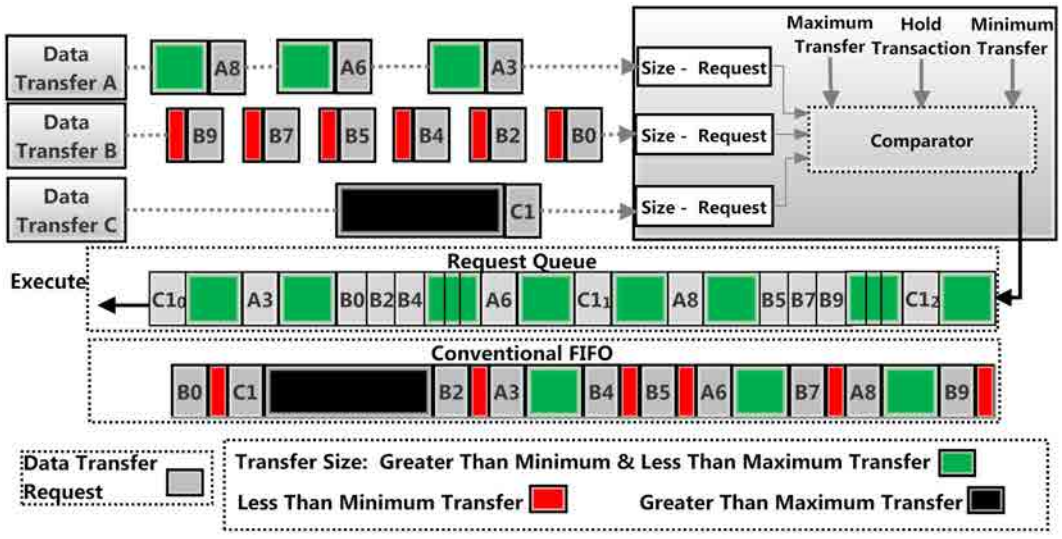


Figure 2.12: PMC: Request, Hold and Grant Policy

processing core using its descriptors. After completion of memory read/write operation, the PMC scheduler receives an interrupt (*ack*) signal from the memory manager unit. This signal informs the scheduler about the selection of the next core to process. The scheduler captures the *ack* signal from the memory manager and assigns the *grant* signal to the appropriate processing core.

The *Scheduler* applies an adaptive run-time managed (RHG) policy to process the memory requests, Figure 2.12 shows an example. The *RHG* depends upon the number of data access requests and transfer sizes. The *RHG* uses three parameters for scheduling which are the *Minimum Transfer*, the *Maximum Transfer* and the *Hold Transaction*. The *transfer* parameters specify the minimum and maximum size of a data transfer request on which the *Scheduler* applies the *RHG* policy. The *Scheduler* immediately grants all requests which are greater than *Minimum Transfer* and less than *Maximum Transfer*. It splits the transfers longer than *Maximum Transfer* and holds transfer requests of size less than *Minimum Transfer*, merging several of them into a single larger request. *Hold Transaction* is the number of requests for which the *Scheduler* holds a request. 128B, 4KB and 8 are the default values of *Minimum Transfer*, *Maximum Transfer*, and *Hold Transaction* respectively. These parameters are programmed at compile-time. If, at run-time, the *Scheduler* receives data transfer requests of 32B (smaller than *Minimum Transfer*) form the core-A, the *Scheduler* holds the requests and keep merging it with the next transfers. In this case the *Scheduler* requires four requests from core-A of 32B to process data transfer. In the same way, if core-A has data transfer size 16KB (larger than *Maximum Transfer*), the *Scheduler* splits the request into four transfer each having 4KB of data transfer size.

A conventional bus controller uses an arbiter that manages the data bus and executes a data access based on the application priority and request arrival time. The application's priorities are programmed at compile-time. The applications having low priority wait until higher priority applications finish and the applications having same priorities are executed as First In First Out (FIFO) which affects the fairness. At run-time, the applications having multiple small memory requests generate scheduling delay and stall the system. Whereas an application request with large data transfer size occupies the whole memory bandwidth and stalls other applications. Unlike the conventional bus controller, in our proposal the *PS* prioritizes access pattern requests at run-time, using the application memory access descriptors. The PMC *Scheduler* rearranges access pattern requests at run-time as shown in Figure 2.12. In the example, *A, B* and *C* are different applications and *B0, C1* to *B9* are their memory requests. The numbers represent the order of generation of the requests. The application *A* has a data transfer *size* greater than the *Minimum Transfer* size, so the *Scheduler* grants the request by placing it in the *request queue*. The application *B* has multiple memory requests of data transfer *size* less than the *Minimum Transfer*, so the *Scheduler* holds the data transfers B0, B2 and B4 merging their memory requests. When the *total transfer size* of the accumulated request of the application exceeds the *Minimum Transfer* size, the *Scheduler* places the merged access request in the *request queue*. For the application C with a request that has a data transfer *size* larger than *Maximum Transfer*, the *Scheduler* splits the transfer into multiple transfer requests ($C1_0$, $C1_1$ and $C1_2$) of size *Maximum Transfer*. If there are requests in the *request queue*, then after transferring data for the $C1_0$ request, the *Scheduler* holds the $C1_1$ and $C1_2$ requests and grants the bus to the waiting requests. Once the waiting requests (A3, B0, B2, B4 and A6) of the *request queue* finish, the *Scheduler* grants the holding transfer ($C1_1$). In order to avoid starvation, after a number of *Hold Transaction* of requests available in the *request queue*, the *Scheduler* steals a transfer and executes the application with data transfer size less than *Minimum Transfer*. Once descriptors are in the *request queue*, a FIFO policy is used.

2.1.4 Main Memory System

The PMC *Main Memory System* uses *Pattern Aware Main Memory Controller* (PAMMC) (shown in Figure 2.13) to transfer data to/from the *Main Memory*. The *PAMMC* uses descriptors for access patterns which improve the memory bandwidth by transferring descriptors to the memory controllers, rather than individual references and by accessing data from the *Main Memory*. Unlike a conventional main memory controller, the *PAMMC* uses descriptors to access data. At run-time *PAMMC* takes descriptors for an access pattern from Memory Manager, decodes it and generates bank, row and column addresses. If the access pattern has unit stride and requires data from a single bank, then the *PAMMC* opens the row buffer of the appropriate bank, transfers data in burst mode and keeps the same bank open and keeps pre-charged contiguous rows.

The *PAMMC*, takes memory access descriptions from the *Memory Manager*. It is responsible for transferring data to/from the *Main Memory*. The *PAMMC* deals with

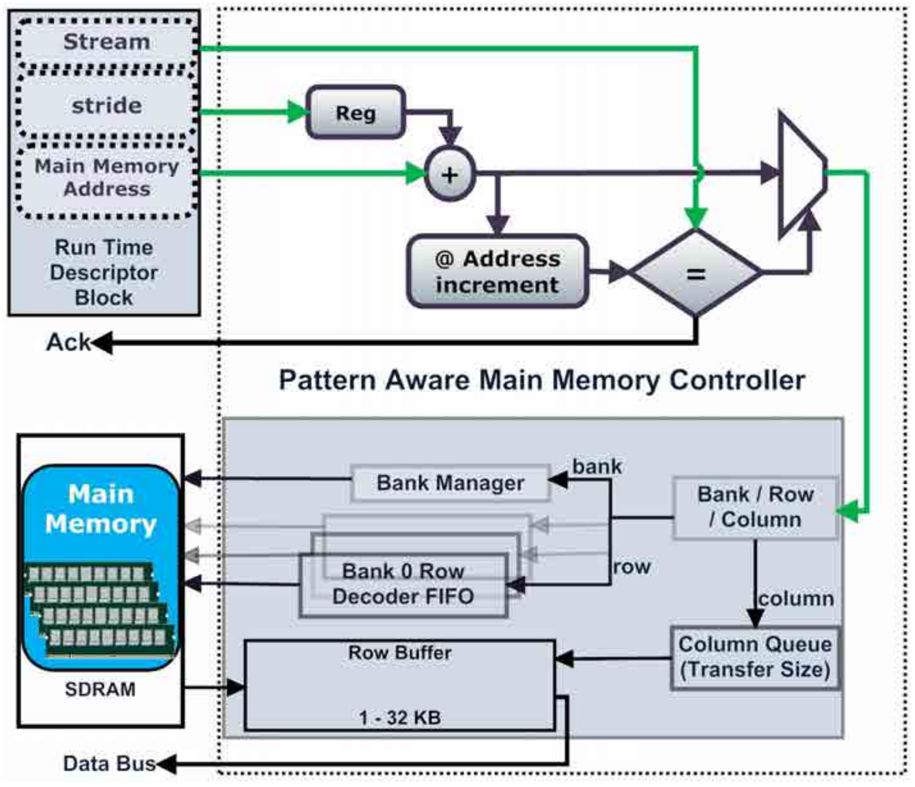


Figure 2.13: PMC: Pattern Aware Main Memory Controller

the *Address*, *Stream* and *Stride* registers of pattern descriptors. It takes a pattern descriptor and translates it into main memory addresses. Each address is split into *Bank*, *Row* and *Column* of SDRAM addresses. Strides between two consecutive accesses are handled by the *PAMMC* without generating delay. For each data transfer, the first data transfer uses an address taken by the descriptor unit and for the rest of the transfers, the address is equal to the address of the previous transfer plus the size of the stride. The *PAMMC* uses separate data lines labeled as *Data In* and *Data Out*. It enables the *data pattern* to be written at the appropriate location of the main memory by generating the write–enable along with the write–data and mask–data control signals. The source of the *data stream* can be a *Source Synchronous Streaming Interface*, a *Processor Bus Interface* or a contiguous *Specialized Scratchpad Memory*. The *PAMMC* supports 64-bit data bus and 8-bits mask bus. For every 8 bits of data bus, there is a mask bit.

The *PAMMC* supports two possible modes of operation for bank management: the single-bank mode and the multi-bank mode. In the single bank mode, the controller keeps one bank and row combination open at any given time. In the multi-bank mode, the controller keeps multiple banks open at any given time. This mode is used when the data access patterns of an application require data from different banks at the same time. The *PAMMC Bank Manager* is integrated with the design to reduce the memory access time and power by managing either the single- or the multi-bank mode according to the memory access pattern descriptions. The multi-bank mode is used for complex data patterns having long strides that access data from multiple banks in parallel. At run-time *PAMMC* takes descriptors for an access pattern from the *Memory Manager*, decodes them and generates bank, row and column addresses. If the access pattern has unit stride and requires data from a single bank then the *PAMMC* opens the row buffer of the appropriate bank, transfers data in burst and keeps the same bank open and if required keeps precharged contiguous rows. This type of access patterns is very common in the MAPC system because access patterns are organized in descriptors and occurs for around 70% of all memory patterns. Depending upon the SDRAM banks, the *PAMMC* processes in parallel multiple patterns, each accessing a single bank. For example, in our current evaluation environment SDRAM has 4 banks, therefore, *PAMMC* processes up to four access patterns in parallel. Depending upon the type of access pattern, the *PAMMC* decides when its associated bank and row should be precharged. If the access pattern has a stride larger than bank length then the *PAMMC* precharges multiple *banks* and *row buffers* for the requesting access.

2.2 Programming Model

When using the PMC model, the programmer does not need to worry about the hardware related programming and configuration constraints. By using PMC, the memory load/store operations are shaped into patterns and are scheduled in parallel with compute operations. The PMC supports complex irregular, strided 1D, 2D, 3D and automated blocking for data access operations to transfer data between *Local* and *Main memories*. The memory access operations optimize complex/irregular memory accesses, provide memory bandwidth performance by accessing only useful data at a fine granularity, and offload memory access overhead by supporting address calcula-

Table 2.1: C/C++ Device Drivers to Program/Operate PMC System

API Function	Description

`PMC_SEND (Local Address,`
`Main Address,`
`Size, Stride,`
`Priority)`
	Vectorizing Regular Data Access
	Stride defines type of Vector Data Access
	Based on Stride value the PMC transfers
	a row, column or diagonal vectors

`SEND_TILE, RECEIVE_TILE,`
`PMC_MEMCPY (`
`SCRATCHPAD_MEMORY,`
`DATA_SET, Priority)`
	Block/Tiled Data Access
	`SCRATCHPAD_MEMORY`
	indicates *Local Memory* buffer
	`DATA_SET` indicates *Main Memory* data set

`3D_STENCIL, 3D_STENCIL_VECTOR(`
`SPECIALIZED_MEMORY,`
`DATA_SET, Priority)`
	Specialized Data Transfer
	`SPECIALIZED_MEMORY` indicates SSM structure
	`DATA_SET` indicates *Main Memory* data set

`SEND_POINTER, SEND_TREE`
`( Local Address,`
`Main Address,`
`Size, Stride, Priority,`
`Offset Left, Offset Right)`
	Irregular Data Access e.g. Pointer, Linked List
	Offset register for 1D pointer
	`Offset Left & Offset Right` registers
	point the address of next left and right
	nodes respectively of tree data structure

tion, data shuffling, and format conversion. The *regular descriptor* is used to transfer complex run-time *known* access patterns. The *Unknown Independent and Dependent* access patterns are supported by using a *irregular descriptor* that describes unpredictable patterns and leverage hardware communication protocol between the multi-cores and memory system to gather the information not known until run-time. The *Multi-core System* communicate with PMC through a group of commands, controls, status and data registers and signals.

2.2.1 Descriptors and Data Structures

PMC supports regular data access patterns such as strided vector and 1D/2D/3D auto tiling. A single descriptor can hold the information of a complex access pattern, e.g. Row, Column, Diagonal vector, etc. For example, a FIR and FFT application kernels have streaming and 1D tiled access pattern respectively with unit stride. To access data patterns for the FIR and FFT application kernels, the PMC uses `PMC_SEND()` (shown in Table 2.1) function call that utilizes a single descriptor. 3D stencil memory requires three descriptors for accessing row, column and bank vectors, each descriptor with a stride of one, *row_size* and *row_size* × *column_size* length respectively.

Irregular data structures (pointer, linked-list, tree, etc.) contain a hierarchical relationship between elements. The overall structure of a *Linked List* is described by using *Offset* registers to connect all its elements together such as the links in the chain. If the data structure is identified at program-time the PMC *Linked List* data structure allocates space for each data element separately using the *Irregular Descriptor Memory* through a *linked list descriptor*. The *Offset* register is set to *Known* or *Unknown* accordingly. A *Tree* data structure *Descriptor Memory* is similar to the *linked list Descriptor Memory* except that instead of a single *Offset* register per descriptor there can be multiple *Offset* registers.

The PMC uses commands, controls and status registers to synchronize the *Multi-core System* with PMC. The proposed system provides comprehensive support for the C and C++ languages. Figure 2.14 shows the C structure that is used to describe the PMC *SSM* and main memory data set. The structure is used to define the *Local Memory SMM* and the *Main Memory* data set of an application kernel. The address values indicates starting address of *SSP* buffer and *Main Memory* data set. Parameters

```
//Specialized Scratchpad
typedef struct
SPECIALIZED_SCRATCHPAD
{ int ADDRESS;
int WIDTH;
int HEIGHT;
int PLANE;
} PMC_SCRATCHPAD
// Main Memory (SDRAM)
typedef struct
PMC_MAIN_MEMORY
{ int ADDRESS;
int WIDTH;
int HEIGHT;
int BANK;
} MAIN_MEMORY
// Main Program
PMC_SCRATCHPAD SSM_1D;
PMC_SCRATCHPAD SSM_2D;
PMC_SCRATCHPAD SSM_3D;
MAIN_MEMORY DATASET_1D;
MAIN_MEMORY DATASET_2D;
MAIN_MEMORY DATASET_3D;
// Part I : 1D Memory
// 1024 Scratchpad Buffer
SSM_1D.ADDRESS=0X00000000;
SSM_1D.WIDTH=1024;
SSM_1D.HEIGHT=1;
SSM_1D.BANK=1;
```

```
// 4096 Data Set
DATASET_1D.ADDRESS=0X10000000;
DATASET_1D.WIDTH=4096;
DATASET_1D.HEIGHT=1;
DATASET_1D.BANK=1;

// Part II : 2D Memory
// 1024x1024 Buffer
SSM_2D.ADDRESS=0X00000400;
SSM_2D.WIDTH=128;
SSM_2D.HEIGHT=128;
SSM_2D.BANK=1;
// 4096x4096 Data Set
DATASET_2D.ADDRESS=0X10001000;
DATASET_2D.WIDTH=1024;
DATASET_2D.HEIGHT=1024;
DATASET_2D.BANK=1;

// Part III : 3D Memory
// 32x32x32 SSM
SSM_3D.ADDRESS=0X00010000;
SSM_3D.WIDTH=32;
SSM_3D.HEIGHT=32;
SSM_3D.BANK=32;
// 4096x4096 Data Set
DATASET_3D.ADDRESS=0X10100000;
DATASET_3D.WIDTH=128;
DATASET_3D.HEIGHT=128;
DATASET_3D.BANK = 128;
```

Figure 2.14: Examples of Specialized Scratchpad Memory and Main Memory Data Structure

Width, Height of *SSP* and *Main Memory* define structure of 1D or 2D buffer (shown in Figure 2.14. The parameters *Plane* and *Bank* of *SSP* and *Main Memory* respectively define 3D specialized memory structure.

2.2.2 Working Examples

Different applications exhibit different data access patterns, forcing the hardware designers to facilitate software programming by keeping a generic interface between the memory management unit and the compute units. This results in a compromise on the achievable performance because of the generic way of describing data transactions. An alternative to achieve higher performance is to avoid the generic interface and manually arrange memory accesses. PMC programming model aims to remove the programmer effort of manually arranging memory accesses but meet the performance requirements of HPC applications. PMC provides a wrapper library that offers function calls to describe the application complex access patterns. The programmer

```
/*Instance of the DMA driver*/
static XDmaCentral DmaCentral;
/* Pointer to local buffer */
int  *SrcBuffer;
SrcBuffer = (int *)  0
   x00000000;
/* Pointer to Dataset  */
int  *DestBuffer;
DestBuffer = (int *)  0
   x000010000;
/* Transfer Size in Bytes */
int   Bytesize=1024;
/* Single Transfer */
XDma_Central_Transfer(
   DmaInstance, SrcBuffer,
   DestBuffer, Bytesize);
```

(a)

```
/*Local Memory and Main Memory
    */
PMC_SCRATCHPAD SSM;
MAIN_MEMORY DATASET;
/* Pointer to SSM */
SSM.ADDRESS=0X11100000;
/* Pointer to Dataset  */
DATASET.ADDRESS=0X10100000;
SSM.SIZE = 1024;
int STRIDE = 4;
int   OFFSET=0x20000000;
int PRIORITY = 1;
/* PMC DMA Transfer */
PMC_SEND(SSM.ADDRESS , DATASET
   .ADDRESS, SSM.SIZE, STRIDE,
    OFFSET, PRIORITY);
```

(b)

Figure 2.15: DMA Transfer Function Calls: (a) Conventional DMA (b) PMC

only needs to identify the appropriate function call to the library and the PMC system automatically transfers the data pattern to the local memory of the compute unit. The function calls improve the performance of application kernels by marshaling data according to the application needs. For the complex memory patterns, a single or multiple descriptors are used at compile-time to reshape and unfold data patterns. At run-time, these descriptors help PMC to transfer memory patterns between the processing cores and the main memory with minimum delay. PMC decouples data accesses from execution, prefetches complex data structures and prevents FPGA stalls. The proposed PMC system provides C and C++ language support. The system can be managed by C based device drivers. Figure 2.15 shows transfer examples in systems with a conventional DMA-based memory controller, and PMC. The conventional DMA system transfers data between main memory and the local memory of the processing core using DMA. DMA transfers are specified using the `XDma_Central_Transfer` function call. Figure 2.15 (a) shows how it is used to transfer `1024` bytes of data stream from local buffer *SrcBuffer* to main memory *DestBuffer*. To transfer a noncontiguous block of data, such a strided memory access pattern, the conventional DMA system uses multiple `XDma_Central_Transfer` calls. In PMC, the `PMC_SEND` function call is used to specify descriptors that will be placed in the *Descriptor Memory*. Figure 2.15 (b) shows an example where `send` is used to set up a descriptor to access `1024` bytes of data with *stride* of 4 bytes between two consecutive elements. Since element size is `4` bytes, PMC will access `1024` consecutive bytes. The *SSM.ADDRESS* and *DATASET.ADDRESS* parameters hold the start address of local (data) memory and main memory data set respectively. The parameters *SIZE* and *STRIDE* define the type of memory access. The PMC `PMC_SEND` function call has the *OFFSET* parameter to provide information of the following transfer at run-time without affecting the computation process. In the example of Figure 2.15 (b), *OFFSET* indicates that the next transfer is unknown until run-time. For more complex memory accesses, the PMC provides the `SEND_IRREGULAR` function call that can have multiple *OFFSET* registers. In both systems (Conventional DMA and PMC) the programmer has to describe the main memory data sets and the local buffer size of application kernels.

A conventional example of 3D stencil access pattern is shown in Figure 2.16. A 3D stencil (n=4 see Section2.1.2.3) memory access has three (row, column and plane)

```
// Stencil Structure
#define Sten_size 4
// 128x128x128 Main Memory Data Set
#define WIDTH 128
#define HEIGHT 128
#define BANK 128
main () {
int X,Y,Z;
X = WIDTH;
Y = WIDTH*HEIGHT;
Z =0;
float Sten[WIDTH*HEIGHT*BANK];
for ( k = Stencil_size ; k < BANK - Sten_size ; k++ )
for ( j = Stencil_size ; j < HEIGHT - Sten_size ; j++ )
for ( i = Stencil_size ; i < WIDTH - Sten_size ; i++ )
{Z = k*(WIDTH*HEIGHT) + (j*WIDTH) + i;
Sten[i+j*X+(k-1)*Y] + Sten[i+j*X+(k+1)*Y] +
Sten[i+j*X+(k-2)*Y] + Sten[i+j*X+(k+2)*Y] +
Sten[i+j*X+(k-3)*Y] + Sten[i+j*X+(k+3)*Y] +
Sten[i+j*X+(k-4)*Y] + Sten[i+j*X+(k+4)*Y] +
Sten[Z] +
Sten[i+(j-4)*X+k*Y] + Sten[i+(j+4)*X+k*Y] +
Sten[i+(j-3)*X+k*Y] + Sten[i+(j+3)*X+k*Y] +
Sten[i+(j-2)*X+k*Y] + Sten[i+(j+2)*X+k*Y] +
Sten[i+(j-1)*X+k*Y] + Sten[i+(j+1)*X+k*Y] +
Sten[(i-4)+j*X+k*Y] + Sten[(i+4)+j*X+k*Y] +
Sten[(i-3)+j*X+k*Y] + Sten[(i+3)+j*X+k*Y] +
Sten[(i-2)+j*X+k*Y] + Sten[(i+2)+j*X+k*Y] +
Sten[(i-1)+j*X+k*Y] + Sten[(i+1)+j*X+k*Y];}
}
```

Figure 2.16: Conventional Load/Store Stencil Access Pattern

```
#define stencil_size 4
#define PRIORITY1 1
#define PRIORITY2 2

// Main Program
PMC_SCRATCHPAD STENCIL;
PMC_SCRATCHPAD SSM_3D;
MAIN_MEMORY DATASET_3D;
// Part I : Local SSM
// Single Stencil Buffer
STENCIL.ADDRESS=0X10000000;
STENCIL.WIDTH=9;
STENCIL.HEIGHT=3;
STENCIL.BANK=1;
// 3D 32x32x32 SSM
SSM_3D.ADDRESS=0X11000000;
SSM_3D.WIDTH=32;
SSM_3D.HEIGHT=32;
SSM_3D.BANK=32;
// Part II : Main Memory
// 3D-Data set
DATASET_3D.ADDRESS=0X00100000;
DATASET_3D.WIDTH=128;
DATASET_3D.HEIGHT=128;
DATASET_3D.BANK=128;
//PART III : DATA TRANSFER
3D_STENCIL (STENCIL, DATASET_3D, PRIORITY1);
3D_STENCIL_VECTOR (SSM_3D, DATASET_3D, PRIORITY2);
```

Figure 2.17: PMC 3D Stencil Access Pattern

vectors. PMC removes the programmer effort of manually arranging memory accesses by using descriptors that reduce address operations and perform address management in hardware. A PMC example of a 3D stencil access is shown in Figure 2.17.

```
// Main Program
PMC_SCRATCHPAD SSM_2D;
PMC_SCRATCHPAD SSM_3D;
MAIN_MEMORY DATASET_2D;
MAIN_MEMORY DATASET_3D;
#define PRIORITY1 1
#define PRIORITY2 2
// Part I : Local SSM
// 2D 32x32 Buffer
SSM_2D.ADDRESS=0X00000000;
SSM_2D.WIDTH=32;
SSM_2D.HEIGHT=32;
SSM_2D.BANK=1;
// 3D 32x32x32 SSM
SSM_3D.ADDRESS=0X10000000;
SSM_3D.WIDTH=32;
SSM_3D.HEIGHT=32;
SSM_3D.BANK=32;
// Part II : Main Memory
```

```
// 2D-Data set
DATASET_2D.ADDRESS=0X00001000;
DATASET_2D.WIDTH=256;
DATASET_2D.HEIGHT=256;
DATASET_2D.BANK=1;
// 3D-Data set
DATASET_3D.ADDRESS=0X00100000;
DATASET_3D.WIDTH=128;
DATASET_3D.HEIGHT=128;
DATASET_3D.BANK=128;
//PART III : DATA TRANSFER
PMC_MEMCPY (SSM_2D,
DATASET_2D, PRIORITY1);
3D_STENCIL_VECTOR (SSM_3D,
DATASET_3D, PRIORITY2);
// PART IV:
//PROCESSING MULTI-CORES
LAPLACIAN (SSM_2D);
STENCIL (SSM_3D);
```

Figure 2.18: PMC Programming Example of 2D and 3D Applications

A single stencil can be accessed by using the `3D_STENCIL()` function call. The `3D_STENCIL()` function requires *Local Memory* and *Main Memory* data set information. The PMC uses three descriptors that transfer the three stencil vectors: row, column and plane. PMC reduces address management by initializing different strides (see Section 2.2.4) for each vector access. PMC improves performance and reduces main loop computation by access stencil in the form of vectors. A `3D_STENCIL_VECTOR()` call is used to transfer multiple stencil vectors by using multiple `3D_STENCIL()` function calls. This function call requires *Specialized Scratchpad Memory* and *Main Memory* data set information.

A multi-core programming example of PMC is shown in Figure 2.18. The multi-cores are initialized with the data structures and the scheduling policy priorities. Like

Pthread scheduling policies the PMC, scheduler initializes priorities at compile time. Unlike Pthread, the PMC executes requests at run-time in hardware. If the same priority is assigned to multiple cores, the PMC scheduler processes the cores in symmetric mode. The PMC automatically adjusts the data set into appropriate *SSP* sizes, to be processed by the processing core. The minimum requirement of each access pattern is that the programmer has to describe the main memory data sets and the size of *SSM* for the application kernel. An example used to program the PMC based multi-core system is shown in Figure 2.18. The program initializes two *ASHA*s and their 2D and 3D tiled data pattern. *Part I* of Figure 2.18 defines a 2D and 3D *SSM* (`SCRATCHPAD`) of 32x32 and 32x32x32 elements respectively. *Part II* identifies main memory data sets information. *Part III* initializes 2D and 3D data transfers. The PMC uses `PMC_MEMCPY` function call to read/write dense data structure and access pattern. The minimum requirement of each access pattern using `PMC_MEMCPY` function call is that the programmer has to describe main memory data sets, specialized memory buffer size and priority of data transfer *Part IV* initializes Laplacian and 3D-Stencil processing cores. The Laplacian kernel requires 2D block of data to process and 3D-Stencil complex 3D access pattern.

2.2.3 Access Patterns

The access patterns in PMC are managed with functions call shown in Table 2.1. Specialized function calls are provided for complex patterns such as 2D/3D tile, binary tree, etc. In order to elaborate the functionality of PMC two types of data access patterns are discussed in this section: Vector Access Pattern and Tiling Access Pattern.

2.2.3.1 Vector Access Pattern

A vector access has 1D data access pattern. The PMC system allows to access vector patterns, using `PMC_SEND()` function call. Each call is associated to a single descriptor block. SoA data access requires unit-stride, whereas the AoS requires strided access. The stride is determined by the size of the working set. Figure 2.19 presents three different vector accesses (`x[n][0]`, `y[0][n]` and `z[n][n]`) for a `n×n` matrix where *vector x* corresponds to the contiguous row having *unit stride*, *vector y*

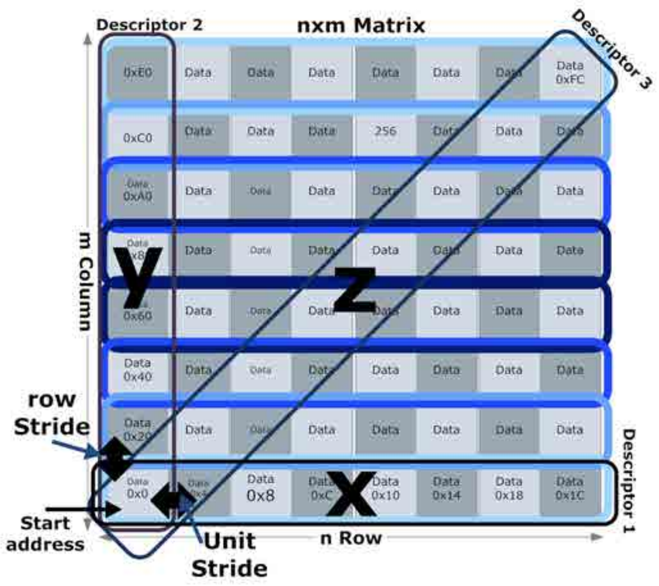


Figure 2.19: PMC Vector Access Patterns

belongs to column access with stride equal to *row length* and *vector z* is a diagonal vector pattern. Stride of `z[n][n]` row length+1.

For example, a 3D stencil access requires three descriptors. Each descriptor accesses a separate (*x, y* and *z*) vector in a different dimension, as shown in Figure 2.20. By combining these descriptors, the PMC exchanges 3D data between the main memory and the *SSM* (Figure 2.21). The value NX, NY and NZ define the width (row_size), height (column_size) and length (plane_size) respectively of the 3D *SSM*. The 3D-Stencil has x, z and y vectors having direction of row, column and plane respectively. The vector *x* has unit stride, the vector *z* has stride equal to *row_size* and the vector *y*

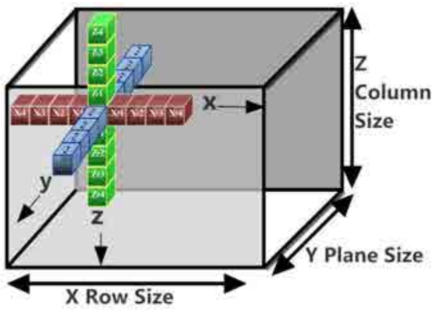


Figure 2.20: 3D Stencil Vector Access

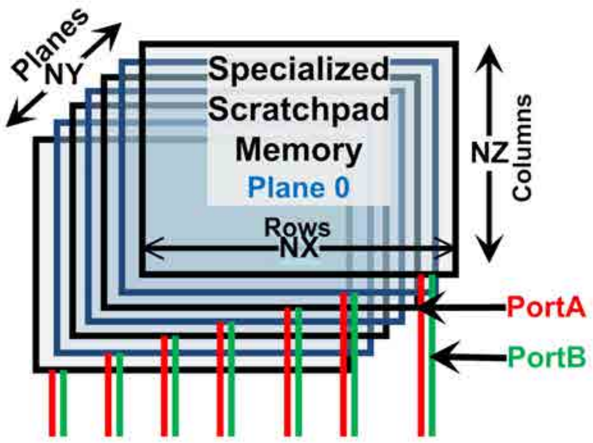


Figure 2.21: 3D Specialized Memory Structure

has stride equal to the size of one plane, i.e. $row\_size \times column\_size$.

2.2.3.2 Tiling Access

The tiling access transfer the data in the form of block/tile. Each block of data can have multiple non-contiguous vector accesses. The tiling technique is employed to improve system performance. It is widely used for exploiting data locality and improving parallelism.

Single Tile Access: The data accesses inside a tile can be arranged into one or multiple descriptors. By combining these descriptors, the PMC exchanges tiled data

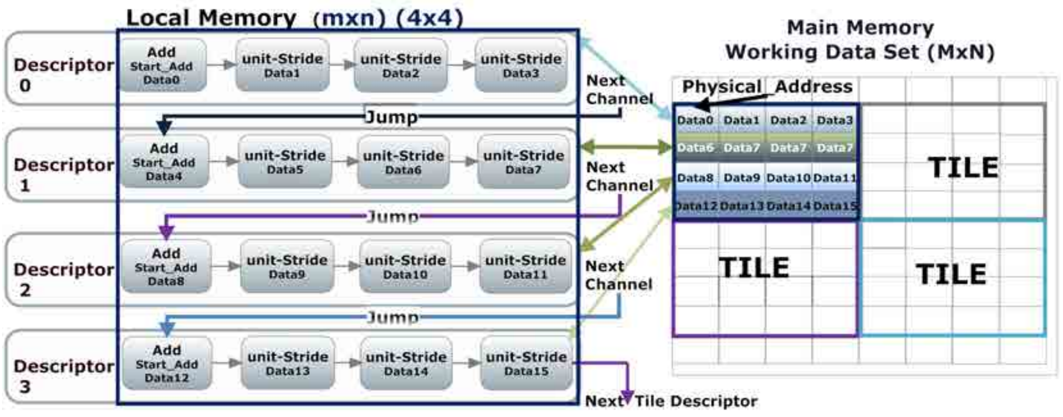


Figure 2.22: PMC Tiling Example

between the *Main Memory* and *SSM*. To read/write a single tile, the `send_tile()` and `receive_tile()` *calls* are used to initialize the *Descriptor Memory* blocks. Each *call* requires two input parameters *Local_Address* and *Main_Address*. Parameter *Local_Address* indicates the starting address of the scratchpad memory where the tile is read/written. *Main_Address* holds the start address of the working data set.

Multi-Tiling Access: The PMC multi-tiling method attempts to derive an effective tiling scheme. It requires information about the *Main Memory Tiled Data set* (M^N) and the *Buffer's Tile size* (m^n) of the computing engine. Where *(M,m)* represents width and *(N,n)* represents dimension of each tile. Depending on the structure of the buffer memory, the PMC partitions the *Main Memory* into multiple tiles. The *number of tiles* for a multidimensional memory access is given by Equation 2.1, assuming an even number of dimensions.

$$\textit{Number of Tiles} = \left\lceil \frac{M^N}{m^n} \right\rceil = \left\lceil \frac{\textit{DataSet Width}^{\text{Dimension}}}{\textit{LocalBuffer Width}^{\text{Dimension}}} \right\rceil \tag{2.1}$$

An example of PMC (4×4) 2D-tile access is shown in Figure 2.22. The PMC uses 4 descriptors to access a single tile. `Descriptor 0` takes *descriptor 0* from *Descriptor Memory* and accesses data starting at location *Physical_Address* with unit-stride and n-stream (row width) size. Depending on the size of the main memory data set (M^N), different strides (address) are used between transfers. The *starting address* of the next transfer is dependent on Equation 2.2. `Descriptor 1` starts right after the completion of `Descriptor 0`. `Descriptor 3` is the last transfer. After completion of `Descriptor 3` transfer, PMC generates an interrupt signal indicating to the external source that the tile data transfer has finished. The PMC will move ahead to the next tile and restart processing.

$$\textit{Start Address} = \textit{Dataset\_Base\_Address}_{Tile} + \textit{Buffer\_Width}_{Tile} * \textit{Descriptor number} \tag{2.2}$$

2.2.4 Programming PMC

At compile time PMC uses wrapper library that contains information of the application's complex access patterns using function calls, e.g. PMC_MEMCPY, 3D_STENCIL, etc. The programmer only needs to identify the appropriate function call to the library and the PMC automatically transfers data pattern to local memory of compute unit. A PMC API is used to parse the annotated code and then program the PMC *Descriptor Memory*. The PMC API takes parameters from the code and automatically rearranges and parallelizes access patterns in software and program the PAC *Descriptor Memory*. PMC API rearranges and parallelizes access pattern based on available and required size of local and main memories. For example, the Laplacian and Stencil applications have 32x32 and 32x32x32 *Local Scratchpad Memory* and 256x256 and 128x128x128 data set size of *Main Memory* respectively. The PMC API converts the *Main Memory* data set into tiles based on *Local Scratchpad Memory* size. For each 2D and 3D access pattern 64 tiles are generated. The PMC API organizes it in *Descriptor Memory* by allocating different different *Main Memory Address* parameters (shown in Figure 2.23). Later PMC software and or hardware mechanism optimizes *Descriptor Memory* and uses only the *Offset* address to access multiple tiles, which reduces the *Descriptor Memory* size. For the complex memory patterns having load/store access, the programmer parallelizes the code by hand using functions such as `PMC_POINTER, or PMC_TREE`. The reasons for requiring hand parallelization are: the access patterns are complex and have address dependencies. Equivalent process is applied to parallelize the access pattern of generic (non PMC) application kernels. While using this PMC model, the programmer does not need to worry about the hardware related programming and configuration constraints. By using PMC, the memory load/store operations are shaped into patterns and are scheduled in parallel with compute operations. Moreover, the optimal scheduling of operations does not depend upon memory latencies and therefore does not effect the scheduling of computations. Compared to the work presented here, the alternative is to do everything manually in order to accelerate the program on FPGA using an accelerator; in this case the programmer needs similar annotated code but having much complex arrangement and explicit scheduling of DMA function calls. PMC avoids all this complexity. The PMC program flow is shown in

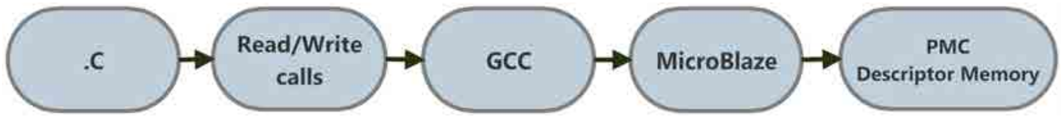
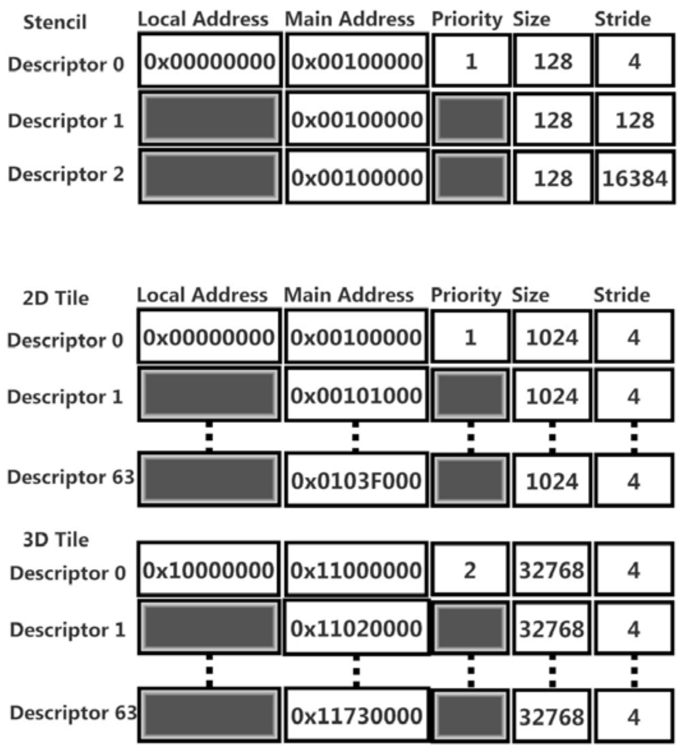


Figure 2.23: PMC Program Flow



Stencil	Local Address	Main Address	Priority	Size	Stride
Descriptor 0	0x00000000	0x00100000	1	128	4
Descriptor 1	▨	0x00100000	▨	128	128
Descriptor 2	▨	0x00100000	▨	128	16384

2D Tile	Local Address	Main Address	Priority	Size	Stride
Descriptor 0	0x00000000	0x00100000	1	1024	4
Descriptor 1	▨	0x00101000	▨	1024	4
⋮	⋮	⋮	⋮	⋮	⋮
Descriptor 63	▨	0x0103F000	▨	1024	4

3D Tile					
Descriptor 0	0x10000000	0x11000000	2	32768	4
Descriptor 1	▨	0x11020000	▨	32768	4
⋮	⋮	⋮	⋮	⋮	⋮
Descriptor 63	▨	0x11730000	▨	32768	4

Figure 2.24: PMC Descriptor Memory

Figure 2.24. The PMC program code is compiled using the GCC compiler. The memory access information is included in the PMC header file and provides function calls (e.g. PMC_MEMCPY(), and PMC_STENCIL(), etc.) that require basic information of the local memory and the data set. The programmer has to annotate the code using PMC function calls. The API takes the code (shown in Figure 2.18), pipelines and overlaps the PMC operations and then programs the *Descriptor Memory*.

Part II

Uni-core Memory System for Regular Data Pattern

3

PPMC: A Programmable Pattern Based Memory Controller

The current trend in the design of HPC systems is shifting towards the integration of microprocessors with accelerators [47] in order to achieve the increasing performance targets. However, high-performance microprocessor/accelerator systems are of little use if the memory hierarchy is unable to provide the necessary data bandwidth. The efficient management of memory accesses by timely prefetching across the set of accelerators and microprocessors in such a scenario are critical for performance. However, it is also very challenging. This chapter deals with the PMC based single core system and regular access patterns. The system supports one core simultaneously either *SSP* and *ASHA*. The main contribution of this chapter is the description of the hardware implementation of a Pattern-based Programmable Memory Controller (PPMC) which takes a data access description (descriptor blocks) and provides 1D, 2D and 3D data in streaming mode. This chapter discusses the evaluation architecture of PPMC and Baseline memory controller and their implementation on a Xilinx Virtex 5 FPGA (ML505 development board), including its integration with a Xilinx MicroBlaze processor. We also show how PPMC can be attached to *ASHA*. We evaluate application kernels on a MicroBlaze-based SoC Architecture using PPMC. This shows that the PPMC system is equally usable as a stand alone component within a reconfigurable system, and also as a slave in a system with a general purpose processor. The results are compared with

[1] Chapter 3 is based on the publications : [34] and [35].

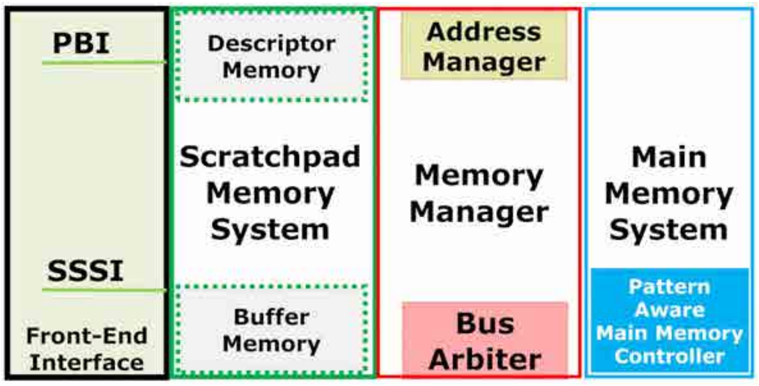


Figure 3.1: PPMC: Programmable Pattern based Memory Controller

a system that lacks the pattern based controller, showing great benefits from the usage of PPMC.

3.1 System Architecture

To demonstrate the operation of PPMC, we first depict its internal architecture in Figure 3.1, which also briefly shows the interconnection with the external processing units. Similar to PMC the PPMC system is comprised of four units with minimal features, which are the *Font-End Interface* (Section 2.1.1), the *Local Memory System* (Section 2.1.2) the *Memory Manager* (Section 2.1.3), and the *Main Memory System* (Section 2.1.4). The *Front-End interface* includes two separate interfaces, the *PBI* and the *SSSI*, to link with the *SSP* and the *ASHA* respectively. The *Local Memory System* consists of the *Descriptor Memory* along with the *Scratchpad Buffer Memory*. The PPMC *Memory Manager* uses an *Address Manager* to manage *Regular Patterns* and a *Bus Arbiter* instead of *Scheduler* to control and arbitrate data transfer requests.

3.2 Evaluation Architecture

In this section, we describe and evaluate the PPMC-based SoC architecture for the FIR, Thresholding, FFT, Laplacian solver, Matrix Multiplication and 3D-Stencils application kernels. In order to evaluate the performance of the PPMC System, the results are compared with a similar system that does not feature a PPMC unit.

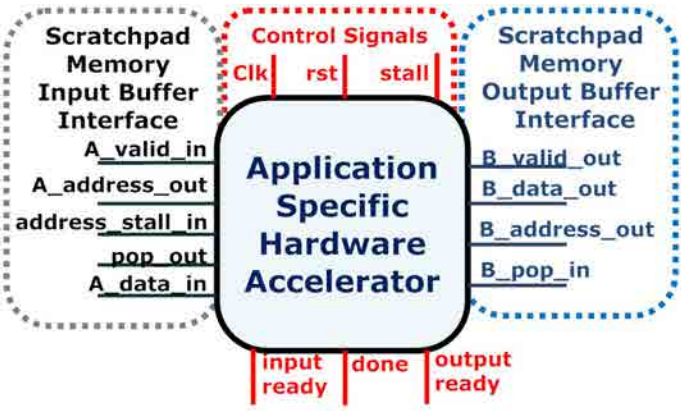


Figure 3.2: ROCCC-based Hardware Accelerator

In our evaluations, Matrix Multiplication and Thresholding applications are tested on the MicroBlaze *SSP* while the other kernels are tested on an *ASHA*. Both systems are configured with and without support of PPMC.

3.2.1 MicroBlaze SSP

MicroBlaze [20] is a 3 stage pipelined RISC soft-core. The processor is implemented with a Harvard memory architecture where instruction and data accesses have separate 32-bit address spaces. The data access to input/output memory is memory mapped. The processor has two bus interfaces for memory accesses: the Local Memory Bus (LMB) and the Processor Local Bus (PLB). Local memory is used for data and program storage and is implemented by using Block RAM. The local memory is connected to MicroBlaze through the LMB, and the BRAM Interface Controllers. MicroBlaze instruction prefetcher improves the system performance by using the instruction prefetch buffer and instruction cache streams. The system consumes 7612 flip-flops, 4988 LUTs and 14 BRAMs on a V5-Lx110T device.

3.2.2 ROCCC ASHA

The *ASHA* (shown in Figure 3.2) is generated by the ROCCC [23] compiler for each of the evaluated application kernels. These dedicated accelerators have the low power consumption and provide high performance. To balance the workload, each accelerator is equipped with two read/write data memories (double-buffers). The *ASHA* has two interfaces that read and write data to/from read buffer and write buffer respectively. The control signals manage the data movement and carry information of buffer memories data transfers. In this evaluation environment, *ASHA*s are executed at 100 Mhz of clock frequency and the highest bandwidth required is 400 MB for *c/a=1*. To manage on-chip data for computationally intensive window operations, ROCCC based *ASHA* can generate smart buffers. These smart buffers can reuse data through loop transformation-based program restructuring. The impact of this transformation on the size of generated hardware is high, i.e. generated hardware consumes approximately three times more FPGA logic. Therefore in our current evaluation, such transformation is not used. The PPMC itself performs off-chip and on-chip data management and supports complex memory accesses. The current evaluation supports complex memory accesses and can provide 3D data to the processing core with a single cycle of latency. The access patterns can be rearranged by re-programming the *Descriptor Memory* which does not require re-synthesis of PPMC.

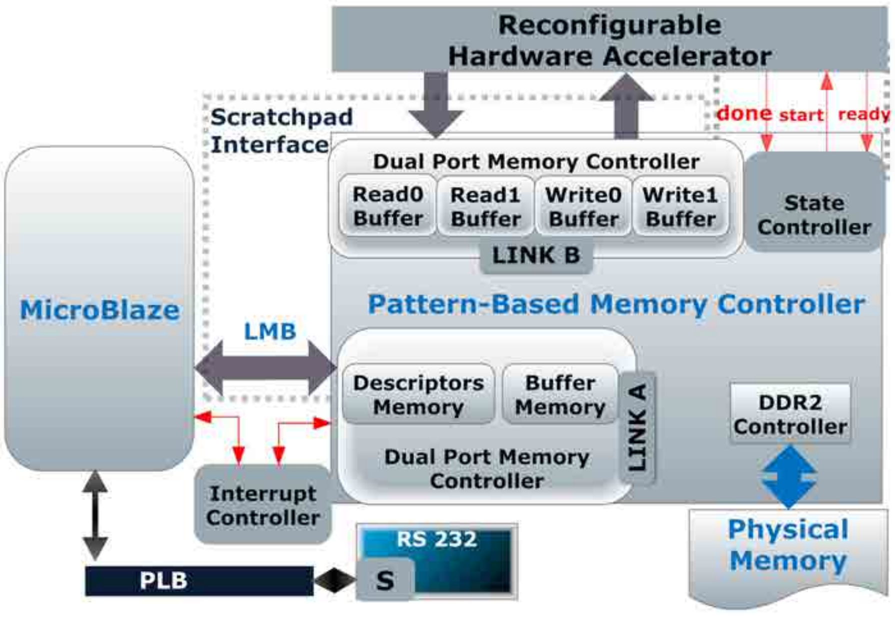


Figure 3.3: Test Environment: PPMC based SoC with Reconfigurable Hardware Accelerator

3.2.3 PPMC based SoC

The PPMC based system is shown in Figure 3.3. In this architecture, dual-port memory controllers are used to share the *Descriptor Memory* between the MicroBlaze and the PPMC. The MicroBlaze is connected with the PPMC for programming PPMC's *Descriptor Memory*, to execute applications (more information in Table 3.1) and to display results using a serial link. The PPMC is connected to the reconfigurable hardware accelerator using PPMC's *SSSI* and a state controller. The PPMC state controller, shown in Figure 2.11 (a), is used to control the hardware accelerator and the buffer memory. A 256 MByte (32M x 16) of DDR2 SDRAM having SODIMM I/O module is used in the design with a peak main memory bandwidth of 1600MB/s, as it has a clock frequency of 100MHz, and a data bus width of 64 bits. Four memory buffers (two read and two write) are specified to accesses tiled data to/from main memory. The system consumes 4280 flip-flops, 3581 LUTs and 10 BRAMs on a V5-Lx110T device. Due to the light weight of PPMC, the proposed architecture consumes 38% less slices than the baseline MPMC based SoC, that is described below.

3.2.4 Baseline MPMC based SoC

A Xilinx FPGA based state of the art High Performance Multi-Port Memory Controller (MPMC) system is used (Figure 3.4) as a baseline. The architecture has 64 kB and 4

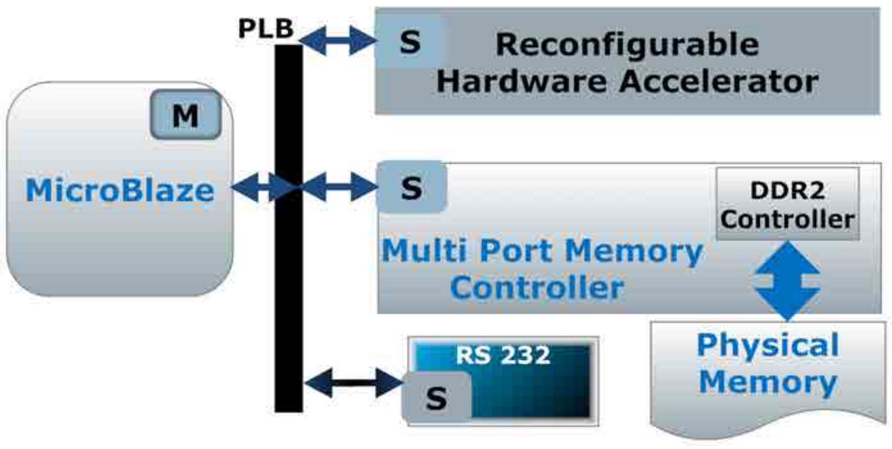

PLB
S
Reconfigurable Hardware Accelerator
M
MicroBlaze
S
DDR2 Controller
Multi Port Memory Controller
RS 232
S
Physical Memory

Figure 3.4: Test Environment: MPMC based SoC with Reconfigurable Hardware Accelerator

kB of data and instruction cache respectively. The design uses Xilinx Cache Links (IXCL/DXCL) for I-Cache and D-Cache memory accesses. MPMC is a fully parameterizable memory controller, providing an efficient interfacing between the processor and SDRAM using the IBM CoreConnect Processor Local Bus (PLB) bus interface. By default, MPMC uses the PLB interface to transfer data from/to SDRAM. PLB supports fixed-burst data transfers, and 8-word cache line read/write transfers. The MPMC controller (like other memory controllers) takes data transfer instructions from the MicroBlaze processor and performs memory operations. A modular DDR2 SDRAM [48] controller (with the PLBv46 Wrapper) is used with the MPMC system to access main memory. The controller accesses data from main memory and performs the address mapping from physical address to SDRAM address. A 256 MByte (32M x 16) of DDR2 SDRAM having SODIMM I/O module is connected with the DDR2 memory controller. The DDR2 controller has a peak main memory bandwidth of 1600MB/s as it has a clock frequency of 100MHz, and a data bus width of 64 bits. The MPMC system can also use Video Frame Buffer Controller (VFBC) and Central Direct Memory Access (CDMA) direct memory access controllers. The VFBC and CDMA are optional and are used to improve the performance of the DDR2 SDRAM controller by managing complex patterns in hardware. For each application, the controller (VFBC or CDMA) that better suits the application access pattern is included in the design. MicroBlaze *SSP* is used to control the system. The target architecture has a single precision floating point unit, 16 KB of each instruction and data cache. The design uses 9612 flip-flops, 9388 LUTs and 14 BRAMs in a Xilinx V5-Lx110T device. Xilinx SDK is used to compile the application kernels using the library generator (libgen) and a platform-specific gcc/g++ compiler and generate the final executable object file, with optimization level (`-O2`).

3.2.5 Test Applications

The application kernels used to validate the design are shown in Table 3.1. These kernels are chosen to characterize different data access patterns. The results are validated by comparing the execution time of these kernels on a MicroBlaze system with PPMC and MPMC support. FIR, FFT and Laplacian kernels are executed in an *ASHA* core.

Table 3.1: Brief description of application kernels

Application Kernel	Description	Access Pattern	Executed By
Thresholding	An application of image segmentation, which takes streaming 8-bit pixel data and generates binary output.	Streaming	*SSP*
Finite Impulse Response	Calculates the weighted sum of the current and past inputs.	Streaming	ASHA
Fast Fourier Transform	Used for transferring a time-domain signal into corresponding frequency-domain signal.	1D Block	ASHA
Laplacian solver	Applies discrete convolution filter that can approximate the second order derivatives.	2D Tiling	ASHA
Matrix Multiplication	X = Y $\times$ Z	2D Tiling	SSP
3D-Stencil Decomposition [49]	An algorithm that averages nearest neighbor points in 3D.	3D Tiling	*SSP*

In order to give a standard interface to ROCCC IPs with the PPMC system, the state controller presented in Section 2.1.3.3 is used.

3.3 Results and Discussion

Figure 3.5 shows a plot with Read/Write data accesses for MPMC and PPMC based systems. The X-axis presents 1D/2D/3D datasets that are read and written from/to the main memory and the scratchpad memory. The Y-axis of the plot represents the number of clock cycles consumed while accessing the tiled dataset with the logarithmic scale. The MicroBlaze with MPMC has load/store (see Section 1.3.1.1.1) data access patterns, whereas PPMC accesses multiple noncontiguous streams using patterns. The results show that PPMC memory access times are 10 times faster than a generic memory controller.

Figure 3.6 shows the execution time (clock cycles) for the application kernels, executed on MPMC and PPMC based systems. The X and Y axis represent application kernels and number of clock cycles, respectively. By using our PPMC system, the re-

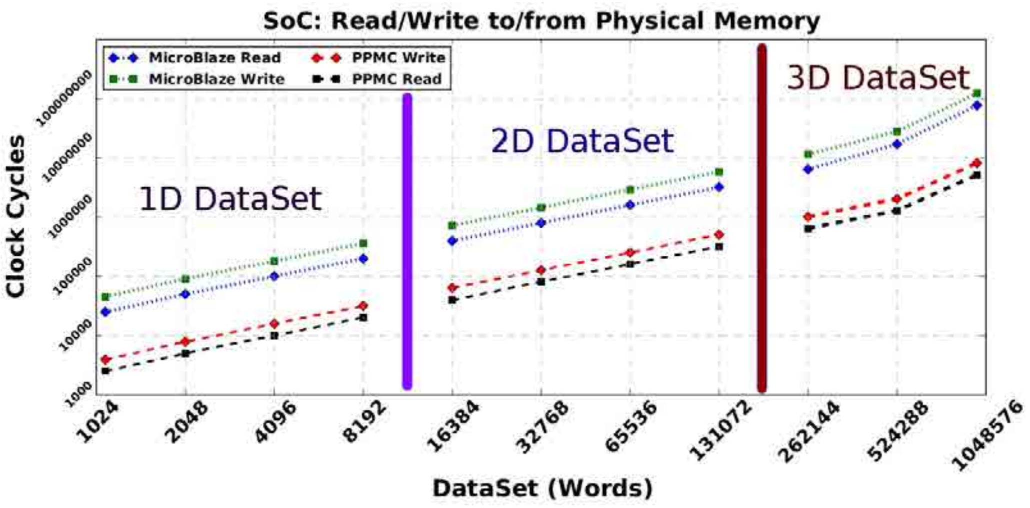


Figure 3.5: Clock Cycles: PPMC and MicroBlaze Read/Write data to/from Main Memory

sults show that the Thresholding and FIR application kernels respectively achieve 32x and 22x of speed-up. Both applications kernels have a streaming data access pattern. The results confirm that PPMC reduces the *Main Memory* data access delay and that it decreases memory request/grant time by overlapping it with application execution time. The FFT application kernel reads a 1D block of data, processes it and then writes it back to the *Main Memory*. This application observes an 8.2x speed-up. The Matrix

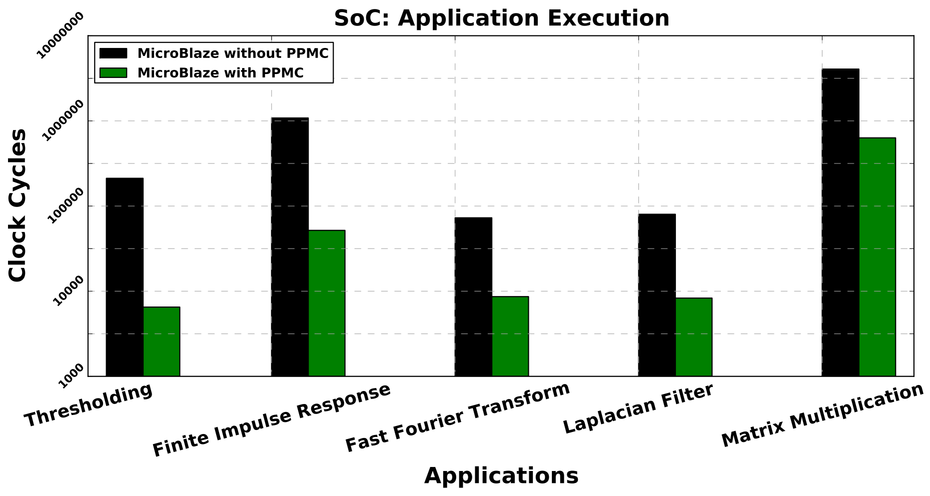


Figure 3.6: Execution time (clock cycles) of the applications

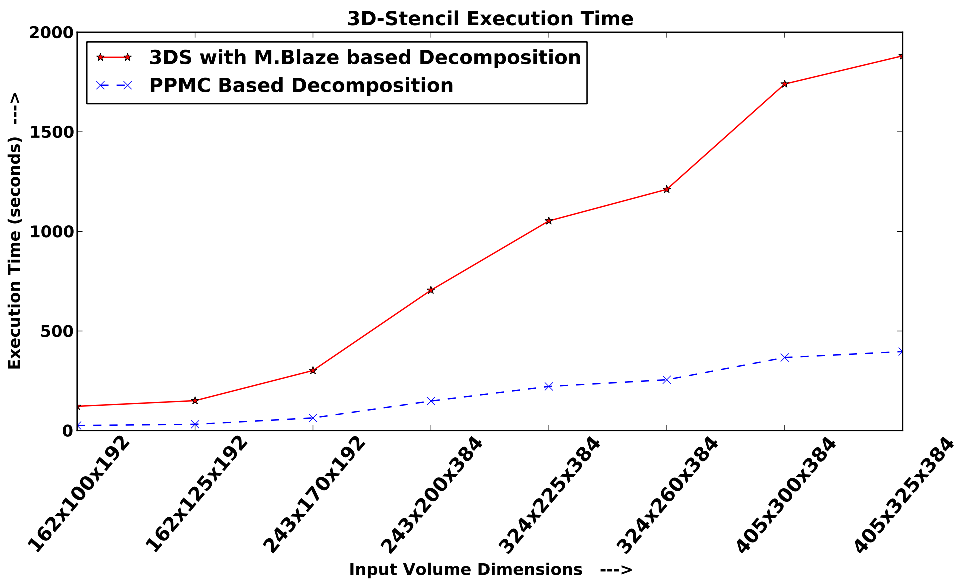


Figure 3.7: Execution time for various volume sizes with domain decomposition done by the host processor and tiling done by the PPMC for the 3D-Stencil

Multiplication and Laplacian applications respectively achieve speed-ups of 6.4x and 9.6x. Both applications have 2D-Data access and both use data tiling.

Figure 3.7 shows the execution time for the 3D-Stencil for various sizes of input volumes. This shows the scalability of MPMC and PPMC with large input data sets. The stencil runs for 500 time steps. The usage of PPMC for directly handling the tiling for 3D volumes improves around 5x the performance of the stencil kernel as compared to software based volume decomposition. In our evaluations for the 3D-stencil, the accelerator directly writes the tiling commands to the PPMC.

3.4 Conclusion

The key to efficiency for many applications is to maximize data reuse, thus fetching input data only once. This is also true for the compute-intensive problems having complex access patterns. To improve the memory-processor data access bottlenecks, a simplified PMC controller is used, called PPMC. The PPMC *Descriptor Memory* handles regular and complex access patterns. The PPMC based SoC supports a single core (ASHA or SSP) at a time. The PPMC uses an arbiter instead of scheduler that processes access pattern requests of a single core using *Address Manager*. A simple *Scratchpad Buffer Memory* is used to store on-chip data with no *Data Manager*. PPMC can be programmed by the microprocessor using HLL or directly from the accelerator using a special command interface. The experimental evaluations based on the Xilinx MicroBlaze accelerator system demonstrate that the PPMC based approach improves utilization of hardware resources and efficiently accesses the *Main Memory*. The PPMC is implemented and tested on a Xilinx ML505 Evaluation Platform with a 65 nm Virtex-5 XC5VLX110 FPGA. In order to prove the effectiveness of the proposed controller, we compared the results with non-PPMC based system which uses a state of the art memory controller that relies on the MicroBlaze softcore as a host processor. The evaluation uses six memory intensive application kernels: Laplacian solver, FIR, FFT, Thresholding, Matrix Multiplication, and 3D-Stencil. The results show that the PPMC system achieves at least 10x of speed-up for 1D, 2D and 3D memory accesses and improves application performance between 6.4x and 32x over a non-PPMC based setup.

4

A Bus Controller for Graphics System

In this era graphics system have shown a great impact in our lives not only as a commodity itself but also for specialized use like medical imaging, defense, communication systems, etc. We have witnessed a major change in graphics applications that are being investigated and developed. Various powerful and expensive platforms to support graphical applications appeared in recent years. All these platforms require a high performance core that manages and schedules the high speed data of graphics peripherals (camera, display, etc.) and an efficient on chip scheduler.

Multimedia graphics hardware has been evolving rapidly from simple architecture to the complex hybrid system. The first color graphics adapter CGA was introduced in the early 80s by IBM. The mobile industry started integrating graphics applications in their products in early 90s using software algorithms of PCs and workstations. The graphics processors have been evolving from a fixed-function unit to a massively powerful computing machine, and they are becoming a common component for hand-held devices. The complexity of graphics processor rises as consumers demand more functionality and diverse performances from such devices. On the other hand, deploying latest features in graphics system increases the cost and power of the system.

A typical system on chip system (SoC) (shown in Figure 4.1) comprises one or two processor cores (master core and graphic core), a programmable crossed bus arbitration module [50], an interrupt and direct memory control module along with control bus and data bus. The low power devices have a single processor and perform camera

[1] Chapter 4 is based on the publications : [36] and [37].

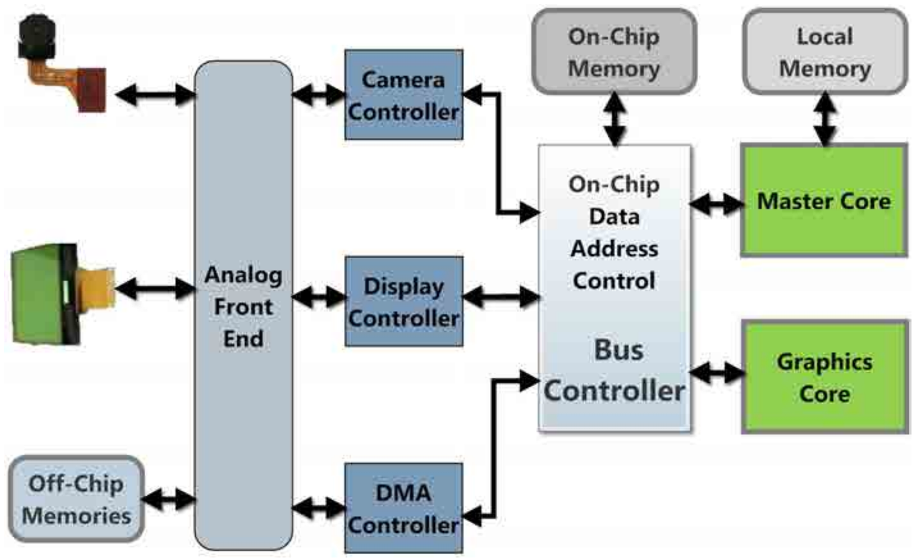


Figure 4.1: Generic Graphics System on Chip

processing in software. The single core itself performs data transfer, data management and computation which affects the system performance. In dual core system, the master core can be used for real time operation, and the graphic core is reserved for image signal processing. To improve performance of graphics system, an intelligent controller is needed that has efficient on-chip specialized memory and memory system, intelligent front-end/back-end scheduler, fast I/O link and which supports programming model that manages memory accesses in software so that hardware can best utilize them. We propose such graphic system in PMC called Programmable Graphics Controller (PGC).

In this work, we integrated PMC with graphics system using low-power and high-performance bus controller called the Programmable Graphics Controller for SoC. The PGC resides in on-chip *Bus Unit* and holds data transfer pattern and control instructions in its *Descriptor Memory*. The PGC *Scheduler* provides on-chip and off-chip bus interconnects and controls data transfer without the support of complex bus matrix and master processor. The approach reduces the master/slave arbitration delay, bus switching time, gives promising interconnection approach for multi-peripherals with the potential to exploit parallelism while coping the memory/network latencies and balances the work load. PGC bus *Scheduler* uses *FIFO* policy that deals single core data transfer requests and provides low-cost and simple control characteristics. The *Address*

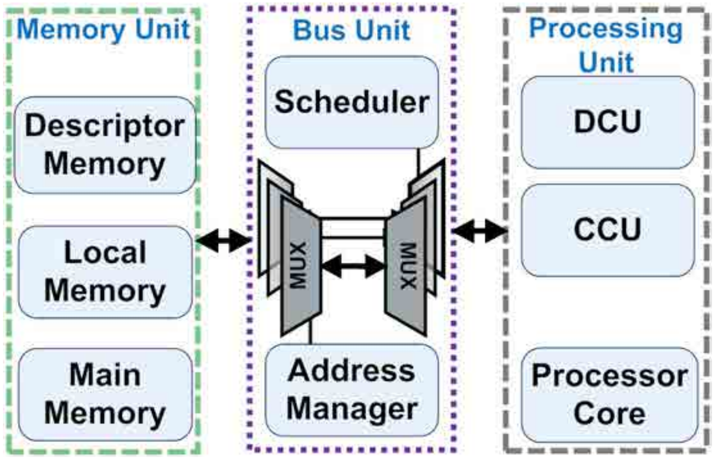


Figure 4.2: PGC : Internal Structure

Manager with no *Data Manager* arranges multiple bus access requests and communicates with integrated processing units. We integrated dedicated hardware accelerators in the design as they have low power consumption and provide high performance. The PGC supports multi-peripherals (camera, display) and processor core without support of the master core and operating system (OS). The integration of PGC with peripherals facilitates the graphics system to overcome wire (interconnection) and memory read/write delays and improves the performance of application kernels by arranging complex on-chip data transfers.

4.1 PGC Graphics System Specification

Architectural investigation for graphics system ranges from high level system architecture to analog and circuit-level design. The PGC architecture covers the reuse of processing elements, data parallelism and the network architecture. In this section, we describe the specification of the PGC system and design its architecture. The section is further categorized into four subsections: *Overview of PGC System*, the *Processing Units*, the *Memory Unit* and the *Bus Unit*.

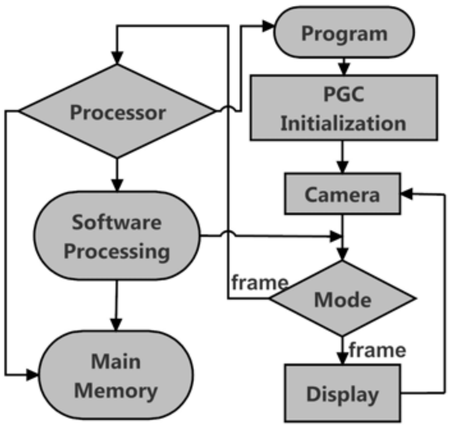


Figure 4.3: PGC : Flowchart

4.1.1 Overview of PGC System

The PGC inner architecture is shown in Figure 4.2, which shows the interconnection with the processing units and memory. The system uses combined hardware/software solution that includes hardware accelerators and an RISC processor core. The camera control unit (CCU) and display control unit (DCU) are custom hardware accelerators and control the camera sensor and the display unit respectively. The *local memory* holds the CCU/DCU data for basic image/video processing using the *Processor Core*. To store high resolution images the *Main Memory* is integrated. The *Descriptor Memory* is used to hold CCU/DCU program description and data transfer information. Depending upon the data transfer the *Address Manager* takes single or multiple descriptor from *Descriptor Memory* and schedules the data movement for *CCU* and *DCU*. The PGC *Scheduler* handles the concurrent bus request by the CCU and DCU and rearranges multiple bus access requests and arbitrates data transfer without creating bus contention.

We define two use cases of graphic system (shown in Table 4.1); the *video mode* and the *snapshot mode*. The *processing* is used to perform filtering, compression, transformation, etc. on input image. Each use case has two variants: with-processing and without-processing. The resolution of video mode is selected to fit in *local memory* of the target device. In current design, the *video Mode* supports resolution image up to 640×480. It reads multiple frames (images) per second (*fps*) from the camera sensor and transfers them to display unit. The *video mode* is further divided into two

Table 4.1: Graphics System: Use Case, Mode of Operations

Use Case	Processing	Pixel$_{Depth}$	Resolution	frame/sec (fps)
Video-Mode				
Single-Camera Video	With/Without	24-bit	VGA = 640×480	variable
Dual-Camera Video	With	24-bit	VGA = 640×480	up to 150
Snapshot Mode	With/Without	24-bit	QSXGA = 2560×2048	1

modes. The *single-camera video* uses a single image sensor and *dual-camera video* supports two image sensors. A dual or multi-camera graphics system can be used for 3D-graphics using geometric transformation and projection plane [51]. The *snapshot mode* of operation takes a still image from the image sensor, performs software processing if required and writes to *Main Memory*. The *snapshot mode* supports a maximum resolution of 2560×2048 with 24-bit pixels (16 Mega colors) depth.

The working operation of the PGC system is shown in Figure 4.3. During programming-time, the *Descriptor Memory* and program memories of PGC and CCU/DCU are initialized. The program memory holds the instructions of CCU/DCU program registers and data transfer. During the initialization, the PGC programs the CCU and DCU according to the different use cases.

4.1.2 Processing Unit

The PGC uses two types of cores: the *ASHA* and the processor core. Camera Control Unit (*CCU*) and Display Control Unit (*DCU*) *ASHA* are used in the design to control camera and display units respectively. The *CCU* grabs raw data from Image sensor processes it and transfers it to the system via on-chip bus. The major function blocks of *CCU* are Camera Interface Front-End, Image Signal Processor, Color processing, Scaling, Compression, and Bus controller. The hardware utilization of each function block in term of memory and logic elements is shown in Figure 4.4. Each *CCU* block has memory mapped internal registers (shown in Figure 4.4) that can be initialized and programmed by the processor core. The *DCU* is used to control and display image data on LCD panel. The *DCU* supports LCD 16bpp up to 24bpp colors and user defined resolution from VGA to QSXGA. Programming is done by register read/write

Module (Name)	Memory bits	Gate Count (k gates)
Image Signal Processor	135128	179
Color Processing	41600	7
Main Resize	83200	34
Memory Interface	0	113
JPEG Encoder	338944	83
Total	598872	416

(a)

Module (Name)	Base Address	End Address	Range Hex
Image Signal Processor	E200 0400	E200 0418	1C
Color Processing	E200 0500	E200 0510	14
Main Resize	E200 0600	E200 064C	50
Memory Interface	E200 0700	0200 0810	114
JPEG Encoder	E200 0900	E200 098C	90
Total	-	-	224

(b)

Figure 4.4: Camera Control Unit: (a) Resource Utilization (b) Reduced Memory Map

transactions using a slave interface. The *DCU* consumes 425 registers and 312 LUTs on a V5-Lx110T FPGA device.

The CCU and DCU data rate is given by Formula *Output*$_{data\ rate}$ (shown in Figure 4.5). The *Analog Interface*$_{width}$ represents the analog port width of Image sensor and display. Both CCU and DCU supports 32-bit parallel interface (Analog Interface$_{width}$) to communicate with image sensor and display (LCD). Each hardware block of *DCU & CCU* is invoked by the processor using memory mapped register sets that change the operation of internal hardware architecture.

A low power and light weight RISC processor core is used to provide programmability, flexibility and software data processing. The processor core changes the features by programming the PGC system using a software interface API. The API can be used to correct design errors, update the system to a new graphic standard and to add more features to the graphics system. The proposed processor core has 16-bit data bus, 16-general purpose registers, custom instruction set, non-pipelined Load/Store access, hardwired control unit, 64KByte address space, total 16-interrupts and memory mapped I/Os. 1KByte of memory is allocated for display and camera control units using chip select. On a V5-Lx110T FPGA device, the core uses 481 registers, 1496

LUTs and 4 Brams.

$$Output_{data\ rate} = Resolution * fps * \frac{Pixel_{Depth}}{Analog\ Interface_{width}}$$

Figure 4.5: CCU and DCU Data Rate

The PGC *Local Memory* is shared between the processor, CCU and DCU. During video mode, two frames buffers are required, one for processing and other for displaying. Each VGA frame has 900KByte of size, therefore, almost 2MBytes of *local memory* are reserved. To save the image for snapshot mode we use main memories. The slowest type of memory in the graphics system is *Main Memory* and is accessible by the whole system. The *Main Memory* may use SDRAM, SD/SDHC cards, etc. interfaces to read/write data.

4.1.3 Bus Unit

Two buses are used in the graphics system which are the *Graphics Bus* and the *System Bus*. The *Graphics Bus* is used as on-chip bus for internal communication between the processing units and *Local Memory*. The *System Bus* is used to communicate with external peripherals such as main memories. Both buses can operate in parallel. The *Bus Unit* is further divided into three sections: the *Bus Specification*, the *Bus Control Unit* and the *Bus Interconnect Network*.

4.1.3.1 Bus Specification

It is important to calculate the required data-rate for each *use case* before selecting and configuring the *Bus Unit*. This section is further divided into three subsections: *Graphics Bus Specification*, *System Bus Specification* and *Bus Usage*.

4.1.3.1.1 Graphics Bus Specification The clock of the camera and display is directly synchronized with the output data hence define the bandwidth of the *graphics bus*. The actual theoretical data rate of the *graphics bus* (GBB) is the total bandwidth of master sources (shown in Figure 4.6). For example, during video mode without-processing the PGC reads streaming data from CCU and writes directly to DCU. For

$$GBB = \sum_{n=1}^{max} Master\ Source\ Bandwidth_n$$

$$GBB_{SC}\text{=}Image\ Sensor[Pixel_{Depth} * Resolution_{max} * fps_{max}]$$

$$GBB_{SC}\text{=}Image\ Sensor[24 * 640 \times 480 * 150]$$

$$\simeq 1.05Gb/s \simeq 132MB/s$$

$$GBB_{SCP}\ \simeq 264MB/s$$

$$GBB_{DC}\ \text{=}Image\ Sensor_a + Image\ Sensor_b$$

$$\simeq 2.1Gb/s \simeq 264MB/s$$

$$SBB_{SN}\ \text{=}Image\ Sensor[24 * 2560 \times 2048 * 1]$$

$$\simeq 120Mb/s \simeq 15MB/s$$

$$Percentage\ of\ Bus_{usage} = \frac{Source_{Bandwidth}}{MasterBus_{freq}} * 100$$

Figure 4.6: Graphics Bus Required Bandwidth

video mode with-processing, the PGC takes video frames from CCU, writes it to local memory for processing and then transfers the processed frame to DCU. In this case the PGC operates CCU and DCU in parallel, and therefore, the bandwidth of the *graphics bus* is the sum of CCU and DCU data-rates. For dual camera, the PGC takes two video frames and transfers them to CCU. The required graphics bus bandwidth (shown in Figure 4.6) with a single camera without processing and with processing is given by the formula GBB$_{SC}$ and GBB$_{SCP}$ respectively. Figure 4.6 also presents the bandwidth of dual camera GBB$_{DC}$. The *local memory* provides high bandwidth and has 2 cycles of latency for individual transfer. The *graphics bus* manages multiple read/write access transactions in a single transfer and pipeline the multiple stream that reduces the overhead of *local memory*$_{Latency}$ and improves the bus performance. After calculating bus bandwidth and considering the different use cases, we selected a bus with 100 Mhz of clock speed and 32 bit-width.

4.1.3.1.2 System Bus Specification The *System Bus* manages the data transfer during snapshot mode. The PGC reads data from image sensor and writes it to main

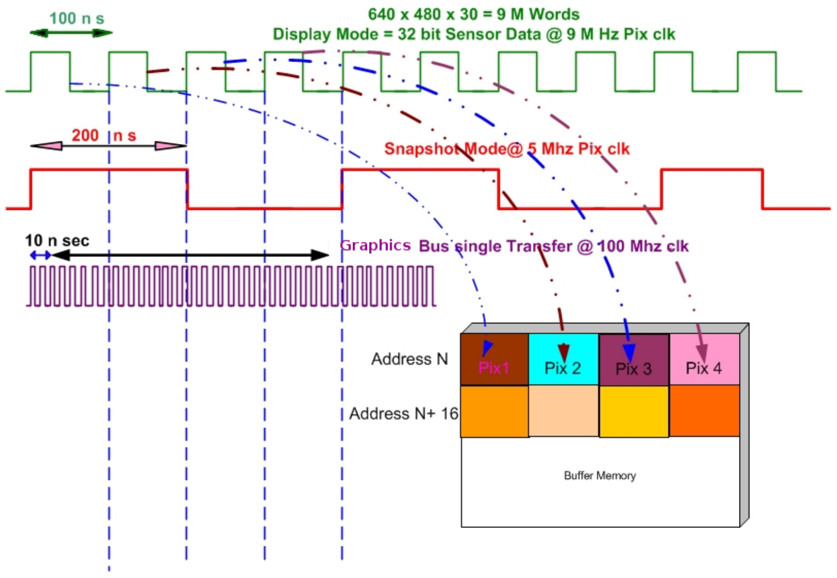

100 n s
640 x 480 x 30 = 9 M Words
Display Mode = 32 bit Sensor Data @ 9 M Hz Pix clk
200 n s
Snapshot Mode@ 5 Mhz Pix clk
10 n sec
Graphics Bus single Transfer @ 100 Mhz clk
Address N
Address N+ 16
Pix1
Pix 2
Pix 3
Pix 4
Buffer Memory

Figure 4.7: Bus Unit Timing Cycles

memory. The bandwidth requirements of *System Bus* for snapshot-mode is given by the Formula SBB$_{SN}$ (shown in Figure 4.6).

4.1.3.1.3 Bus Usage Figure 4.7 shows the bus timing and bus usage information for the video and snapshot modes. The finalized graphics bus has 400 MB/s of bandwidth that means it takes 10 nsec to transfer 1 (32bit) pixel of data. For example, the graphics bandwidth for video of 30 *fps* (without-processing) is 9 Mega pixels per second. This means each pixel takes 111 nsec and occupies graphics bus for 9% of its time, given by the formula $Percentage\ of\ Bus_{usage}$ (shown in Figure 4.6). For video mode with-processing, the *Graphics Bus* takes 111 nsec to transfer one pixel from CCU to *local memory*, and the same time is taken while transferring it to display accelerator. Similarly, video mode needs 18 Mega pixels of bus bandwidth, that takes 56 nsec to transfer a pixel between image and display accelerators. The display camera interface for video mode utilizes the graphics bus for approximately 14% of total bus time and relinquishes bus for another period of time. The snapshot mode requires bus bandwidth of 5 Mega pixels to transfer one image of QSXGA resolution from the CCU to the *Main Memory*.

4.1.3.2 Bus Control unit

The PGC control unit uses *Descriptor Memory*, *Scheduler* and *Address Manager*, to manage the processing units and memory units. The *Descriptor Memory* holds *Descriptors* that define the data movement between CCU/DCU, processor core and memory unit. The *Descriptors* allow the programmer to describe the shape of memory patterns and its location in memory. A single *Descriptor* is represented by the parameters source address, destination address, stream and stride. The address parameters specify the master and slave cores. Stream defines the number of pixels to be transferred. Stride indicates the distance between two consecutive memory addresses of a stream. C/C++ function calls are provided to define a complex pattern in software.

The PGC bus *Scheduler* along with *Address Manager* (shown in Figure 4.7) arrange requests coming from single or multi-bus masters and arbitrate master processing units. A *bus master* provides address and control information to initiate read and write operations. A *bus slave* responds to a transfer that is initiated by one of the master's core. The *Address Manager* holds the address and control information of *bus slaves*. The *Scheduler*'s *interrupt controller* reads requests from master cores and routes them to the slave. The *Address Manager*'s *decoder* determines for which slave a transfer is destined for. The PGC bus holds two types of status registers: the source status and the slave status registers. The status registers indicate the states of each master and slave, such as request, ready, busy and grant. The *Scheduler* and *Address Manager* administer the status register of master and slave cores respectively.

At run-time, a master core generates a request, the *interrupt controller* reads the request and updates its status registers. The *Scheduler* reads data transfer information of the master and slave cores from *Descriptor Memory* and transfer slave core information to the *Address Manager*. The PGC *Address Manager* decodes the address of each transfer and provides a select signal for the slave that is involved in the transfer and provides a control signal to the multiplexers. A single *master-to-slave* multiplexer (*MUX*) is controlled by the *Scheduler*. The *master-to-slave MUX* multiplexes the write data bus and allocates the data bus for corresponding master after getting a response signal from a *slave-to-master MUX*. A *slave-to-master MUX* multiplexes the read data bus and response signals from the slaves to the master. Multiple *master-to-slave* and *slave-to-master* multiplexers can be added to implement a multi-layer *Bus Unit*. The

PGC *Bus Unit* can be programmed up to eight layer bus which requires eight pairs of *master-to-slave* and *slave-to-master* multiplexers.

4.1.3.3 Bus Interconnect

To connect the graphics components together, a bus interconnection is described (shown in Figure 4.8). We select a double layer *Bus Interconnect* (*System Bus* and *Graphics Bus*) for the design due to its design simplicity and low power consumption. Each layer is controlled by a pair of *master-to-slave* and *slave-to-master* multiplexers. The PGC *Scheduler* and *Address Manager* control the pairs of multiplexers. The bus is used to read/write high resolution image to/from main memories. The *Graphics Bus* is employed to provide high speed link between the CCU, DCU, processor and local memory components. Current PGC *Graphics Bus* has 5 Masters and 4 Slaves, therefore, the *Bus Unit* is configured accordingly. The proposed *Bus Unit* provides standard

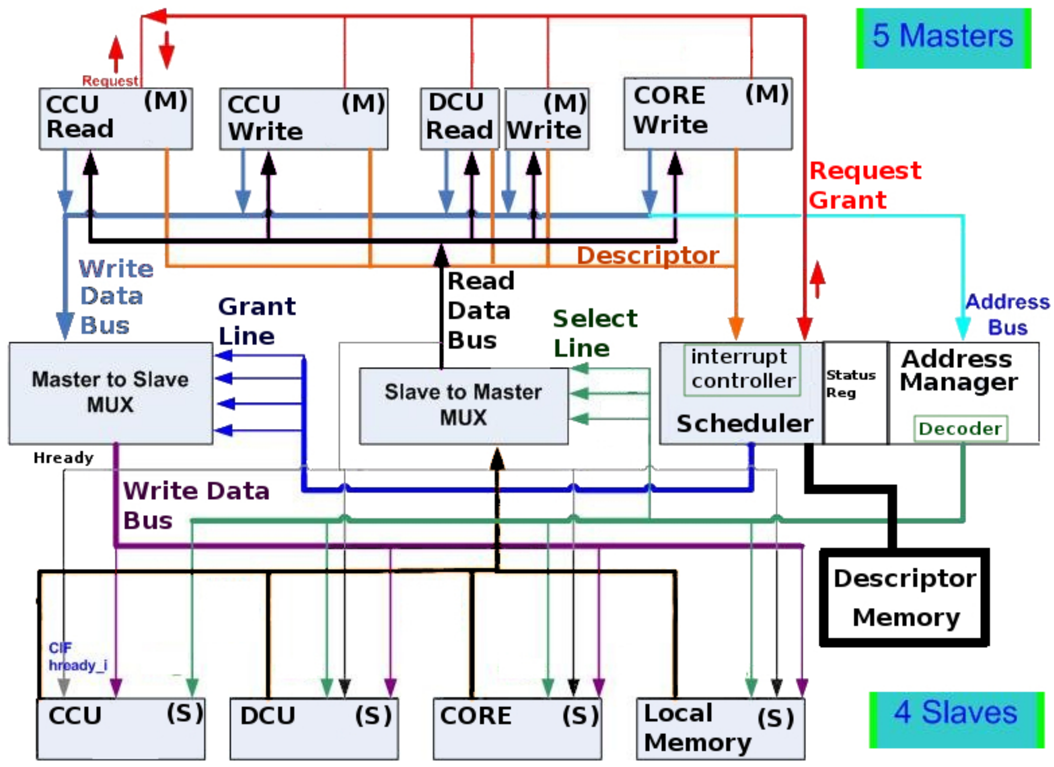


Figure 4.8: PGC Graphics Bus Unit

communication protocol and implements the features required for high-performance.

4.2 Experimental Framework

In this section, we describe and evaluate the PGC based graphics system. In order to evaluate the performance of the PGC system, the results are compared with a generic graphics system managed by the MicroBlaze processor. The Xilinx Integrated Software Environment and Xilinx Platform Studio are used to design the graphic systems. The power analysis is done by Xilinx Power Estimator (XPE). A Xilinx ML505 [52] development board is used to test the systems. For the implementation of graphics system the THDB-D5M Camera and the TRDB-LTM LCD Touch Panel by Terasic have been chosen. This section is divided into two subsections: the *MicroBlaze based Graphics System* and the *PGC based Graphics System*.

4.2.1 MicroBlaze based Graphics System

The FPGA based MicroBlaze system is proposed (Figure 4.9) to operate graphics system. The design (without CCU & DCU) uses 8047 flip-flops, 6643 LUTs and 51 BRAMs in a Xilinx V5-Lx110T device. The system architecture is further divided into the *Processor Core* and the *Bus Unit*.

The Processor Unit: Two MicroBlaze cores [20] are used in the graphics system which are the Master core and the Graphics Core. The Master core is used to program, schedule and manage the system components. The camera and display hardware

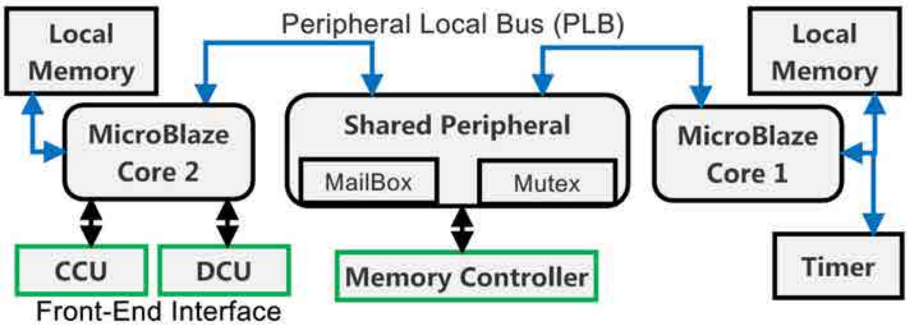


Figure 4.9: MicroBlaze based Graphics System

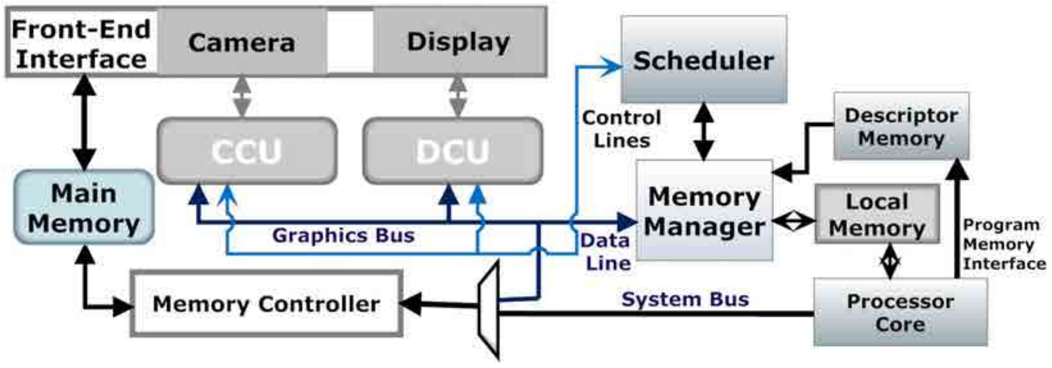


Figure 4.10: PGC: Graphics System

scheduling and data memory management are controlled by Graphics processor. Both cores use local memory Bus (LMB) [53] to link with local-memory (FPGA BRAM) that offers single clock cycle access to the local BRAM.

The Bus Unit: In the design, a Processor Local Bus (PLB) [54] provides connection between peripheral components and microprocessors. The PLB has 32 bit-width and is connected to a bus control unit, a watchdog timer, separate address read/write data path units, and an optional DCR (Device Control Register) slave interface that provides access to bus error status registers. The Bus is configured for single master (MicroBlaze) and multi slaves. The PLB provides maximum of 400 MBytes of bandwidth while operating at 100Mhz and 32-bit width, with byte enables to write byte and half-word data.

4.2.2 PGC based Graphics System

The PGC based graphics system is shown in Figure 4.10 having components similar to the MicroBlaze based graphic system. The implementation details of PGC based graphics system are addressed in Section 4.1. The main difference between PGC and MicroBlaze based systems is that PGC system takes descriptors during initialize-time and at run-time it manages and schedules data transfer without the support of the processor. The processor core and *System Bus* remain free for the use cases which do not involve processing. The design (without CCU and DCU) uses 4547 flip-flops, 3843 LUTs and 35 BRAMs in a Xilinx V5-Lx110T device.

4.3 Results and Discussion

This section analyzes the results of different experiments conducted on the different graphic systems. The experiments are classified into four subsections: *Bus Performance*, *Snapshot Mode Performance*, *Applications Performance* and *Area & Power*.

4.3.1 Bus Performance

To measure the bus performance, the graphic systems are executed on *Video Mode* (without-processing) having fixed resolution (640×480) and variable frame rate (*frame per second* - *fps*). The image sensor is programmed to transfer a variable number of frames (*fps*), and each frame has VGA quality. Inside DCU, we have integrated a controller that detects the *fps* and the speed at which frames are coming. A hardware timer is added to the on-chip bus controller that measures clocks used to transfer frames between master to slave peripherals. This section discusses results for *Single-Camera Bus Bandwidth* and *Multi-Camera Bus Bandwidth*.

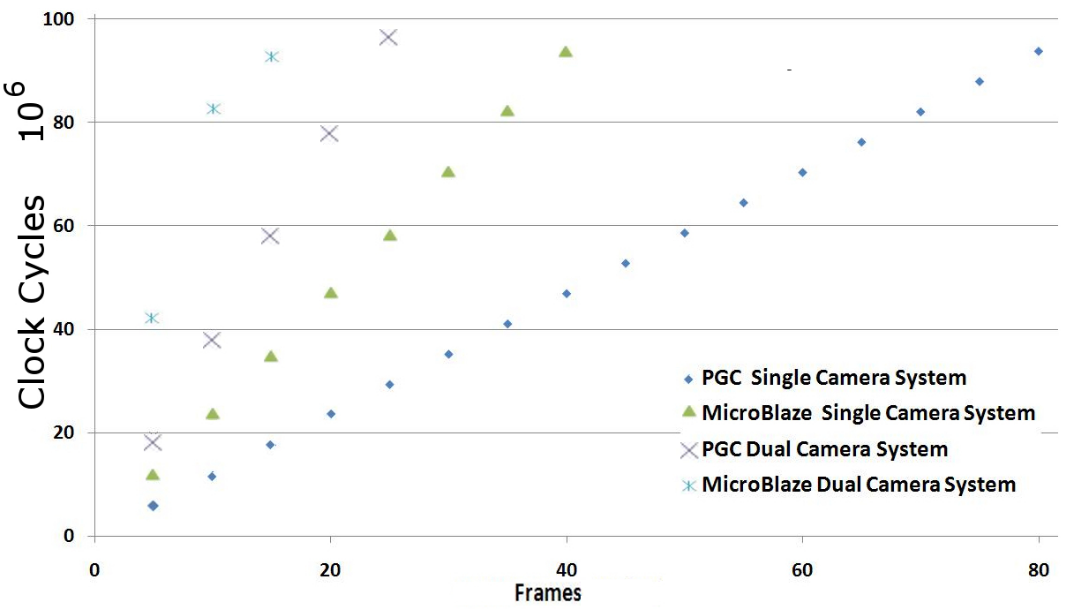


Figure 4.11: Single and Dual Camera Systems: Bandwidth For Different Frame Rate

4.3.1.1 Single-Camera Bus Bandwidth

In this section, we compare the bus performance of graphic systems while using single image and display units. Figure 4.11 shows the on-chip data bus transfer speed of PGC and MicroBlaze systems for different video frame rates. A single THDB-D5M image sensor is used. It can operate up to 150 *fps* with VGA resolution. The X-axis presents video of different *fps*. The Y-axis shows measured bandwidth for different videos frame rates. To measure the bandwidth we calculate the time to transfer video from CCU to DCU. Theoretically the PLB and the graphic bus support video of VGA quality more than 100 *fps*. In practice, there are on-chip bus arbitration and request grant time delays. By using the PGC system, the results show that the system manages video for higher *fps*. While the MicroBlaze based graphic system supports video up to 40 *fps*, with higher *fps* the video starts flicking. The system uses a separate processor core that manages the data movement of CCU and DCU. The PGC allows graphics system to operate *Video Mode* up to 85 *fps*. The PGC *control unit* controls the CCU and DCU without the intervention of the processor core which reduces the master/slave request/grant time.

4.3.1.2 Multi-Camera Bus Bandwidth

In this section, two THDB-D5M image sensors are used that generate two separate simultaneous video streams and apply *Alpha blending* application to evaluate the performance of the dual camera system. Each camera is operating at VGA color resolution. The video of dual image sensors is combined into a single stream, processed by the graphics core and then displayed. The key issue of the dual-camera system is receiving the images synchronously, in the right format and on the right bus. The graphic system sends the configuration data to both image sensors and ensures that they are properly configured and synchronized. Once both sensors are set up and synchronized, both sensors begin to transmit image data. The graphic system looks for the appropriate control characters, so it recognizes the start of the frame and start of a line for each sensor. The PGC system performs it by looking for a control character and sequence of sensors commands. *Alpha blending* is applied to give a translucent effect to the incoming video stream. The application blends the color value of the consecutive pixels of image sensors of the same position. This blending is done according to the alpha value associated with the pixel. After blending, the result color value is updated to the frame buffer of the DCU. Results show (Figure 4.11) that PGC system handles dual camera system and support system up to 30 fps. The MicroBlaze based dual-camera graphics system supports videos only up to 15 *fps*. The PGC on-chip *scheduler* and *decoder* update multi-camera information in status register. This allows both cameras to synchronize without using extra clocks.

4.3.2 Snapshot Mode Performance

In this section, the performance is measured by executing PGC and MicroBlaze systems in *Snapshot Mode*. During *Snapshot Mode* the system reads one still image of QSXGA resolution from CCU using *Graphics Bus* and writes it to *Main Memory* using *System Bus*. The MicroBlaze based system and PGC take 22.17M and 7.07M clocks respectively to transfer an image. The *Snapshot Mode* results show that the PGC system achieves 3.1x of speed compared to MicroBlaze based system. The PGC directly controls the *Graphics Bus*, *System Bus*, *CCU* and *Main Memory*, therefore it takes less clocks to read data from *CCU* to synchronize different units, transfer data from *Graphics Bus* to *System Bus* and write data to *Main Memory*. The MicroBlaze based system uses a separate bus controller that controls bus system and a DMA controller that transfers data from *CCU* to *Main Memory*.

4.3.3 Applications Performance

In this section, we execute some application kernels that perform image processing. The image is saved in *Main Memory* (SDRAM), the processor core reads the 4KBytes

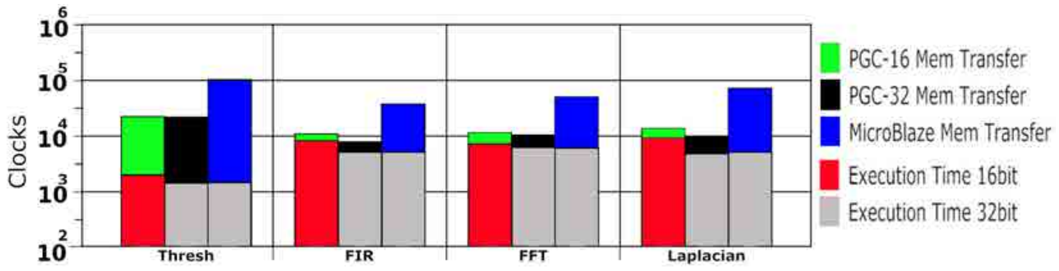


Figure 4.12: Application Performance

image, performs computation and then writes it back to *Main Memory*. To achieve low power, the application kernels are executed by a 16-bit processor core on PGC system. Alternatively, a 32-bit MicroBlaze core is also used with PGC system to get higher performance. Figure 4.12 shows time (clock cycles) to process application kernels. The X and Y axis represent application kernels and number of clock cycles, respectively. The Y-axis has logarithmic scale. Each bar represents the application kernel's execution time with 16/32 bit cores and memory access time. By using the PGC system with 16-bit and 32-bit cores, the results show that thresholding (Thresh) applications achieve 4.6x and 4.7x of speedup respectively over the MicroBlaze-based graphics system. This application kernel requires single pixel element and very few operations, and therefore, it achieves almost the same performance on 16-bit and 32-bit cores. The FIR application has streaming data access pattern and performs multiplication and addition. The PGC 16-bit and 32-bit cores achieve 3.4x and 4.7x of speedup respectively. The FFT application kernel reads a 1D block of data, perform complex computation and writes it back to *Main Memory*. This application achieves 4.4x and 4.8 of speedup. The Laplacian application processes over 2D block of data and achieve 5.2x and 7.4x of speedup. The PGC places access patterns on *Descriptor Memory* at program-time and are programmed in such a way that few operations are required for generating addresses at run-time. The MicroBlaze based system uses multiple load/store or DMA calls to access complex patterns. The speedups are possible because PGC can manage data transfers with a single *descriptor*. At run-time, PGC takes *descriptor* from *Descriptor Memory* and manages them, whereas the baseline system is dependent on the processor core that feeds data transfer instructions. The stand-alone working operation of PGC removes the overhead of processor/memory system request/grant delay.

4.3.4 Power

In comparison with on-chip power in a Xilinx V5-Lx110T device, the MicroBlaze based system dissipates 3.45 watts and the PGC system 2.7 watts. Results show that PGC system consumes 42% fewer slices than the MicroBlaze system. While comparing on-chip static power of MicroBlaze graphics system with the PGC, results show that PGC system consumes 22% of less on-chip static power.

4.4 Conclusion

With the increase of image resolution, the graphics system demands low power, low cost and high performance architecture. In this work, we applied PMC in a low cost and low power graphics system that include a display sensor and multiple image sensors, called PGC. The PGC supports *Scheduler* which handles single core data transfer requests. It takes high resolution images and supports video at higher frame rate without the support of the processor. The PGC system uses *Address Manager* to handle strided, scatter/gather and tiled access pattern that eliminates the overhead of arranging and gathering address/data. The PGC based system is implemented and tested on a Xilinx ML505 FPGA board. The performance of the PGC is compared with the MicroBlaze processor based system having conventional on-chip CoreConnect PLB bus. When compared with the MicroBlaze based system, the results show that the PGC captures video at 2.5x of higher frame rate and achieves 3.4x to 7.4x of speedup while executing different image processing applications.

5

PVMC: Programmable Vector Memory Controller

Data Level Parallel (DLP) accelerators such as GPUs [55] and Vectors [56; 57; 58], are getting popular due to their high performance per area. DLP accelerators are very efficient for HPC scientific applications because they can simultaneously process multiple data elements with a single instruction. Due to the reduced number of instructions, the Single Instruction Multiple Data (SIMD) architectures decrease the fetch and decode bandwidth and exploit DLP for data intensive applications e.g. matrix & media oriented, etc. While hard-core architectures offer excellent packaging and communication advantages, a soft vector core on FPGA offers the advantage of flexibility and lower part costs. A soft vector architecture is very efficient for HPC applications, because it can be scaled depending upon the required performance and available FPGA resources. Therefore, the number of FPGA based soft vector processors have been proposed [30; 31]. A soft vector unit typically comprises a parameterized number of vector lanes, a vector register file, a vector memory unit and a crossbar network that shuffles vector operands.

Typically, the vector processor is attached to the cache memory that manages data access instructions. In addition, the vector processors support a wide range of vector memory instructions that can describe different memory access patterns. To access strided and indexed memory patterns the vector processor needs a memory controller that transfers data with high bandwidth. The conventional vector memory unit incurs in

[1] Chapter 5 is based on the publications : [39] and [38].

delays while transferring data to the vector processor from local memory using a complex crossbar and bringing data into the local memory by reading from DDR SDRAM. To get maximum performance and to maintain the parallelism of HPC applications on vector processors, an efficient memory controller is required that improves the on/off-chip bandwidth and feeds complex data patterns to processing elements by hiding the latency of DDR SDRAM.

In this chapter, we used PMC in vector architecture called programmable vector memory controller (PVMC). The PVMC efficiently accesses complex memory patterns using a variety of memory access instructions. The PVMC manages memory access patterns in hardware using *Address Manager* and *Data Manager* thus improves the system performance by prefetching complex access patterns in parallel with computation and by transferring them to the vector processor without using a complex crossbar network. This allows a PVMC-based vector system to operate at higher clock frequencies. The PVMC includes a *Scratchpad Memory* unit that holds complex patterns and efficiently accesses, reuses, aligns and feeds data to a vector processor. The PVMC *Scheduler* supports multiple data buses that increase the local memory bandwidth and reduce on-chip bus switching. The design uses a *Multi DRAM Access Unit* that manages memory accesses of multiple *SDRAM module*s.

We integrated the proposed system with an open source soft vector processor, VESPA [30] and used an Altera Stratix IV 230 FPGA device. We compare the performance of the system with vector and scalar processors without PVMC. When compared with the baseline vector system, the results show that the PVMC system transfers data sets up to 2.2x to 14.9x faster, achieves between 2.16x to 3.18x of speedup for 5 applications and consumes 2.56 to 4.04 times less energy.

5.1 Vector Processor

The structure of a vector processor is shown in Figure 5.1. Modern vector memory units use local memories (cache or scratchpad) [31] and transfer data between the main memory and the VLs. The vector system has memory instructions for describing consecutive, strided, and indexed memory access patterns. The index memory patterns can be used to perform scatter/gather operations. A scalar *SSP* core is used to initialize the control registers that hold parameters of vector memory instructions such as the

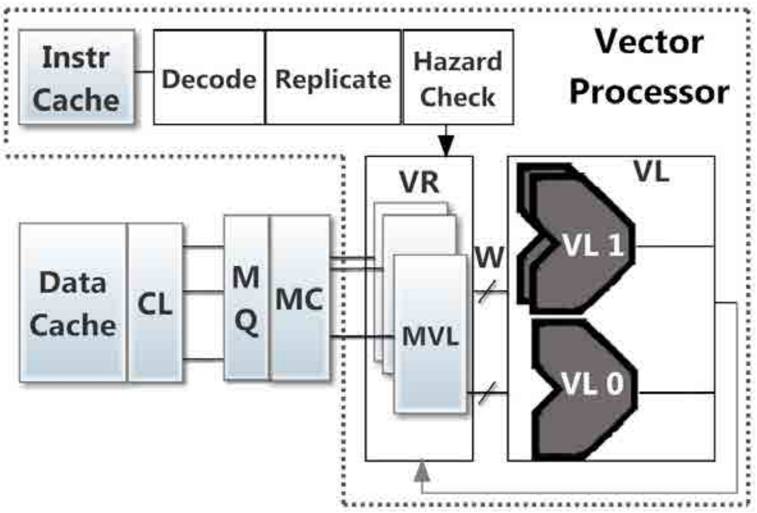


Figure 5.1: Generic Vector Processor

base address or the stride. The memory crossbar (MC) is used to route each byte of the cache line (CL) accessed simultaneously to any lane. The vector memory unit can take requests from each VL and transfers one CL at a time. Several MCs can be used to process memory requests concurrently. The memory unit of the vector system first computes and loads the requested address in the Memory Queue (MQ) for each lane and then transfers the data to the lanes. If the number of switches in the MC is smaller than the number of lanes, this process will take several cycles. Vector chaining [59] sends the output of a vector instruction to a dependent vector instruction, bypassing the vector register file, thus avoiding serialization, thus allowing multiple dependent vector instructions to execute simultaneously. Vector chaining can be combined with increasing the number of VLs. It requires available functional units; having a large MVL improves the impact on performance of vector chaining. When the loop is vectorized, and the original loop count is larger than the MVL, a technique called strip-mining is applied [60]. The body of the strip-mined vectorized loop operates on blocks of MVL elements.

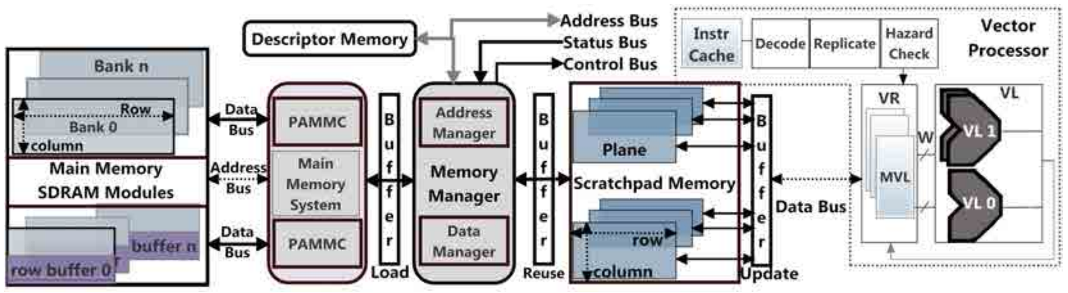


Figure 5.2: PVMC: Vector System

5.2 Programmable Vector Memory Controller

The Programmable Vector Memory Controller (PVMC) architecture is shown on Figure 5.2, including the interconnection with the vector lanes and the main memory. PVMC is divided into the *Bus System*, the *Memory Hierarchy*, the *Memory Manager* and the *PAMMC*. The *Bus System* transfers control information, address and data between processing and memory components. The *Data Bus* is used to transmit data to/from *SDRAM module*s. To minimize the data access latency the PVMC scales data bus bandwidth by using multiple data buses, with respect to the performance of the vector core and the capacity of the memory module (SDRAMs). The *required* and *available* bandwidths of the vector system are calculated by using the formulas 5.1 and 5.2. Vector$_{clock}$, Vector$_{Lanes}$ and Lane$_{width}$ define the vector system clock, the number of lanes and the width of each lane respectively. SDRAM$_{number}$, Controller$_{clock}$ and SDRAM$_{bus\ width}$ represent the number of separate *SDRAM module*s, the clock speed of each SDRAM controller and the data width of SDRAM controller respectively. To reduce the impact of the memory wall, PVMC uses a separate *data bus* for each *SDRAM module* and local memory (i.e. *Scratchpad Memory* , see Section 2.1.2.3). To improve bus performance, the *bus clock* can be increased up to a certain extent. The address bus is shared between multiple *SDRAM module*s. The chip select signal is used to enable a specific *SDRAM module*. The control, status and address buses are connected between PVMC and *SDRAM module*s.

The *Memory Hierarchy* includes the *Descriptor Memory*, the *Buffer memory*, the *Scratchpad Memory*, and the *Main memory*. The *Descriptor Memory* is used to hold data transfer information while the rest keep data. Depending upon the data transfer the *Address Manager* takes single or multiple instructions from the *Descriptor Memory* and transfers a complex data set to/from the *Scratchpad Memory* and *Main Memory*. The *Data Manager* is used to rearrange the output data of vector lanes for reuse or update. The data memory uses the *reuse*, *update* and *load buffers* (shown in Figure 5.3) to load, rearrange and write vector data. When input and output vectors are not aligned the data manager rearranges/shuffles data between lanes. In the case of strip mining, the data manager reduces the loop overhead by accessing the incremented data and reuses previous data when possible. For example, if the increment is equal to 1 the data manager shifts one data element and requests one element to load from the main memory. The incremented address is managed by the address and data managers that align vector data if required. The *Buffer Memory* architecture implements the following features:

- It aligns data when input and output vector elements are not the same.
- It handles the increment of the base address, thus reducing loop overhead when applying strip-mining.
- This is used to implement vector chaining from/to vector memory instructions.

Double-buffering can be used to overlap almost completely computation and memory transfer for each read and write *Scratchpad Memory*. In the case of double-buffering, the PVMC prefetches data from main memory into the *Scratchpad Memory* without interrupting the data path of the vector lanes. In the meantime, vector lanes keep working on data from the *Scratchpad Memory* that has already been accessed. The *PAMMC* reads/writes data from/to multiple *SDRAM module*s of *Main Memory*. There is a chip-select network that connects the *PAMMC* to each *SDRAM module*. Each bit of chip select operates a separate *SDRAM module*. *PAMMC* can integrate multiple *PAMMC*s using separate data buses, which increases the memory bandwidth.

$$Required_{Bandwidth} = Vector_{clock} \times Vector_{Lanes} \times Lane_{width} \tag{5.1}$$

$$Available_{Bandwidth} = SDRAM_{number} \times Controller_{clock} \times SDRAM_{bus\ width} \tag{5.2}$$

5.3 PVMC Functionality

In this section, we discuss the important challenges faced by the memory unit of soft vector processors and explain our solution.

5.3.1 Memory Hierarchy

The proposed soft vector (such as VESPA [30]) core system uses the cache hierarchy to improve the data locality by providing and reusing the required data set to functional units. With a high number of vector lanes, the vector memory unit does not satisfy the data spatial locality. PVMC improves the data spatial locality by accessing more data elements than MVL into its *SSM* and later transferring them using the *buffer memory*. Non-unit stride accesses do not exploit spatial locality offered by cache resulting in considerable waste of resources. PVMC manages non-unit stride memory accesses similar to unit-stride. Like a cache of soft vector processor, the PVMC *SSM* temporarily holds data to speed up later accesses. Unlike a cache, data is deliberately placed in the *SSM* at a known location, rather than automatically cached according to a fixed hardware policy. The PVMC *memory manager* along with the *Buffer Memory* hold information of unit and non-unit strided accesses, update and reuse it for future accesses.

To load and store data in a conventional vector system, the vector register file is connected to the data cache through separate read and write crossbars. When the input to the vector lanes is mismatched the vector processor needs an extra instruction that

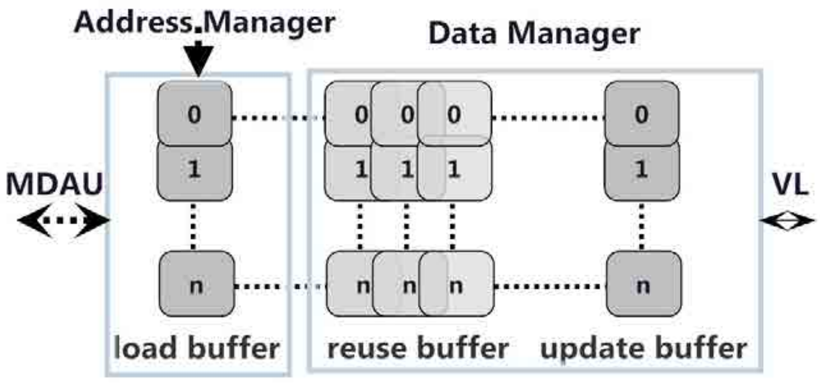


Figure 5.3: Data Memory Buffers: Load, Reuse & Update

aligns the vector data. The PVMC uses the *Buffer Memory* to transfer data to the vector register file which is simpler than using crossbar and data alignment. The *Buffer Memory* aligns data when input and output vector elements are not the same. It also reuses and updates existing vector data and loads data which is not present in the *SSM*.

5.3.2 Address Registers

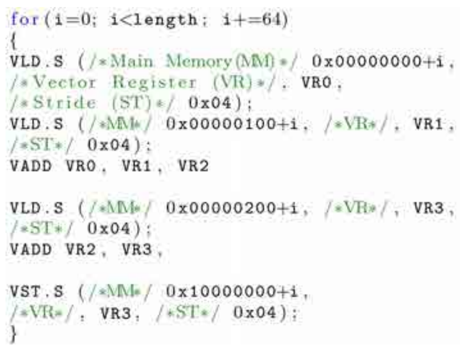
The vector processor uses address registers to access data from *main memory*. The memory unit uses address registers to compute the effective address of the operand in *main memory*. A conventional vector processor supports unit-stride, strided, and indexed accesses. In our current evaluation environment, the PVMC system uses a separate register file to program the *Descriptor Memory* using data transfer instructions to comply with the MIPS ISA. The PVMC *Descriptor Memory* can perform accesses longer than the MVL without modifying the instruction set architecture. PVMC uses a single or multiple descriptors to transfer various complex non-stride accesses.

5.3.3 Main Memory Controller

The conventional Main Memory Controller (MMC) uses a direct memory access (DMA) or Load/Store unit to transfer data between main memory and cache memory. Thus, the vector memory unit uses a single DMA request to transfer one unit-stride access between main memory and a cache line. But for complex or non-unit strided accesses the memory unit uses multiple DMA or Load/Store requests which require extra time to initialize addresses, synchronise on-chip buses and SDRAMs. The PVMC MMC uses a single descriptor for unit and non-unit stride accesses which improve the memory bandwidth by transferring descriptors to the memory controllers, rather than individual references and by accessing data from multi-SDRAM devices.

5.3.4 Programming Vector Accesses

Figures 5.4 (a) and (b) show vector loops (with MVL of 64) for a conventional vector architecture and the PVMC, including the PVMC memory transfer instructions respectively. The `VLD.S` instruction transfers data with the specified stride from main memory to vector registers using cache memory. For long vector accesses and a high

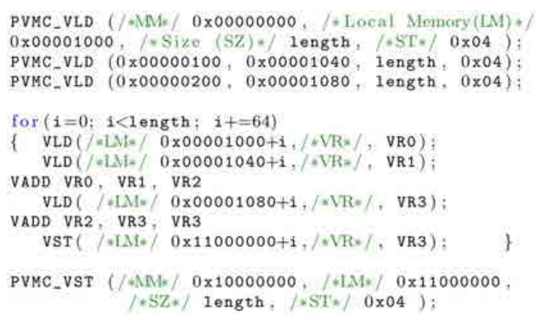


```
for(i=0; i<length; i+=64)
{
VLD.S (/*Main Memory(MM)*/ 0x00000000+i,
/*Vector Register (VR)*/, VR0,
/*Stride (ST)*/ 0x04);
VLD.S (/*MM*/ 0x00000100+i, /*VR*/, VR1,
/*ST*/ 0x04);
VADD VR0, VR1, VR2

VLD.S (/*MM*/ 0x00000200+i, /*VR*/, VR3,
/*ST*/ 0x04);
VADD VR2, VR3,

VST.S (/*MM*/ 0x10000000+i,
/*VR*/, VR3, /*ST*/ 0x04);
}
```

(a)



```
PVMC_VLD (/*MM*/ 0x00000000, /*Local Memory(LM)*/
0x00001000, /*Size (SZ)*/ length, /*ST*/ 0x04 );
PVMC_VLD (0x00000100, 0x00001040, length, 0x04);
PVMC_VLD (0x00000200, 0x00001080, length, 0x04);

for(i=0; i<length; i+=64)
{   VLD(/*LM*/ 0x00001000+i,/*VR*/, VR0);
    VLD(/*LM*/ 0x00001040+i,/*VR*/, VR1);
VADD VR0, VR1, VR2
    VLD( /*LM*/ 0x00001080+i,/*VR*/, VR3);
VADD VR2, VR3, VR3
    VST( /*LM*/ 0x11000000+i,/*VR*/, VR3);      }

PVMC_VST (/*MM*/ 0x10000000, /*LM*/ 0x11000000,
            /*SZ*/ length, /*ST*/ 0x04 );
```

(b)

Figure 5.4: (a) Vector Loop (b) PVMC Vector Loop

number of vector lanes, the memory unit generates delay when data transfers do not fit in a cache line. This also requires complex crossbars and efficient prefetching support. Delay and power increase for complex non-stride accesses and crossbars. The PVMC_VLD instruction uses a single or multiple descriptors to transfer data from the *main memory* to the *SSM*. PVMC rearranges and manages accessed data in the *Buffer Memory* and transfers it to vector registers. In Figure 5.5, PVMC prefetches vectors longer than MVL in the *SSM*. After completing the first transfer of MVL, the PVMC sends a signal to the vector processor that acknowledges that the register is available for processing. In this way PVMC pipelines the data transfers and parallelizes computation, address management and data transfers.

A common concern, when using soft vector processors, is compiler support. A soft core vector processor typically requires in-line assembly code that translates vector instructions with a modified GNU assembler. In order to describe how PVMC is used, the supported memory access patterns are discussed in this section. We provide

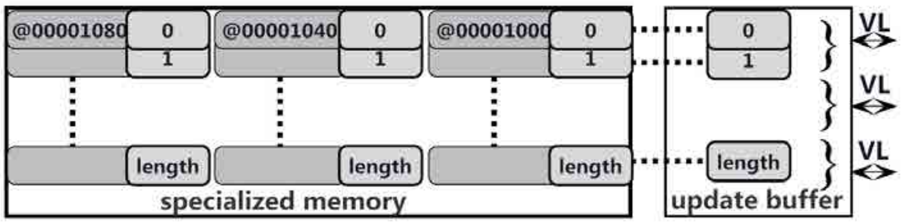


Figure 5.5: PVMC Data Transfer Example

C macros which ease the programming of common access patterns through a set of function calls, integrated with an API. The memory access information is included in the PVMC header file and provides function calls (e.g. 3D_STEN(), 3D_TILE(), etc.) that require basic information of the local memory and the data set. The programmer has to annotate the code using PVMC function calls. The function calls are used to transfer the complete data set between *main memory* and *SSM*. PVMC supports complex data access patterns such as strided vector accesses and transfers complex data patterns in parallel with vector execution. For multiple or complex vector accesses, PVMC prefetches data using function calls (e.g. 3D-Stencil, etc.), arranges them according to the predefined patterns and buffers them in the *SSM*. The PVMC memory manager efficiently transfers data with long strides, longer than MVL size and feeds it to the vector processor.

5.4 Experimental Framework

In this section, we describe the PVMC and VESPA vector systems as well as the Nios scalar system. The Altera Quartus II version 13.0 and the Nios II Integrated Development Environment (IDE) are used to develop the systems. The systems are tested on an Altera Stratix-IV FPGA based DE4 board. The section is further divided into three subsections: the *VESPA system*, the *PVMC system* and the *Nios system*.

5.4.1 The VESPA System

The FPGA based vector system is shown in Figure 5.6 (a). The system architecture is further divided into the *Scalar core*, the *Vector core* and *Memory System*.

5.4.1.1 SSP

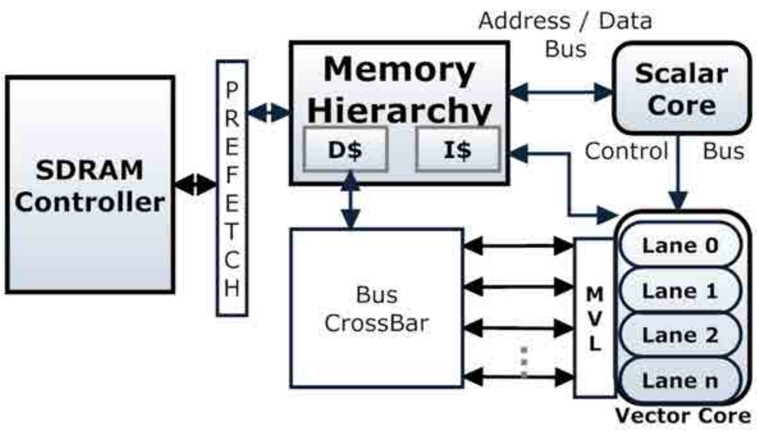
An SPREE [22] scalar processor is used to program the VESPA system and perform scalar operations. The SPREE is a 3-stage MIPS pipeline with full forwarding core and has a 4K-bit branch history table for branch prediction. The SPREE core keeps working in parallel with the vector processor with the exception of control instructions and scalar load/store instructions between the two cores.

5.4.1.2 Vector Processor

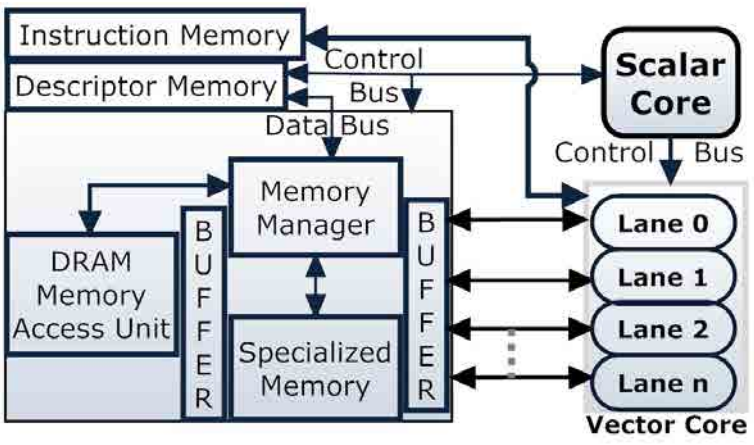
A soft vector processor called VESPA (Vector Extended Soft Processor Architecture) [30] is used in the design. VESPA is a parameterizable design enabling a large design space of possible vector processor configurations. The vector core uses a maximum vector length (MVL) of 128.

5.4.1.3 Memory System

The baseline VESPA vector memory unit (shown in Figure 5.6 (a)) includes an SDRAM controller, cache and bus crossbar units. The SDRAM controller transfers data from



(a)



(b)

Figure 5.6: (a) Baseline VESPA System (b) PVMC System

Table 5.1: Brief description of application kernels

Application	FIR	1D Filter	Tri-Diagonal	Matrix Multiplication	Gaussian	Motion Estimation	3D-Stencil
Access Pattern	Stream	1D Block	Diagonal	Row & Column	2D Block	2D Block	3D-Tiling

main memory (*SDRAM modules*) to the local cache memory. The Vector core can access only one cache line at a time that is determined by the requesting lane with the lowest lane identification number. Each byte in the accessed cache line can be simultaneously routed to any lane through the bus crossbar. Two crossbars are used, one read crossbar and one write crossbar.

5.4.2 The Proposed PVMC System

The PVMC based vector system is described in Sections 5.1 and 5.2 and shown in Figure 5.6 (b). The major difference between the PVMC and VESPA systems is the memory system. The PVMC system manages on-chip data and off-chip data movement using the *Buffer Memory* and the *Descriptor Memory*. The memory crossbar of VESPA is replaced in PVMC with the *Buffer Memory* which rearranges and transfers data to the vector lanes. The *Scratchpad Memory* is used instead of a cache memory.

5.4.3 The Baseline Nios System

The Nios II *SSP* [21] is a 32-bit embedded-processor architecture designed specifically for the Altera family of FPGAs. The Nios II architecture is an RISC soft-core architecture which is implemented entirely in the programmable logic and memory blocks of Altera FPGAs. Two types of systems having different Nios cores are used; the Nios II/e and the Nios II/f. The Nios II/e system is used to achieve the smallest possible design consuming less FPGA logic and memory resources. The core does not support caches and saves logic by allowing only one instruction to be in-flight at any given time which eliminates the need for data forwarding and branch prediction logic. The Nios II/f system has a fast Nios processor for high performance that implements a barrel shifter with hardware multipliers, branch prediction and 32Kbyte Data and

Instruction caches. An Altera Scatter-Gather DMA (SD-DMA) along with SDRAM controller is used that handles multiple data transfers efficiently.

5.4.4 Applications

Table 5.1 shows the application kernels which are executed on the vector systems along with their memory access patterns. The set of applications covers a wide range of patterns allowing us to measure the behaviour and performance of data management and data transfer of the systems in a variety of scenarios.

5.5 Results and Discussion

In this section, the resources used by the memory and bus systems, the application performance, the dynamic power and energy and the memory bandwidth of the PVMC vector system are compared with the results of the non-PVMC vector system and the baseline scalar systems.

5.5.1 Memory & Bus System

Multiple memory hierarchies and different bus system configurations of PVMC & VESPA systems are compiled using Quartus II to measure their resource usage, maximum operating frequency and leakage power.

Table 5.2 (a) presents the maximum frequency of the memory system for 1 to 64 vector lanes with 32kB of cache/scratchpad memory. The VESPA system uses crossbars to connect each byte of the cache line to the vector lanes. Increasing the number

Vector Lanes	1	2	4	16	32	64
VESPA fmax	142	130	125	115	114	110
PVMC fmax	195	187	187	185	182	180

(a)

Sys Bus	1 Layer	2 Layer
VESPA	157	-
PVMC	292	282

(b)

Table 5.2: (a) Local Bus Maximum Frequency (MHz) (b) Global Bus Maximum Frequency (MHz)

Table 5.3: Resource Utilization of the Memory Hierarchy

	Local Memory 32KB			Main Memory		Leakage
	Line Size	Reg, LUT	Memory Bits	Controller	Reg, LUTs	Power
VESPA	128	1489, 3465	304400	1	2271, 1366	1.02
	256	1499, 5529	305632	1	2271, 1366	1.15
PVMC	128	90, 1030	613134	1	1742, 1249	0.70
	256	108, 1047	615228	2	3342, 2449	0.80

of lanes requires more crossbars and a larger multiplexer that routes data between vector lanes and cache lines. This decreases the operating frequency of the system. For the VESPA vector processor, results show that by increasing the number of vector lanes from 1 to 64 requires larger crossbar multiplexer switches and operates at lower frequency. The PVMC *Scratchpad Memory* uses separate read and write scratchpad memories that reduce the switching bottleneck. The vector lanes read data from read *Scratchpad Memory* for processing and transfer it back to the write *Scratchpad Memory* . The on-chip data alignment and management is done by the *Data Manager* and the *buffer memory*. This direct coupling of the *Scratchpad Memory* and vector lanes using the *update buffer* is very efficient and allows the system to operate at a higher clock frequency. Table 5.2 (b) presents the maximum frequency for the data bus to operate multiple memory controllers. The PVMC data bus supports a dedicated bus for each SDRAM controller which increases the bandwidth of the system. The data bus of VESPA system supports only a single SDRAM controller.

Table 5.3 shows the resource utilization of the memory hierarchy of the VESPA and PVMC systems. The memory hierarchy is compiled for 64 lanes with 32KB of memory and several line sizes. Column *Line Size* presents cache line and *update buffer* size in bytes of the VESPA and PVMC systems respectively. The VESPA system cache memory uses cache lines to transfer each byte to the vector lanes. The PVMC *update buffer* is managed by the *data manager* and is used to transfer data to the vector lanes. Column *Reg, LUT* shows the resources used by the cache controller and the memory manager of the VESPA and PVMC systems respectively. Column *Memory Bits* presents the number of BRAM bits for the local memory. The PVMC memory system uses separate read and write scratchpad memories, and, therefore, it occupies twice

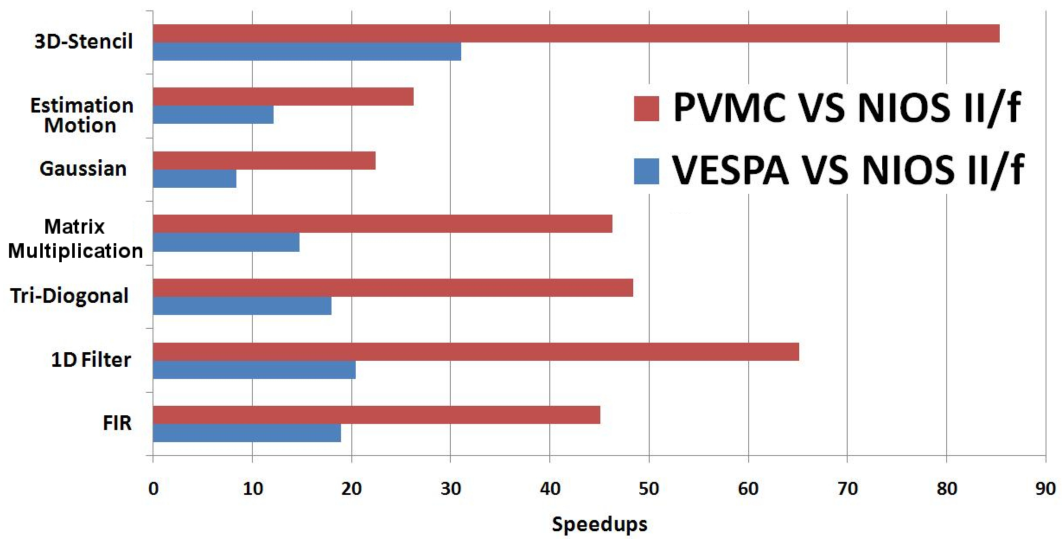


Figure 5.7: Speedup of PVMC and VESPA over Nios II/f

the number of BRAM bits. The data manager of the PVMC memory system occupies 3 to 5 times less resources than the VESPA memory system. Column *Main Memory* presents the resource utilization of the SDRAM controllers. The VESPA system does not support dual SDRAM controllers. Column *Power* shows leakage power in watts for the VESPA and PVMC memory systems. The leakage current of the VESPA system is higher than in PVMC, because it requires a complex crossbar network to transfer data between the cache and the vector lanes and requires more multiplexers.

5.5.2 Performance Comparison

For performance comparisons, we use the applications of Table 5.1. We run the applications on the Nios II/e, Nios II/f and VESPA systems and compare their performance with the proposed PVMC vector system. Nios II/e, VESPA and PVMC systems run at 100 MHZ. The VESPA and the PVMC systems are compiled using 64 lanes with 32kB of cache and *SSM* respectively. The Nios II/f system operates at 200 Mhz using data and instruction caches of 32KB each. All systems use a single SDRAM controller to access the main memory.

Figure 5.7 shows the speedups of VESPA and PVMC systems over Nios II/f. Results show that vector execution with the PVMC is 8.3x and 31.04x faster than the Nios II/f. Results for Nios II/e are not shown in Figure 5.7). When compared with the Nios

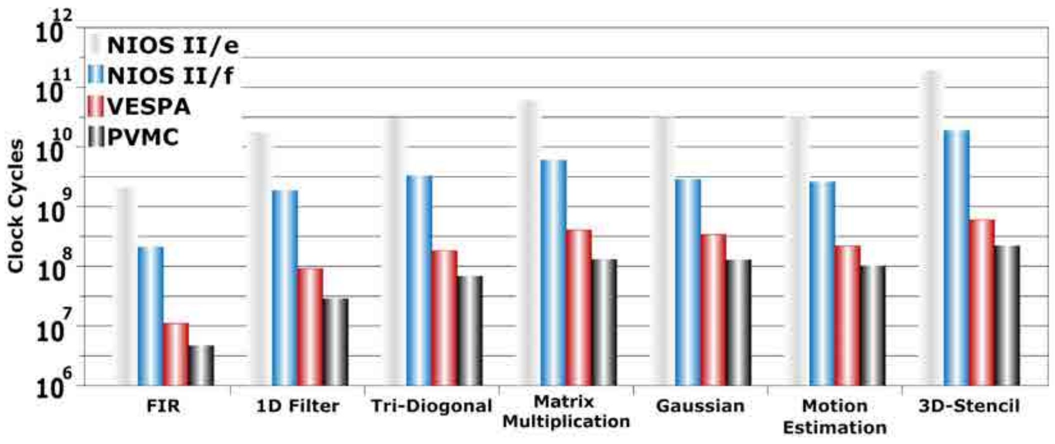


Figure 5.8: Vector & Scalar Systems: Application Kernels Execution Clocks

II/e, the PVMC improves speed between 90x and 313x which shows the potential of vector accelerators for high performance.

In order to discard that the speed ups over the scalar processor NIOS are caused by using SPREE as the scalar unit of the vector processor, we execute FIR, Matrix Multiplication and 3D-Stencil application kernels on a SPREE scalar processor, i.e. with the vector processor disabled. While comparing performance of FIR, Matrix Multiplication and 3D-Stencil kernels on SPREE, Nios II/e and Nios II/f scalar processors, the results show that SPREE improves speed between 5.2x and 8.6x over Nios II/e, whereas against Nios II/f the SPREE is not efficient. The Nios II/f achieves speedups between 1.27x and 1.67x over SPREE scalar processor. The results show that Nios II/f performs better than Nios II/e and SPREE scalar processors.

By using the PVMC system, the results show (Figure 5.8) that the FIR kernel achieves 2.37x of speedup over VESPA. The application kernel has streaming data accesses and requires a single descriptor to access a stream that reduces the address generation/management time and on-chip request/grant time. The 1D Filter accesses a 1D block of data and achieves 3.18x of speedup. The Tri-diagonal kernel processes the matrix with sparse data placed in diagonal format. The application kernel has a diagonal access pattern and attains 2.68x of speedup. The Matrix Multiplication kernel accesses row and column vectors. PVMC uses two descriptors to access the two vectors. The row vector descriptor has unit stride whereas the column vector has a stride equal to the size of a row. The application yields 3.13x of speedup. The Motion Estimation and Gaussian applications take 2D block of data and achieve 2.67x and 2.16x of speedup respectively. The PVMC system manages addresses of row and column vectors in hardware. The 3D-Stencil data uses row, column and plane vectors and achieves 2.7x of speedup. The vectorized 3D-stencil code for VESPA always uses the whole MVL and unit-stride accesses and accesses vector data by using vector address registers and vector load/store operations. The VESPA system multi-banking methodology requires a larger crossbar that routes requests from load/store units to cache banks and another one from banks back to ports. This also increases the cache access time but reduces the simultaneous read and write conflicts.

5.5.3 Dynamic Power & Energy

To measure voltage and current the DE4 board provides a resistor to sense current/voltage and 8-channel differential 24-bit analogue to digital converters. Table 5.4 presents dynamic power and energy of different systems using a filter application kernel with

System @MHz	Lanes	Reg, LUTs	Dynamic Power and Energy			
			FPGA Core	SDRAM	Total	Energy
Nios II/e @100		7034 , 7986	1.47	1.76	3.23	581.17
Nios II/e @200		8612 , 8076	1.65	2.26	3.91	342.46
Nios ll/f @100		9744 , 10126	2.086	1.686	3.760	82.56
Nios ll/f @200		12272 , 10256	3.109	2.513	5.822	48.379
VESPA @100	1	7227 , 7878	1.54	2.24	3.78	101.99
	4	7867 , 12193	1.874	2.24	4.17	60.04
	16	10090 , 31081	3.191	2.24	5.353	14.36
	32	13273 , 57878	4.666	2.24	6.29	9.42
	64	19641 , 103857	5.57	2.25	7.78	7.026
PVMC @100	1	5227 , 5587	1.1	2.11	3.23	39.85
	4	5856 , 6193	1.30	2.11	3.51	20.08
	16	8817 , 21261	1.91	2.11	4.32	4.16
	32	10561 , 45658	2.86	2.11	4.97	2.54
	64	15564 , 88,934	4.01	2.11	6.13	1.75

Table 5.4: Systems: Resource, Power and Energy utilization

2M Byte of input data set, 1D block (64 elements) of data access and 127 arithmetic operations on each block of data. Column System@MHz shows the operating frequency of the Nios II/e and Nios II/f cores and the VESPA and PVMC systems. The vector cores execute the application kernel using different numbers of lanes while the clock frequency is fixed to 100 MHz. To control the clock frequencies all systems use a single phase-locked loop (PLL). Columns *Reg, LUTs* and *Mem Bits* show the amount of logic and memory in bits respectively utilized by each system. The Nios II/e does not have a cache memory and only uses program memory. Column *Dynamic Power and Energy* presents run time measured power of scalar and vector systems while executing the filter application kernel and calculated energy for power and execution time. Column FPGA Core includes the power consumed by on-chip FPGA resources and PLL power. Column SDRAM power presents the power of the SDRAM memory device. The power of Nios II/e and Nios II/f increases with frequency. Results show that the PVMC draws 21.2% less power and 4.04x less energy than the VESPA system, both using 64 lanes. For a single lane configuration, PVMC consumes 14.55% less power and 2.56x less energy. This shows that PVMC improves system performance and handles data more efficiently results improve with a higher number of lanes. The PVMC using a single lane and operating at 100 MHz draws 14%, 44% less power and 14.5x, 8.5x less energy than a Nios II/f core operating at 100 MHz and 200 MHz respectively. Whereas, when compared to a Nios II/e core at 100 MHz and 200 MHz, the PVMC system draws .03% and 17.3% less power respectively and consumes and 2.07x, 1.21x times less energy.

5.5.4 Bandwidth

In this section, we measure the bandwidth of the PVMC, VESPA and Nios ll/f systems by reading and writing complex memory patterns. The PVMC with a single SDRAM controller is also executed on a Xilinx Virtex-5 ML505 FPGA board, and results are very similar. The processing cores have 32 bit on-chip data bus operating at 100 MHZ that provides a maximum bandwidth of 400 MB. The PVMC can achieve maximum bandwidth by using data transfer size equal to the data set. In order to check the effects of memory, bus and address management units over the system bandwidth, we transfer data between processor and memory using different pattern and transfer sizes. The

X-axis presents random load/store, streaming, a 2D and 3D tiled data sets of 2MB that are read and written from/to the main memory. The load/store access patterns read/write 4B from a random location. A single streaming access pattern accesses 1KB of stream and a 2D access pattern reads/writes a 2D block with row and column size of 1KB. The 3D tiled benchmark reads a 3D tile of 128x128x128 bytes (rows, column and plane) and writes it back to the main memory. The Y-axis shows the bandwidth in MB per second for single and double *SDRAM Controller*(s). Figure 5.9 shows a bar chart of different data transfers for the PVMC, VESPA and Nios ll/f - based systems. While using single and double *SDRAM Controller*(s), the results show that PVMC random load/store data transfers are 2.2x, 3.4x and 3.5x, 2.3x times faster than VESPA and Nios ll/f systems respectively. The load/store data transfers require large control information (e.g. bus and memory initialization, etc.) that limits the bandwidth. The data transfer is further improved up to 4.9x, 9.95x and 4.5x, 4.8x times while transferring streaming data. While transferring 2D tiled data, the PVMC achieves 8.8x, 14.9x and 8.2x, 8.94x of speedup. For complex data transfers, PVMC improves bandwidth 10.3, 9.8 and 16.4, 12 times. The VESPA system uses a single data bus to transfer to/from main memory; therefore, it is unable to get the benefit from double *SDRAM Controller*s. The Nios ll/f system uses an SG-DMA controller that transfers data using unit-stride and forces to follow bus protocol. The PVMC system

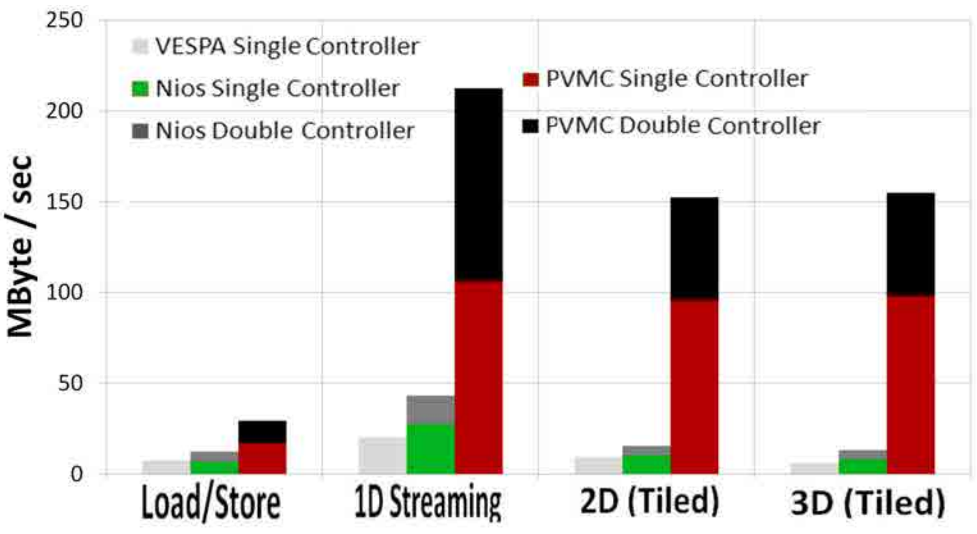


Figure 5.9: Vector & Scalar Systems: Memory Bandwidth

has a dedicated data bus for each *SDRAM controller* therefore, it efficiently accesses data from single- or multi- main memories and manages multi-*SDRAM Controller*(s) without support of a microprocessor. The PVMC complex patterns use few descriptors that reduce run-time address generation and address request/grant delay. For 3D tiled data transfer, PVMC improves bandwidth by managing addresses at compile-time and by accessing data from multi-DRAM devices and multiple banks in parallel.

5.6 Conclusion

The memory unit can easily become a bottleneck for vector accelerators. In this chapter, we suggested PMC for vector processor architectures called PVMC, that manages memory accesses without the support of a scalar processor and complex crossbar network. A *Scratchpad Memory* and a *Data Manager* are integrated that efficiently access, reuse, align and feed data to the vector processor. A *Multi DRAM Access Unit* is used to improve the main memory bandwidth which manages the memory accesses of multiple *SDRAM*s. We implemented and validated the proposed PVMC system on an Altera DE4 Development board with 40 nm Stratix IV EP4SGX230 FPGA family. We compare the performance of our proposal with a vector system without PVMC as well as a scalar only system. The experimental evaluation based on the VESPA vector system with conventional cache memory system, demonstrates that the PVMC based approach improves the utilization of hardware resources and efficiently accesses main memory data. The benchmarking results show that PVMC achieves between 2.16x to 3.18x of speedup for 5 applications, consumes 2.56 to 4.04 times less energy and transfers different data patterns up to 2.2x and 14.9x faster than the baseline vector system.

Part III

Multi-core Memory System for Regular Data Pattern

6

PMSS: A Programmable Memory System and Scheduler for Complex Memory Patterns

Deep research has been conducted to improve the performance of HPC systems. One way to improve the performance is to build a multi-ASHA/core system [61; 62], manage/schedule [63; 64] its hardware and memory [65; 66] resources efficiently and write parallel code [67; 68] to execute on the system. A task-based programming model [69; 70] is useful and convenient for such architectures, as it identifies the tasks in software that can be executed concurrently.

The effects of the memory wall can be observed in the multi-processor systems having transaction-based workloads [71], where processors remain stall during 80% of the time [9; 35] (Figure 6.1). The load and store stall time include delay while reading and writing data to external memories. The stall time is mainly dependent on the resource scheduling, Input/output synchronization, and memory accesses. Figure 6.1 shows that a major portion of system time is consumed by memory load/store accesses. Efficient management of the system memory recourses can improve performance of the overall system. Integrating multi-core platforms while using low clock frequencies can balance the ratio between clock speed of processor and memory. However, the cumulative memory bandwidth requirement of all processing elements is still increasing, adding a

[1] Chapter 6 is based on the publications : [9] and [40].

new dimension to the problem. Integrating more memory controllers [72; 73] on the system platform can increase bandwidth, but it has drawbacks that are as follow.

- An SDRAM controller is an expensive component both in terms of area and power dissipation.
- A higher number of Input/output pins is required.

Modern platforms often contain multiple accelerators and memory controllers, to provide a good balance between performance, cost, power consumption and flexibility. Hence, multiple memory controllers are often not an option, emphasizing to use the existing SDRAM bandwidth as efficiently as possible. To improve performance of the system we implemented a PMC based memory controller in hardware called Programmable Memory System and Scheduler (PMSS) which handles multiple *ASHA*s. The PMSS *Scheduler* applies *Symmetric* and *Asymmetric* scheduling polices (Section 2.1.3.3) on multi-ASHAs. The PMSS arranges data/address in hardware and schedules computation tasks of multi-ASHAs without intervention of processor (Master) core and operating system. We further parameterize *Scratchpad Memory* for multiple *ASHA*s to allow the designer to more powerfully trade area for performance scaling for data parallel applications. By combining all the functions into one chip, the system becomes faster and less power consuming. The support of standard C/C++ language for high level data access pattern decreases the complexity of writing applications.

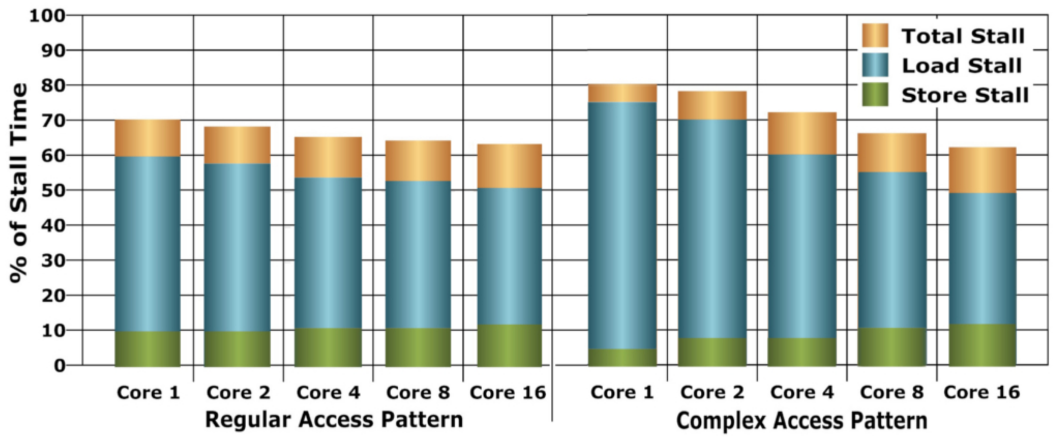


Figure 6.1: System Stall Time

These calls support specialized *ASHA* IPs which are integrated with the system. Some salient features of the proposed PMSS architecture are included below:

- PMSS handles multiple *ASHA* cores using event driven handshaking methodology without support of the master core or OS.
- The controller gathers multiple memory read/write requests of multi-ASHA system and maximizes utilization of SDRAM open banks. This removes the overhead from opening and closing rows as well as idle cycles on the data bus.
- PMSS *Scheduler* along with *Address Manager* improves performance of the system by efficiently prefetching and managing complex patterns.
- Due to the light weight of PMSS the system consumes less power.
- As a consequence, PMSS increases the effective bandwidth achieved by the system.

6.1 PMSS Architecture

Programmable Memory System and Scheduler (PMSS) is based on high level data patterns that simplify programming of HPC applications while ensuring high performance and efficiency. To present the functioning of PMSS, we first depict its inner architecture in Figure 6.2, which also briefly shows the interconnection with the external processing units. PMSS design supports multiple *ASHA*s using special event driven handshaking methodology. Each *ASHA* has local *Scratchpad Memory* to operate. Therefore, other data transfer requests cannot affect the execution process.

During compile-time an API is used to provide information, which separates, pipelines, overlaps and schedules memory read/write operations and generate executable PMSS binary file. At initialize-time (a), the PMSS uses *Program Line* to write PMSS binary file to the *Descriptor Memory* (Section 2.1.2.1). At run-time (b), the *Memory Manager* accesses memory patterns from *Descriptor Memory*. During step (c), the *Memory Manager* (Section 2.1.3) takes scheduling information from the *Scheduler* (Section 2.1.3.3) and prioritize memory accesses appropriately. The *Scheduler* keeps gathering regular data transfer memory requests from external sources and places them

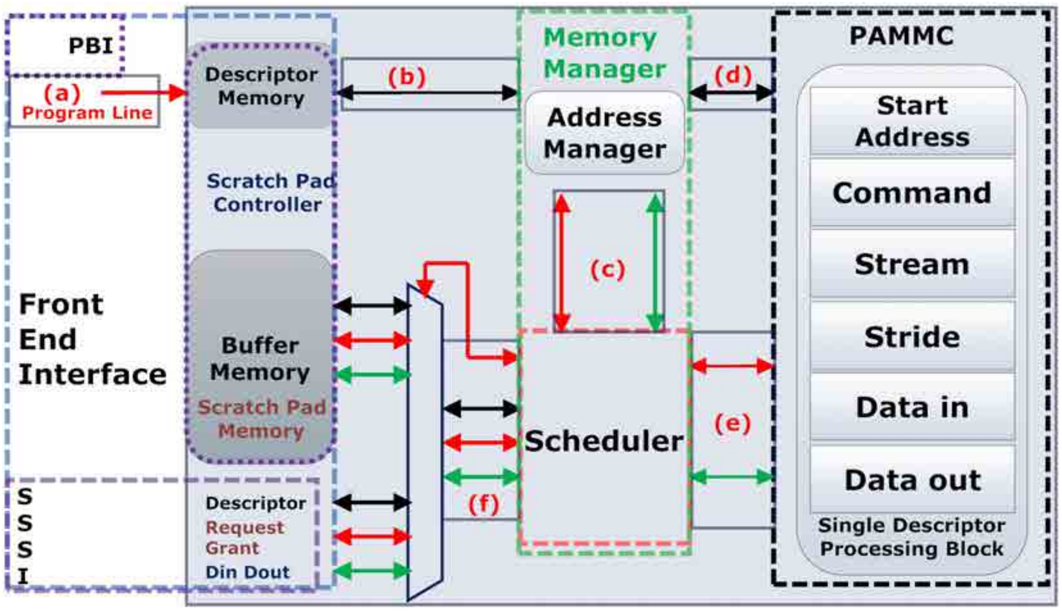


Figure 6.2: PMSS Architecture

in the local *Scratchpad Memory* buffer. The PMSS *Descriptor Memory* holds the information of multi-ASHA memory access patterns and scheduling methodology. The priorities are assigned by the programmer and at run-time PMSS applies symmetric and *Asymmetric Scheduling* polices (Section 2.1.3.3). During step (d and e), the *Pattern Aware Main Memory Controller* (PAMMC) (Section 2.1.4) takes a single descriptor from *Memory Manager* and read/write data to/from SDRAM memory. During step (f), the *Scheduler* provides a link to the external processing core to access *Data Memory*.

6.1.1 Memory Manager

The Memory management performs the key role in PMSS multi-ASHA system. It improves the command and data efficiency of the system by arranging/managing address/data signals. The *ASHA* address (scratchpad memory) and physical address (main memory) information is placed in PMSS *ASHA Descriptor Memory*. The memory space allocated to an *ASHA* as part of one request can be addressable through single or multiple descriptors. The Memory Manager loads blocks of data to the local *ASHA* buffer. Once the *ASHA* finishes processing, it writes back the processed data to main memory. The Memory Manager also manages run-time generated memory accesses using the *Descriptor Memory*. At run-time, the Memory Manager allocates a single descriptor block for each processing core. The Memory Manager takes memory requests from a processing core, buffers them and compares consecutive requests. If the addresses of consecutive memory requests have constant strides, the Memory Manager allocates a descriptor block by defining *stride* and *size* parameters. If the request has variable strides then, the *Memory Manager* uses the multiple descriptors that can access complex pattern. The memory hierarchy of PMSS includes the following features.

6.1.1.1 Scheduling

The PMSS *Scheduler* manages read, write and execute data transfer operations of multiple *ASHA* . PMSS supports two scheduling policies, *Symmetric* and *Asymmetric*, that execute *ASHA*s efficiently. In **Symmetric** multi-ASHA strategy, the PMSS scheduler manipulates the available *ASHA* 's request in FIFO (First In First Out). The *task buffer* (shown in Figure 2.11 (b)) manages the *ASHA* 's request in FIFO order. The **Asymmetric** strategy emphasizes on priority and incoming requests of the *ASHA*s. Like

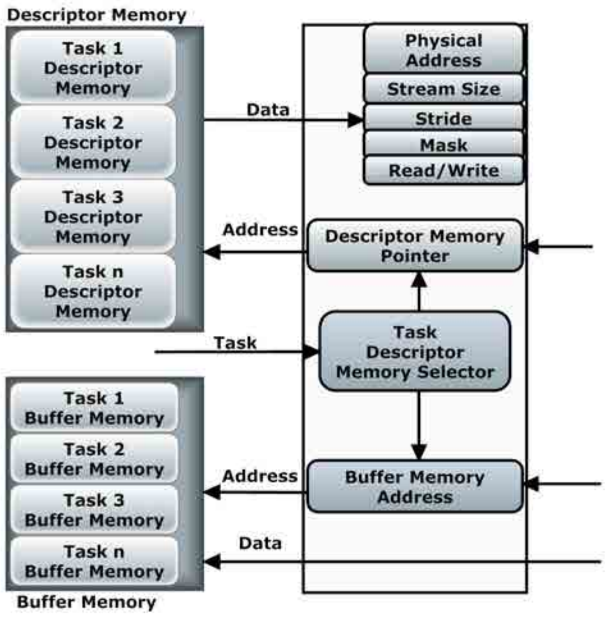


Figure 6.3: PMSS:Memory Manager

Xilinx Xilkernel [1] scheduling model, the PMSS scheduling policies are configured at program-time. Unlike Xilkernel, the PMSS executes requests in hardware at run-time. The number of priority levels can be configured for asymmetric scheduling.

6.1.1.2 Memory Organization

PMSS core holds information of *Main Memory* and local memory address space that is partitioned into patterns (Figure 6.3). The patterns are organized, in the same way, as the program is written. PMSS provides protection at the pattern level i.e. segment can be read/written by the *ASHA* for which it is allocated but not by others. Multiple noncontiguous accesses of memory lead to a high degree of read/write delay due to the control selection of SDRAM memory. PMSS overcomes this problem by organizing multiple noncontiguous memory accesses together.

6.1.1.3 Locality and Isolation

The PMSS provides isolation by allocating separate *Descriptor Memory* and scratchpad (local) buffers for each *ASHA* . This guarantees that no other *ASHA* can access the memory belonging to a given *ASHA*. PMSS keeps the knowledge of memory whether a certain memory area is in the *ASHA* 's local scratchpad. This knowledge allows the PMSS to manage the placement of memory as well as reusing and sharing already accessed memory. As the dataset access pattern and size description is known at program time, PMSS efficiently utilizes these access patterns at run time.

6.1.1.4 Programmability

The PMSS provides functions to allocate and map application kernel local memory buffer and physical dataset. A `PMSS_MEMCPY` instruction is created which reads/writes a block of data from the *Main Memory* to the *ASHA* 's local memory buffer. By managing read/write command PMSS reduces delay occurred by other commands that are already issued in the access cycle. PMSS gathers consecutive read/write commands that increase the burst length, as smaller bursts result in more activate and precharge commands that degrade the *command efficiency*.

[1]The Xilkernel is a small, robust, and modular kernel with the features of the operating system. It supports the embedded kernel, with a POSIX API.

6.2 Evaluation Architecture

In this section, we describe and evaluate the PMSS-based multi-ASHA system having Thresholding, Radian Converter, FIR, FFT, Matrix Multiplication, Smith Waterman, Laplacian solver, and 3D-Stencils application kernels. In order to evaluate the performance of the PMSS System, the results are compared with a similar system having MicroBlaze master core. The Xilinx Integrated Software Environment [74] and Xilinx Platform Studio [75] are used to design the systems. Xilinx Power Estimator [76] is used to analyze the system power. A Xilinx ML505 evaluation FPGA board [52] is used to test the multi-ASHA system.

6.2.1 MicroBlaze based Multi-ASHA System

A MicroBlaze based Multi-ASHA system is proposed (Figure 6.4). The MicroBlaze soft-core processor is used that controls the resources of the system. The Real-Time Operating Systems (RTOS) Xilkernel [77] is executed on the MicroBlaze soft processor. The Xilkernel has POSIX support and can declare threads at program time that start with the kernel. From the main, application is spawned as multiple parallel threads using the *pthread* library. Each thread controls a single *ASHA* and its memory accesses. The Xilinx Multi-Port Memory Controller (MPMC) is employed as it provides an efficient means of interfacing the processor to SDRAM. MicroBlaze accesses memory either through its Local Memory Bus (LMB) port or the On-chip Peripheral Bus (OPB). The LMB provides fast access to on-chip block RAM (BRAM) memories. The OPB provides a general purpose bus interface to on-chip or off-chip memories

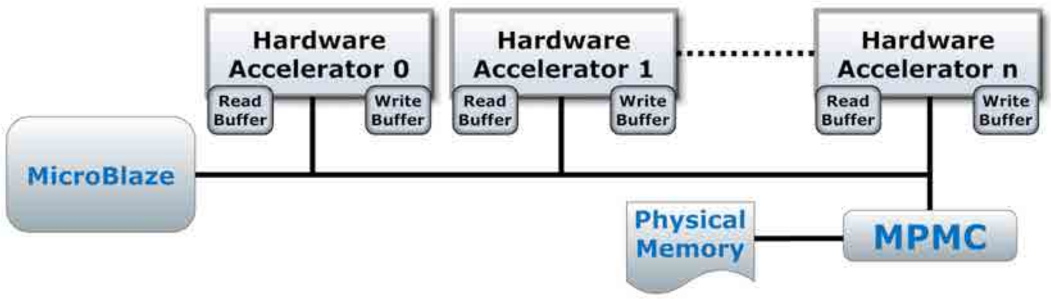


Figure 6.4: Test Architectures: MicroBlaze based multi-ASHA

as well as other non-memory peripherals. The target architecture has 16 KB of each instruction and data cache. The design (excluding *ASHA*s) uses 7225 flip-flops, 6842 LUTs and 14 BRAMs.

6.2.2 PMSS based multi-ASHA System

The PMSS based multi-ASHA system is shown in Figure 6.5. PMSS schedules multi-ASHA similar to the Xilkernel scheduling model. Scheduling is done at the *ASHA* event level. The PMSS provides a hardwired scheduler whereas Xilkernel performs scheduling in software while using fewer hardware resources. In the current (PMSS and MicroBlaze) designs, each *ASHA* is equipped with two read/write buffers to balance the workload. The read/write buffers are connected with PMSS via *SSSI* and a state controller (Figure 2.11 (a)). In the current implementation of PMSS, on a Xilinx ML505 evaluation FPGA board, a 256 MByte (32M x 16) of DDR2 memory having SODIMM I/O module is connected. The system (excluding *ASHA*s) consumes 4786 flip-flops, 3830 LUTs, 10 BRAMs.

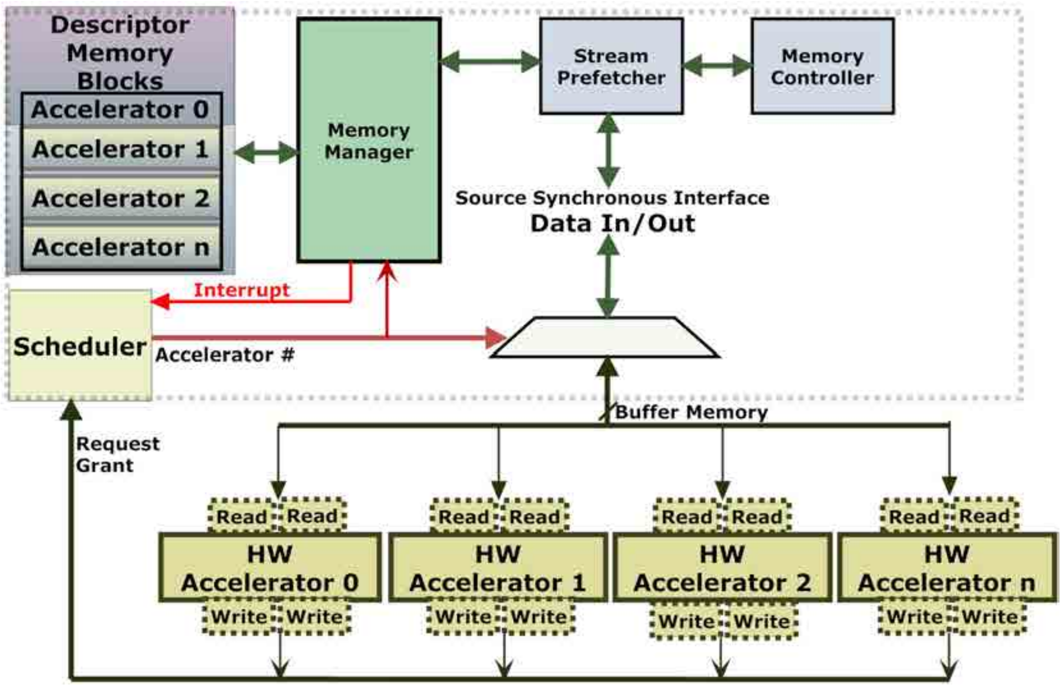


Figure 6.5: Test Architectures: PMSS based multi-ASHA

6.2.3 Test Applications

The application kernels that are used in the design are shown in the Table 6.1. These kernels are selected to test the performance of the system for different data access patterns. The results are validated by comparing the execution time of these kernels on the PMSS system and with the MicroBlaze based system. Separate ROCCC generated *ASHA* cores are used to execute the kernels. In order to give a standard control and interface to *ASHA* cores with the PMSS system, the state controller is used.

6.3 Results and Discussion

This section analyses the results of different experiments conducted on PMSS and MicroBlaze based systems. The experiments are divided into four subsections: *Memory Access*, *Application Performance*, *System Performance* and *Power*

Table 6.1: Brief description of application kernels

Application Kernel	Description	Access Pattern	Registers , LUT	Operations
Radian Converter	Converts degree into radian	Load/Store	68, 67	2
Thresholding	An application of image segmentation, which take streaming 8-bit pixel data and generates binary output.	Load/Store	2289, 2339	1
Finite Impulse Response	Calculates the weighted sum of the current and past inputs.	Streaming	3953, 2960	31
Fast Fourier Transform	Used for transferring a time-domain signal into corresponding frequency-domain signal.	1D Block	4977, 2567	48
Matrix Multiplication	Output= Row[Vector] $\times$ Column[Vector] X = Y $\times$ Z	Column and Row Vector	2925, 1719	62
Smith Waterman	Determining optimal local alignments between nucleotide or protein sequences	Tiled	6205, 2853	12
Laplacian solver	Applies discrete convolution filter that can approximate the second order derivatives.	2D Tiling	3380, 2616	17
3D-Stencil Decomposition [16]	An algorithm that averages nearest neighbor points (size 8x9x8) in 3D.	3D-Tiling	6977, 5567	37

6.3.1 Memory Bandwidth

We compared PMSS systems bandwidth with MicroBlaze processor based system by executing thresholding application having load/store memory access pattern. Figure 6.6 shows a plot with Read/Write data accesses. The X-axis presents data sets that are read and written by the image thresholding application from/to the main memory. The Y-axis of the plot represents the number of clock cycles consumed while accessing the dataset. The main memory single access latency on a 125MHz MicroBlaze processor with 125MHz DDR SDRAM is measured to be almost 50 cycles. The results show that PMSS system while accessing complex memory accesses is 1.4 times faster than MicroBlaze based system. The reason of this speed up that is PMSS manages address in the hardware. The memory access speed up is further improved, up-to 33x, for tiled access patterns. This is because PMSS manages complex patterns independently without support of processor/OS and uses few descriptors that reduce run-time address generation and (on-chip) address request/grant delay.

6.3.2 Application's Performance

The application kernels are executed on PMSS and MicroBlaze based systems. Figure 6.7 shows the execution time (clock cycles) of the application kernels. Each bar represents the application kernel's computation time on *ASHA* and execution time on

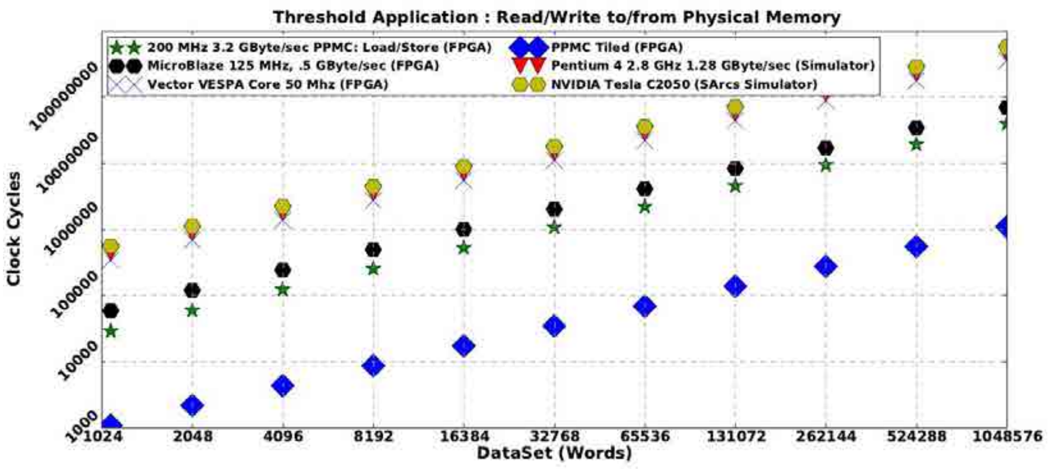


Figure 6.6: DataSet Read /Write Time by different System

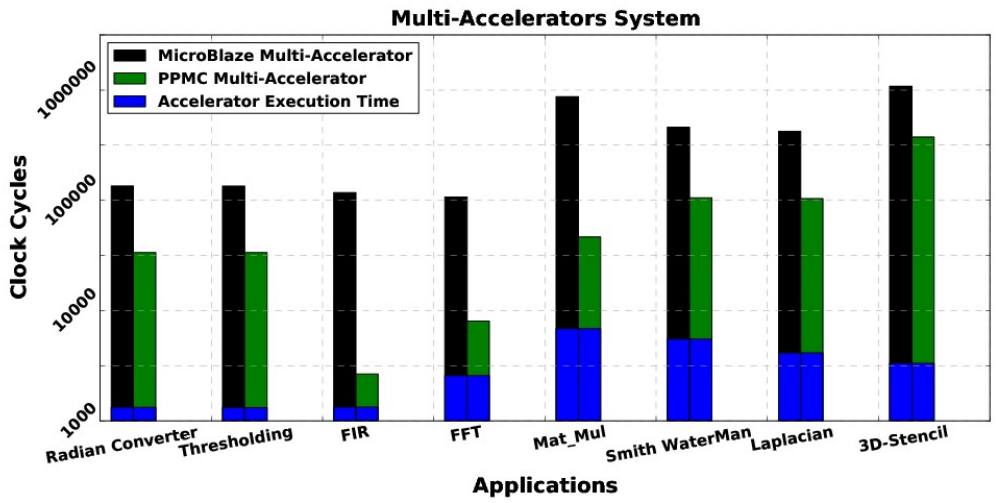


Figure 6.7: Multi-ASHA system execution

the system. The application kernel time contains task execution, scheduling (request/-grant) and data transfer time. The X and Y axis represent application kernels and number of clock cycles, respectively. The results show that by using PMSS system, Thresholding and Radian converter applications achieves 3.5× speed-ups compared to the MicroBlaze based system. These applications have load/store memory access pattern and achieve less speed-up compared to other application kernels. The FIR application has streaming data access pattern with 26.5× speed-up. The FFT application kernel reads a 1D block of data, processes it and writes it back to *Main Memory*. This application achieves 11.9× speed-up. The Matrix Multiplication kernel accesses row and column vectors. The application attains 14× speed-up. The Laplacian and Smith Waterman applications take 2D block of data and achieve 36× and 38× speed-ups respectively. The 3D-Stencil data decomposition achieves 52× speed-up. This speed-up is gained because PMSS stores 3D access patterns in *Descriptor Memory* which reduces address generation time.

6.3.3 System Performance

In the multi-ASHA system, the total execution time includes multiple delays such as interconnect, memory architecture, cache coherence and memory consistency proto-

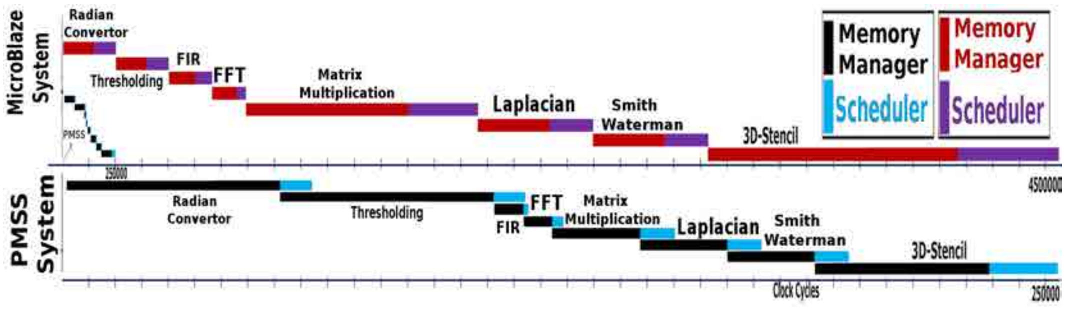


Figure 6.8: Memory Access and Scheduling of multi-ASHA System

cols, bus arbitration, on-chip/off-chip bus translation and flow control. Figure 6.8 illustrates the execution time of the system and categorizes execution time into two factors: arbitration (request/grant) time among the scheduling, and the memory management (bus delay and memory access) time. The computation time of application kernels in both systems overlap under the scheduling and memory access time (shown in Figure 6.7). In the PMSS system memory management, time is dominant, and the PMSS overlaps scheduling and computation under memory access time. The complete PMSS multi-ASHA system achieves 18.6× speed-up.

6.3.4 Static Power

Studies [78] have shown that discrete GPUs can offer performance higher than CPUs and FPGAs. However, a compute-capable discrete GPU can draw more than 200 watts by itself. The On-Chip Static Power in a Xilinx V5-Lx110T device is 1.97 watts while running MicroBlaze based system. PMSS system draws 1.33 watts on-chip static power on a V5-Lx110T device. Due to the light weight of PMSS, the architecture consumes 38% less slices and 32% less on-chip power than the MicroBlaze based system.

6.4 Conclusion

In this work, we have proposed PMSS, which is the result of integrating PMC in a system with multiple ASHAs. The PMSS *Scheduler* efficiently handles multi-ASHAs using *Symmetric* and *Asymmetric* scheduling policies. The *Address Manager* improves the multi-ASHA system performance by managing the resources in a more efficient way, that results in reducing the impact of the ASHA and memory performance gap. The proposed PMSS system is implemented and tested on a Xilinx ML505 evaluation FPGA board. The performance of the system is compared with a microprocessor based system that has been integrated with the Xilinx Xilkernel operating system. The experimental evaluations based on the Xilinx MicroBlaze multi-ASHA system having Xilkernel (RTOS) demonstrates that PMSS based multi-ASHA system best utilizes hardware resources and efficiently accesses the physical data. Results show that the modified PMSS based multi-ASHA system consumes 38% less hardware resources, 32% less on-chip power and achieves approximately a 19x speedup compared to the MicroBlaze based system.

7

AMC: Advanced Multi-accelerator Controller

The rapid advancement, use of diverse architectural features and introduction of High Level Synthesis (*HLS*) tools in FPGA technology have enhanced the capacity of data level parallelism on a chip. A generic FPGA based *HLS* multi-ASHA system requires a microprocessor (master core) which manages memory and schedules accelerators. In a real environment, such *HLS* multi-ASHA system does not give the perfect performance due to the memory bandwidth issues. Thus, system demands a memory manager and a scheduler, that improves performance by managing and scheduling multi-ASHA's memory access patterns efficiently.

In last few years density of FPGAs [11; 79] and performance per watt [80] have improved, which allows High Performance Computing (HPC) industry to increase and provide more functionalities on a single chip. 3D ICs [81] open another dimension in HPC industry that emulates three-dimensional stacked chips by rapidly reconfiguring their two-dimensional fabrics in third spatial dimension of time/space. Such devices have the power to reconfigure their fabric up-to 1.6 billion times per second [81]. The Stacked Silicon Interconnect (SSI) [82] technology provides another dimension to high density FPGAs which satisfies the needs of on-chip resources for HPC system. SSI combines two or more FPGAs for larger and complex systems. As the designs grow larger and complex, the chances of error and complexity increase, thus demands an abstract level design methodology.

[1] Chapter 7 is based on the publications : [16] and [41].

Current emerging technologies and application requirements are indeed changing the way HPC systems are designed. HPC industry wants to execute multi-applications with high performance and low power. This demands a design environment which has dense, and flexible hardware, consumes less power and has abstract level programming tool. Reducing design time and the size of the team are crucial. Numerous architectures (RISC, CISC), design methodologies (SoC, MPSoc), and programming tools (OpenMP, MPI) are available in HPC domain. These systems do not give the perfect performance due to the processor memory speed gap [83; 84] given by the formula
$A\,L = P_{rob} \ast On\,Chip\,Memory_t + (1 - P_{rob}) \ast [On/Off_{chipbus} + DRAM]_t$. The Access Latency *(AL)* depends upon the probability (P_{rob}) to access data from different units such as $On\ Chip\ Memory_t$ (On-Chip data arrangement and probability of reuse), $on/off_{chipbus}$ (flow control, arbitration, translation, interconnection) and $DRAM_t$ (bank, column and row selection).

As has been mentioned in Chapter 1, HLS tools have been strengthened and become truly production-worthy [16] by supporting fundamental technologies such as pipelining and FSM. Initially confined to data path designs, HLS tools are now commenced to address complete systems, including control logic and complex on-chip, off-chip interconnection. The HLS [85] tools provide design modeling, synthesis, and validation at a higher level of abstraction. This makes computation tasks easy to program and less power hungry, thus enabling more sophisticated system design for complex HPC applications. The HLS based multi-ASHA system always require a master processor core which performs memory management, data transfer and scheduling of multi-ASHAs. The master core adds overhead of address/data management, bus request/grant and external memory access time.

To improve HLS based multi-ASHA system performance an intelligent controller is needed that has efficient on-chip *Scratchpad Memory* and *Data Manager*, intelligent front-end/back-end *Scheduler*, fast SSSI link and supports programming model that manages memory accesses in software so that hardware can best utilize them. A PMC based controller with the above mentioned features is integrated with an HLS HCE [29] tool called Advanced Multi-accelerator Controller (AMC). AMC permits HLS programmers to write ASHA with data access patterns that eliminate the requirement of extra core for data transfer and data management. AMC multi-ASHA works

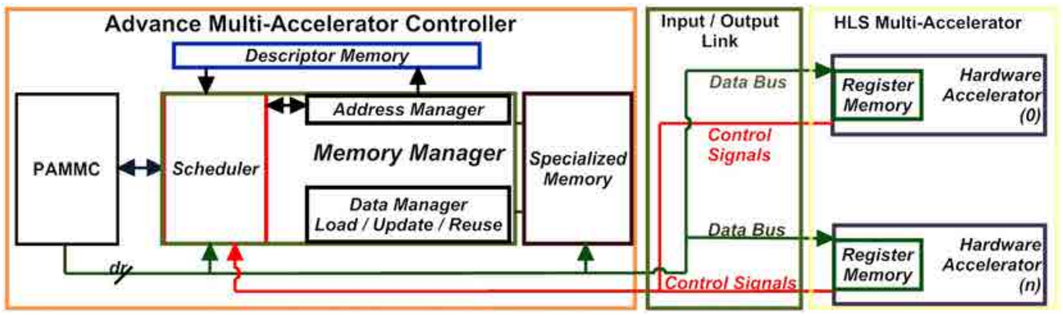


Figure 7.1: Architecture of Advanced Multi-Accelerator Controller Based System

as a standalone system that manages local memory data, handles main memory access patterns and perform automatic parallelization of multiple ASHAs. Some salient features of the proposed AMC architecture are given below:

- The AMC *Address Manager* and *Data Manager* performs data management for complex and regular patterns that efficiently accesses, reuses and feeds patterns to HLS *ASHA*s without support of extra master core.
- The AMC *Scheduler* supports multiple HLS hardware accelerator IPs using event driven handshaking methodology, and this decreases the time-to-market and complexity of hardware.
- The AMC schedules requests from multi-ASHAs taking into account both the processing and memory requirements defined at program-time. At run-time, the AMC back-end scheduler reorders accelerator's memory requests considering SDRAM open banks. This removes the overhead from opening/closing rows/banks and idle cycles on the data bus.

7.1 Architecture

In this section, we describe the Advanced Multi-Accelerator Controller (AMC) system (shown in Figure 7.1) for HLS tool. The architecture is based on four units: the *SSSI*, the *Local Memory System*, the *Memory Manager*, and the *Main Memory System*.

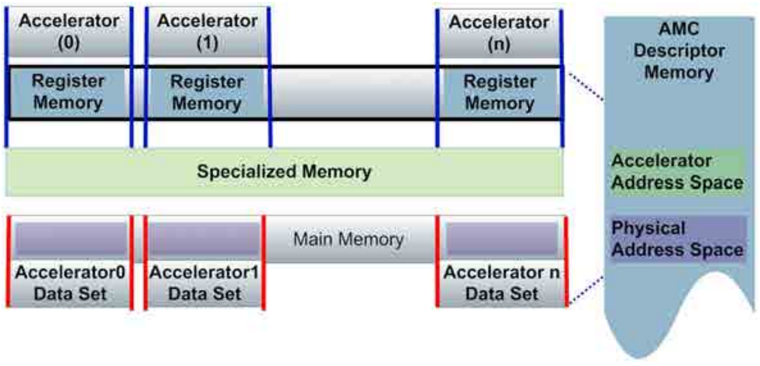


Figure 7.2: Memory Hierarchy of AMC System

The *SSSI* provides an interface between AMC and HLS *ASHA* unit. The *SSSI* (see Section 2.1.1.1.1) is used to read/write high-speed data to/from local memory of HLS multi-ASHA. Transfer of data is accomplished according to the system clock. The state controller takes accelerator data *requests* and manages multiple buffers using *request* and *grant* signals. The AMC *Local Memory System* uses the *Register Memory*, the *Scratchpad Memory* and the *Descriptor Memory*.

To achieve maximum memory bandwidth, the AMC organizes complex access patterns in single or multiple descriptors at compile-time. At run-time, the AMC *Memory Manager* transfers complete pattern to/from the main memory and *Specialized Memory* in multiple noncontiguous strided streams. The AMC *Memory Manager* has a view of main memory address space that is partitioned into data sets (shown in Figure 7.2). Each segment contains a single or multiple descriptors that hold the information of specialized and main memory for each accelerator.

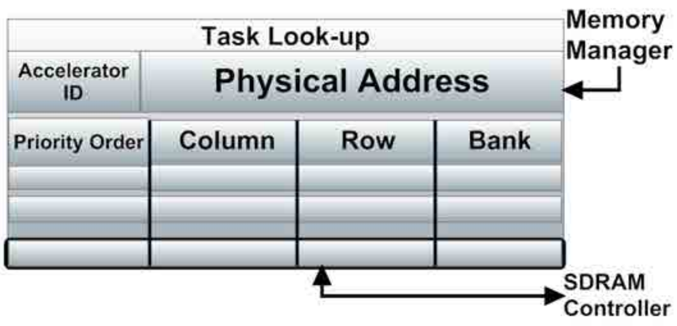


Figure 7.3: AMC: Back-End Scheduler Lookup Table

The AMC *Memory Manager* applies protection at the segment level e.g. a segment can be read/written by the accelerator for which it is allocated. Within a segment, AMC organizes and rearranges multiple noncontiguous memory accesses simultaneously that reduces read/write delay due to the control selection of SDRAM memory. To reduce false sharing, the *Specialized Memory* is by default dedicated to a single accelerator unit. AMC keeps the knowledge of memory as to whether or not a certain memory area is in the accelerator's *Specialized Memory*. This knowledge allows the AMC to manage the placement of memory as well as reuses and shares already accessed memory. The *Memory Manager* is further divided into three parts which are: the *Scheduler* the *Address Manager* the *Data Manager*.

The AMC scheduler manages read/write memory accesses and controls operations of multiple accelerators. The scheduler is divided into two sections, the *Front-End Scheduler* and the *Back-End Scheduler*. AMC *Front-End Scheduler* supports two scheduling policies, *symmetric* and *asymmetric* that execute accelerators efficiently. The AMC *Back-End Scheduler* employs a strategy that gathers multiple memory requests, manages them with respect to physical addresses (SDRAM) and maximizes the reuse of open SDRAM banks that decrease the overhead of opening and closing of rows. This strategy imposes conditions on the arrangement of the memory accesses and affects the worst-case latency and gross/net bandwidth of external memory. The scheduling of memory accesses is dependent upon the physical address of current and next transfer. At run-time the *Back-End Scheduler* gathers memory requests from multi-ASHA and places them in the *Address Look-up Table* (shown in Figure 7.3). The *Address Look-up Table* contains unordered memory access requests. Each memory access is categorized into four parts i.e. *ID*, *Bank*, *Row* and *Column*. The *ID* holds the information of an accelerator and its local memory. The *Bank*, *Row* and *Column* belongs to the *Main Memory* address space. At run-time, the *Back-End Scheduler* schedules memory accesses of multiple accelerators by giving highest priority to the bank and row address. The lookup table executes the AMC policy called bank/row address management policy. The policy manages addresses in a lookup table so that it accesses SDRAM memory that is available in the row buffer or have the same bank. The fastest memory access occurs when accessing the same row buffer as the previous access and only requires column access. The longest memory access (when Bank conflicts) requires a pre-charge signal, followed by row and column accesses. The

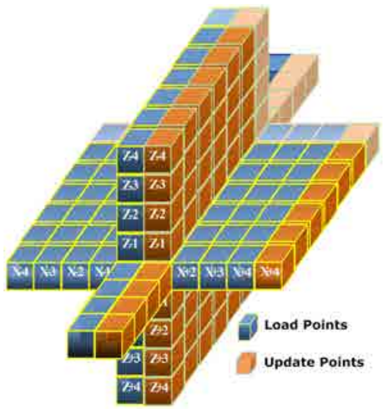


Figure 7.4: N-Stencil Vector Load & Update Points

Memory Manager (Section 2.1.3) of AMC has separate *Descriptor Memory* (register set) for each accelerator unit, shown in Figure 7.2. These descriptors are masked with interrupt and request signals. Once a request is generated the Memory Manager starts memory operation for the requested accelerator using its descriptors. After completion of memory read/write operation, the AMC scheduler receives an interrupt (*ack*) signal from the Memory Manager unit. This signal informs the scheduler about the selection of the next *ASHA* to execute. The scheduler captures the *ack* signal from the Memory Manager and assigns the *grant* signal to the appropriate accelerator unit.

The *Address Manager* fetches single or multiple descriptors depending on the access pattern, translates/reorders in hardware, in parallel with AMC Read/Write operations. The *Data Manager* improves the *(computed$_{point}$ / accessed$_{point}$) (c/a)* ratio by organizing and managing the memory accesses. For an accelerator generating single *computed$_{point}$*, the maximum achievable (ideal) *c/a* ratio is 1. To provide efficient data access and reuse, the *Data Manager* is further divided into three units: the *Load Unit*, the *Update Unit* and the *Reuse Unit*. The *Load unit* accesses all points of an access pattern, which are required for a single *Computed$_{point}$*. After accessing the pattern once (*points*), the memory manager transfers control to *Reuse unit* and *Update unit*. The *Reuse unit* keeps reusing input *points* as much as possible. The *Update Unit* is responsible to update remaining memory access (*points*) (not already accessed) required for the application kernels.

For example, a generic stencil structure (shown in Figure 2.5) when n=4, requires 25 points to compute one central point. This means the *computed$_{point}$/access$_{points}$ (c/a)* ratio is `0.04`. To improve the *(c/a)* ratio the *Data Manager* is deployed to increase data reuse by feeding data efficiently to the computation engine. The *Data Manager* accesses multiple stencils from the input volume in the form of a stencil vector. This is shown in Figure 7.4. *(N × (1+(n× 4)) + (n × 2))* represents the size of a single 3D-Stencil vector. Here, N represents the number of planes. Since the 3D-stencil's output is generated by processing consecutive neighboring points in 3 dimensions, a large amount of data can be reused with an efficient data management. The *Load unit* accesses the 3D-Stencil vector at the start of each row of the 3D-Memory volume. For the following vector access, the Load unit transfers control to the Update unit. For a single 3D-Stencil volume, the number of accessed points is dependent on the number of planes and the stencil size. In this case, a single Stencil vector (Figure 7.4) needs `600` points. The load unit reduces these numbers by reusing the points in the y-dimension. These points ($point_y$) are reused when they are part of a neighboring plane of the stencil vector. The number of Load unit points is mentioned in Equation 7.1, where $Ghost\_point_z$ refers to the points of the extended base volume and $Point_c$ indicates the central point.

$$Load\ Unit\ Points = (Planes \times (Point_x + Point_z + Point_c)) + Ghost\_Point_z \tag{7.1}$$

After reading the first stencil vector, the *Memory Manager* shifts the control to the *Update Unit*. For each new access, the *Update unit* slides the stencil vector towards the x-direction and accesses further points while reusing neighboring points. The number of points required by the *Update Unit* for a stencil vector is presented in Equation 7.2.

$$Update\ Unit\ Points = (Planes \times (Point_x + Point_c)) + Ghost\_Points_z \tag{7.2}$$

After accessing the first row, the Reuse unit accesses the rest of the volume. This unit accesses only the central point and generates a stencil vector while reusing the existing rows and columns as mentioned in Equation 7.3.

$$Reuse\ Unit\ Points = (Planes \times Point_c) + Ghost\_Points_z \tag{7.3}$$

The Memory Manager improves 3D-Stencil *(c/a)* ratio. The Memory Manager uses a single input point 25 times before discarding it from the internal memory. In practice, this is not achievable due to the ghost points (i.e. points belonging to a neighboring tile that are necessary for the current computation) present on the boundaries of the input volume. The Memory Manager improves data reuse ratio for large base volumes.

In the current AMC design, the SDRAM device uses four banks of memory per device therefore 2 address bits are used to select the memory bank. For the selection of the appropriate row and column within that row of memory 13 and 10 address lines are used respectively that completes the address mapping from physical address to the memory address. The *PAMMC* has a peak bandwidth of 3.2 GB/s since it has a clock frequency of 200 MHz, a data rate of 2 words per clock cycle, and a data bus width of 64 bits.

7.2 Evaluation Architecture

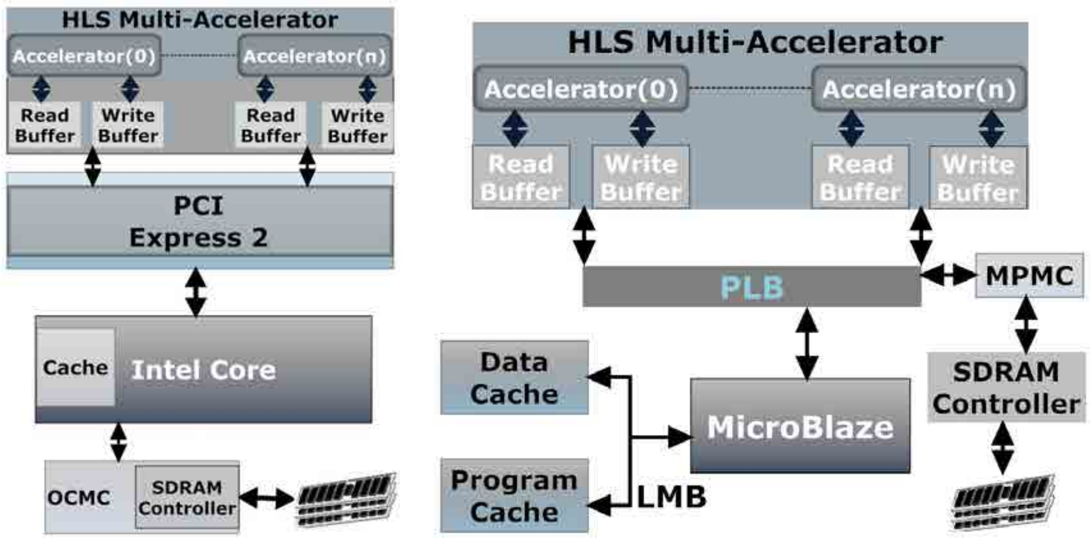
In this section, we describe the AMC based HLS multi-ASHA system. In order to evaluate the performance of the AMC system, the results are compared with a MicroBlaze and Intel core based HLS multi-ASHA systems. The Xilinx Integrated Software Environment [74] and Xilinx Platform Studio [75] are used to design the HLS multi-ASHA system. The power analysis is done by Xilinx Power Estimator [76]. A Xilinx ML505 evaluation FPGA board [52] is used to test the multi-ASHA systems. The section is divided into four sub-sections, the *Intel based HLS multi-ASHA System*, the *MicroBlaze based HLS multi-ASHA System*, the *AMC based HLS multi-ASHA System* and the *HLS multi-ASHA Kernels*.

7.2.1 Intel based HLS multi-ASHA System

The Intel system with HLS multi-ASHA is shown in Figure 7.5 (a). The system architecture is further divided into three sections the *Master Core*, the *Programming API*, and the *Bus Unit*.

7.2.1.1 The Master Core

The Intel Core i7 CPU is used to manage the memory system and schedule the HLS multi-ASHA. To achieve the maximum performance the HLS multi-ASHA system is executed on an optimized multi-threaded reference implementation, written in C++,



(a) (b)

Figure 7.5: HLS multi-ASHA Systems: (a) Intel Core System (b) MicroBlaze System

compiled with g++ with optimization -O3, and executed on system having a quad-core Intel Core i7-2600, 3.4 GHz, with 16 GB RAM, 1333 MHz bus. The system uses Ubuntu 11.04 OS with Linux kernel version 2.6.3. These higher memory baselines are required to enable sufficient memory for the HLS multi-ASHA kernels. In the current system, Performance Application Programming Interface (PAPI) hardware performance counter used to collect execution clock cycles for each accelerator kernel.

7.2.1.2 The Programming API

OpenMP (Open Multiprocessing) is an API that caters multi-platform shared memory systems and extend it beyond real HPC systems that contain embedded systems, real time systems, and *ASHA*s. Tasks-based execution is a significant feature of OpenMP 3.0. OpenMP 3.0 task pragmas make it compatible with the idea of using an HLS multi-ASHA. The shared memory parallelism is specified by C/C++ program using a set of compiler directives and runtime routines that improve the run-time performance of the system. OpenMP executes them independently ensures that all defined accelerators and data transfer tasks are completed at some point.

7.2.1.3 The PCI Bus Unit

Intel core manages data movement between *On Chip Memory Controller* (OCMC) and multi-ASHA by using PCI bus. The PCI bus has multiple DMA channels which manage the data transfer requests. The data transfer requests issued to PCI DMA channels are according to available Send/Receive interfaces which forward them to the appropriate DMA channel. The XpressLite2 IP [86] is used in the design to manage 8 separate data flows using DMA channels. The PCI Express XpressLite2 IP transfers the data sets from the host machine to multi-ASHA. The PCI Express IP is programmed to work at 1 G Byte/s using a 125 MHz clock speed and a 64-bit data bus.

7.2.2 MicroBlaze based HLS multi-ASHA System

The FPGA based MicroBlaze system is proposed (Figure 7.5 (b)) to execute HLS multi-ASHA kernels. The design (excluding hardware accelerators) uses 7225 flip-flops, 6142 LUTs and 15 BRAMs. A MicroBlaze *SSP* is used in the HLS multi-ASHA system that perform scheduling and data memory management.

In the design, a Processor Local Bus (PLB) [54] provides connection between hardware accelerators and microprocessor. The PLB has 128 bit-width and connected to a bus control unit, a watchdog timer, separate address/read/write data path units, and an optional DCR (Device Control Register) slave interface that provides access to a bus error status registers. Bus is configured for single masters (MicroBlaze) and multi slaves (HLS multi-ASHAs). An arbiter is used to grant access to the *ASHA*. The Input/Output (I/O) Module [87] is a light-weight implementation of a set of standard (I/O) functions commonly used in a MicroBlaze processor sub-system. The (I/O) bus provides access to external modules using MicroBlaze Load/Store instructions. The Input/Output Bus is mapped in the MicroBlaze memory space, with the *I/O* bus address directly reflecting the byte address used by MicroBlaze Load/Store instructions. The PLB provides a maximum of 2 GByte of bandwidth while operating at 125MHz and 128-bit width, with byte enables to write byte and half-word data.

The target architecture has 32 KB of each instruction and data cache. To access data from main memory, a parameterizable Multi-Port Memory Controller (*MPMC*) [88] is employed. *(MPMC)* provides an efficient interfacing between the processor and

SDRAM. The *MPMC* connects SDRAM with MicroBlaze processors using IBM CoreConnect Processor Local Bus (PLB). A DDR2 controller is used with *MPMC* to access data from DDR2 SDRAM memory. The supported DDR2 memory has a peak bandwidth of 1 GByte/s as it has a clock frequency of 125MHz, and a data bus width of 64 bits.

A small light-weight easy-to-use Real-Time Operating System (RTOS) Xilkernel [77] is employed on the MicroBlaze soft processor. Xilkernel is highly integrated into the design tools of Xilinx, which makes it possible to configure and build an embedded system using MicroBlaze and Xilkernel. Xilkernel API performs scheduling, inter-process communication and synchronization with a Portable Operating System Interface *POSIX* interface. The Xilkernel POSIX support statically declares threads that start with the kernel. From the main function, application is spawn into multiple parallel threads using pthread library. Each thread controls a single HLS accelerator and its memory access. The software application consists of Xilkernel and application kernel threads executing on top of the main program.

MicroBlaze system uses Xilinx Software Development Kit (SDK) [89] that compiles the application kernels using library generator (libgen) [90] and a platform-specific gcc/g++ compiler and generates the final executable object file. The Xilinx library generator (libgen) is used to produce libraries and header files necessary to build an executable file that controls HLS multi-ASHA. Libgen parses the system hardware and sets up drivers, interrupt handling, etc. and creates libraries for the system. The libraries are then used by the MicroBlaze GCC compiler to link the program code for the MicroBlaze based HLS multi-ASHA system. The object files from the application and the Software Platform are linked together to generate the final executable object file.

7.2.3 AMC based HLS multi-ASHA System

The AMC based HLS multi-ASHA System is shown in Figure 7.6. AMC controls the HLS multi-ASHA and performs scheduling and memory management without intervention of microprocessor or operating system (OS). In the current implementation of AMC, on a Xilinx ML505 evaluation FPGA board, a 256 MByte (32M x 16) of DDR2 memory having SODIMM I/O module is connected with AMC. The main memory has

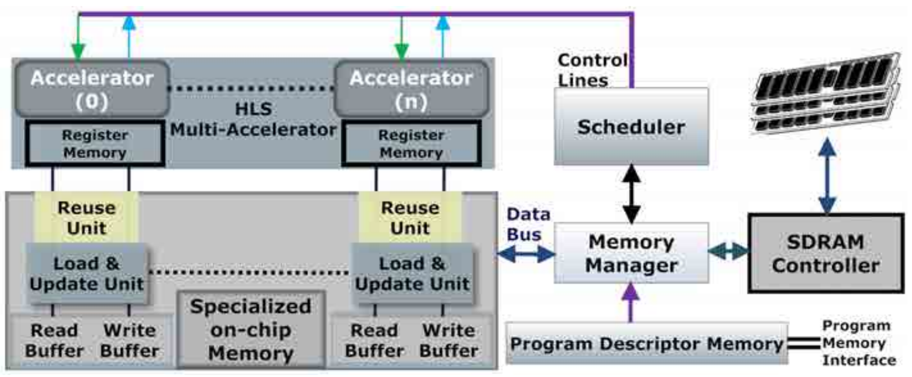


Figure 7.6: HLS multi-ASHA Systems: AMC System

a peak bandwidth of 1GByte/s since it has a clock frequency of 125MHz, a data rate of 2 words per clock cycle, and a data bus width of 32 bits. The system (excluding HLS accelerators units) consumes 4986 flip-flops, 4030 LUTs, 12 BRAMs.

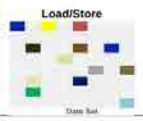
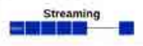
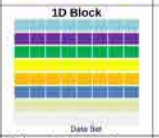
7.2.4 HLS multi-ASHA Kernels

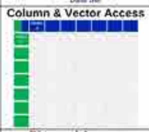
The application kernels that are used in the design are shown in Table 7.1. The *ASHA*s are generated by the HLS ROCCC[23] compiler for the evaluated application kernels. A wrapper module and a *state controller* (Figure 2.11 (a)) is integrated with each hardware accelerator to manage multiple buffers and makes it feasible to be integrated in the AMC, MicroBlaze and Intel core based systems. Column *Access pattern* of Table 7.1 presents memory access patterns of the application kernels. Each color represents a separate memory access pattern. The column *Reg, LUT* describes the slices (Registers, LUTs) utilized by HLS based hardware accelerators on Virtex-5 ML505 device. The column *Points* shows a number of inputs *($access_{point}$)* required to generate a single output.

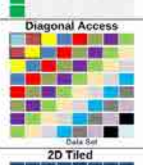
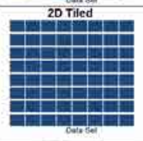
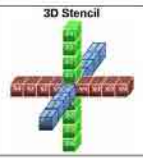
7.3 Results and Discussion

This section analyzes the results of different experiments conducted on AMC, MicroBlaze and Intel based systems. The experiments are classified into four subsections: *Application Performance*, *System Performance*, *Memory Access Unit*, and *Area and Power*.

Table 7.1: Brief description of application kernels

Application Kernel	Description	Access Pattern	*Reg, LUT*	GFLOPS
Radian Converter	Converts Degree into radian		68, 67	0.375
Thresholding	An application of image segmentation, which takes streaming 8-bit pixel data and generates binary output.		2289, 2339	0.125
FIR Finite Impulse Response	Calculates the weighted sum of the current and past inputs.		3953, 2960	3.875
FFT Fast Fourier Transform	Used for transferring a time-domain signal into corresponding frequency-domain signal.		4977, 2567	6.0

Matrix Multiplication	Output= Row[Vector] × Column[Vector]
	X=Y×Z
			2925, 1719	7.750

Smith Waterman	Determining optimal local alignments between nucleotide or protein sequences		3380, 2616	1.125
Laplacian Solver	Applies discrete convolution filter that can approximate the second order derivatives.		6977, 5567	2.125
3D-Stencil Kernel	An algorithm that averages nearest neighbor points (size 8x9x8) in 3D.		6205, 2853	4.625

7.3.1 Application Performance

The application kernels (Table 7.1) are executed individually on AMC, MicroBlaze and Intel based systems. The section is further divided into two subsections that are: *Application Performance without On-Chip Memory* and *Application Performance with On-Chip Memory*.

7.3.1.1 Applications Performance without On-Chip Memory

This section presents execution time (clock cycles) of each application kernel by disabling on-chip (specialized/cache) memories of the systems (shown in Figure 7.7). Each bar represents the application kernel's computation time and memory access time. The application kernel computation time contains the HLS accelerator process-

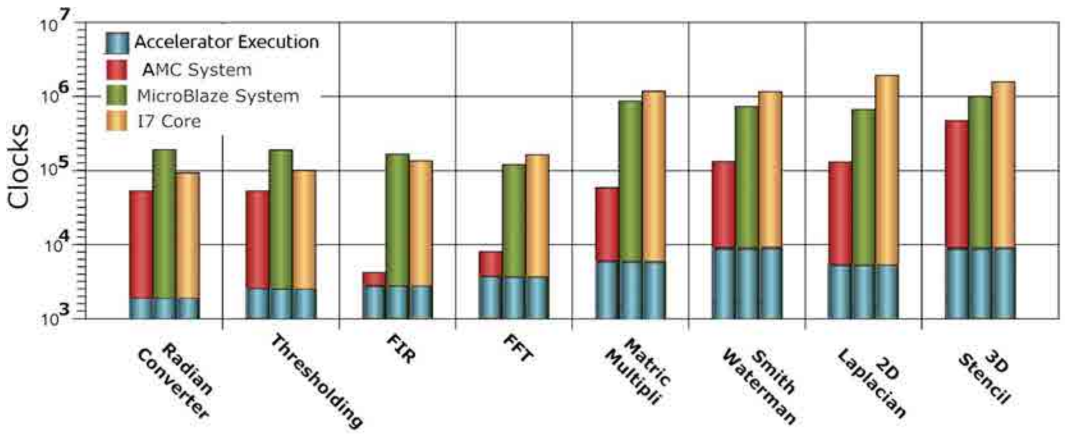


Figure 7.7: Application Kernels Execution Time Without On-chip Memory Support

ing time for 4KByte of data. Memory access time holds address/data management and request/grant time from main memory unit. X and Y axis represent application kernels and number of clock cycles, respectively. The vertical axis has logarithmic scale in the Figure 7.7. By using the AMC system, the results show that Radian converter achieves 3.94x and 1.75x of speed-up compared to the MicroBlaze and Intel based systems respectively. The Thresholding application achieves 7.1x and 1.89x of speed-up. These applications have load/store memory access pattern. The FIR application has streaming data access pattern with 43.2x and 32x of speed-up. The AMC system requires only one descriptor block to access the data pattern thus reduces address generation/-management and on-chip communication time. The FFT application kernel reads a 1D block of data, processes it and writes it back to the *Main Memory*. This application achieves 22x and 20x of speed-up. The Matrix Multiplication kernel accesses row and column vectors. The AMC system manages complex data patterns of the application

Table 7.2: AMC: $computed_{point}/access_{point}$ Ratio of Local Memory System

Application Kernel	Load Value	Reuse Value	Update Value	*c/a* without Memory Manager	Achieved *c/a* with Memory Manager
Radian Converter	1	0	1	1	1
Thresholding	1	0	1	1	1
FIR	128	127	1	0.0078	1
FFT	64	0	64	0.015	0.015
Matrix Multiplication 32x32 (Row x Column)	64	32	32	0.015	0.031
Smith Waterman	3	0	3	0.33	0.33
2D Laplacian (3x3)	9	8	1	0.1	1
3D Stencil (8x9x8)	25	24	1	0.04	0.125

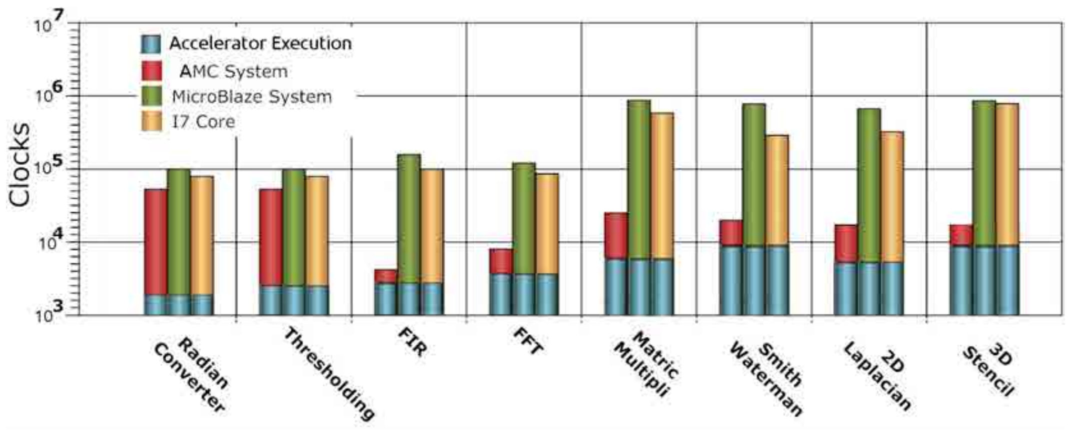


Figure 7.8: Application Kernels Execution Time with On-chip Memory Support

in hardware and attains 29x and 19.9x of speed-up. The Smith Waterman application achieves 10.4x and 8.7x of speed-up. The Laplacian filter achieves 6.15 and 14.83 of speed-up. Both Laplacian and Smith-Waterman applications have 2D Tiled (block) access pattern. The 3D-Stencil data decomposition achieves 3.6x and 3.3 of speed-up. Results show that Intel system data transfer is always faster than MicroBlaze system due to its efficient prefetching and data handseling support. The AMC system manages data transfers in patterns and reuse it in local memory that improves the system bandwidth and overall system performance.

7.3.1.2 Applications Performance with On-Chip Memory

In this section, we calculate *computed*$_{point}$*/access*$_{point}$ (c/a) ratio of AMC system for each application kernel and compare the performance with MicroBlaze and Intel systems. The column *c/a without memory manager* of Table 7.2 represents data elements required to generate a single output. *Radian Converter* and *Thresholding* kernels have load/store access pattern with 1 *c/a* ratio, it means only a single element from main memory is required to the computing unit. Due to irregular data access pattern both applications are unable to get the benefit from on-chip memory. As shown in Table 7.2 and Figure 7.8 the application *FIR, FFT, Matrix Multiplication, Smith Waterman, 2D Laplacian, and 3D-Stencil* need more than a single data element for each computations. A 128-Tap FIR filter requires 128 number of inputs to generate a single output. The

AMC Data Manager improves c/a ratio of FIR and 2D Laplacian Kernel to 1 by reusing accessed points. FFT and Smith Waterman kernel's data elements are not reused by AMC system due to complex access pattern. Matrix Multiplication application kernel takes 32 element wide row and column vector to generate a single element. The AMC system reuses row vector and accesses only column vector for each multiplication. For generic stencil (n=4) application kernel, 25 points are required to compute one central point. This means the *computed*$_{point}$*/access*$_{point}$ *(c/a)* ratio is `0.04`. The AMC memory system improves the *(c/a)* ratio by reusing and feeding data efficiently to the computation engine. The AMC Memory system keeps updating/using all memories, so that memory accesses do not affect the performance of the system. When executing application on the system by enabling on-chip (cache/specialized) memory units results show that Radian converter and Thresholding applications do not improve performance due to their irregular memory pattern and having no temporal data locality. While executing FIR application, the results show that AMC system achieves 26.5x and 14.4x of speed-up compared to MicroBlaze and Intel systems respectively. The FIR application has streaming data access pattern with maximum sequential data locality. The FFT application kernel reads a 1D block of data, processes it and writes it back to the main memory. This application achieves 11x and 8.5x of speed-ups. The Matrix Multiplication kernel accesses row and column vectors. The application attains 14x and 9.6x of speed-up. The Smith Waterman and Laplacian application have 2D block/tiled data access pattern. The Smith Waterman application has no data locality and achieves 36.3x and 14.3x of speed-up. The Laplacian filter has temporal data locality and achieves 38.8 and 18.73 of speed-up. The 3D-Stencil application kernel has 3D tiled memory access pattern with complex data locality. The AMC system's 3D data and *Descriptor Memory* manages/reuses 3D-Stencil data and 3D tiled access pattern respectively and achieves 58x and 53.7 of speed-up.

7.3.2 System's Performance

The system performance is measured by executing HLS multi-ASHA kernels all together on AMC, MicroBlaze and Intel based systems. All application kernels are executed simultaneously with the different set of priorities. At run time, AMC system and baseline systems manage executions and pipeline/overlap data transfer where

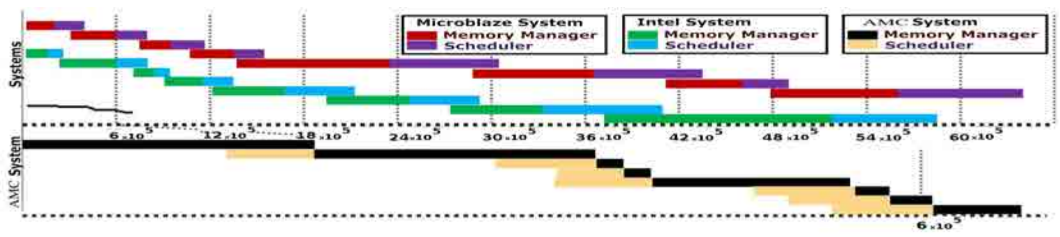


Figure 7.9: HLS multi-ASHA Systems Execution Time

possible. Figure 7.9 illustrates the execution time of the system and categorizes execution time into three factors: computation (application processing) time, arbitration (request/grant) time among the scheduling, and the memory management (bus delay and memory access) time. The computation time of application kernels in all systems is overlapped under the scheduling, and memory access time (shown in Figure 7.7); therefore, it is not shown in Figure 7.9. The Intel based system holds PCI bus communication which takes extra time to access data from the *Main Memory*. In the AMC system, memory management time is dominant, and the AMC overlaps scheduling and computation under memory access time. While running all HLS accelerator kernels together, the results show that the AMC based system achieves 10.4x and 7x of speed-up compared to MicroBlaze and Intel Core based systems. The AMC system efficiently schedules multi-ASHA and manages memory access patterns.

7.3.3 Area and Power

In this section, we measured the static power of AMC, MicroBlaze and Intel systems without having HLS multi-ASHA system. The Intel Core i7 CPU (4 Cores) with a system clock of 2.4 GHz and 8 GByte of global memory consumes 15.80 watts static power [91]. The MicroBlaze and AMC systems without having HLS multi-ASHA consume 2.75 and 2.1 watts respectively on Xilinx V5-Lx110T FPGA device. Both systems have a system clock of 125 MHz and 256 MByte of global memory. Due to the light weight of AMC, the system consumes 32% fewer slices and 23% less on-chip power than the MicroBlaze based system.

7.4 Conclusion

HLS based multi-ASHA system suffers from poor performance on FPGA architectures due to processor memory speed gap. A generic HLS multi-ASHA system requires a master core (microprocessor) that controls multi-ASHA and manages the memory system. In this work, we integrate PMC in HLS based multi-ASHA environment called AMC, which handles regular and complex pattern requests of multiple *ASHA*s. The AMC *Address Manager* and *Data Manager* improve the HLS based multi-ASHA system performance by managing complex memory patterns. The AMC *Scheduler* applies *Symmetric* and *Asymmetric* scheduling policies that handles multiple *ASHA*s requests without the support of processor and operating system. The proposed environment can be programmed by the microprocessor using High Level Language (HLL) API or directly from an accelerator using a specific command interface. The AMC system is evaluated with memory intensive accelerators – High Performance Computing (HPC) applications – implemented and tested on a Xilinx ML505 evaluation FPGA board. The experimental evaluations based on the MicroBlaze and Intel based HLS multi-ASHA systems with Xilkernel (RTOS) and Linux kernel respectively demonstrates that AMC based HLS multi-ASHA system best utilizes hardware resources and efficiently accesses physical data. The performance of the system is compared with the microprocessor based systems that have been integrated with the operating system. Results show that the AMC based HLS multi-ASHA system achieves 10.4x and 7x of speed-up compared to the MicroBlaze and Intel core based HLS multi-ASHA systems.

Part IV

Uni-core Memory System for Irregular Data Pattern

8

APMC: Advanced Pattern based Memory Controller

There have been a number of techniques proposed to overcome the problem of the *memory wall*. Cache memories [3] are very effective but only if the working set fits in the cache hierarchy and there is locality. In many HPC applications, the data sets can be large and have irregular patterns, thus reducing the spatial locality. Hardware prefetching [6] is effective in hiding the memory latency when the accesses are known in advance, but it cannot be tailored for data-dependent accesses, e.g. lists, trees, etc. Adaptive software prefetching [7] can be utilized to change prefetch distances during runtime, but it is difficult to insert prefetch information for irregular access patterns at runtime. A number of intelligent and high-performance memory systems [8; 9] exist to manage the processor/memory speed gap. Unfortunately, these memory controllers rely on a master processor and are typically limited to applications with regular memory access patterns thereby prohibiting the acceleration of applications with irregular patterns. Due to limited support of irregular access patterns in existing memory controllers, the FPGA systems are normally not capable of effectively leveraging code with pointer-based data structures [10].

In this chapter, we use a PMC based single core system with regular and irregular access patterns support. We propose the *Advanced Pattern based Memory Controller* (APMC), a technique to accelerate both regular and irregular memory access patterns

[1] Chapter 8 is based on the publications : [43] and [42].

by rearranging memory access patterns to minimize access latency based on the information provided by pattern descriptors. APMC operates independently from the master core at run-time. APMC keeps pattern descriptors in a separate memory and prefetches the complete data structure into a *Specialized Scratchpad Memory* (Section 2.1.2.3). Memory accesses are arranged in the pattern *Descriptor Memory* at program- and run-time to reduce access latency. APMC manages data movement between the main memory and the *Specialized Scratchpad Memory*, data present in the on-chip memory is reused and/or updated when accessed by several patterns. Support for irregular access patterns is provided with a specialized data structure that describes irregular patterns, and a hardware communication protocol is implemented between the compute unit and APMC. The salient contributions of the proposed APMC architecture are:

- Support for both regular and irregular memory access patterns using the memory pattern descriptors thus reducing the impact of memory latency.
- A *Specialized Scratchpad Memory* that tailors local memory organization and maps complex access patterns.
- A *Data Manager* that efficiently accesses, reuses and feeds irregular data access patterns.
- Data management and handling of unpredictable, irregular memory accesses at run-time, without the support of a processor or the operating system.
- When compared to the baseline system implemented on the Xilinx FPGA, APMC transfers regular and irregular data sets up to 20.4x and 3.4x faster respectively and achieves between 3.5x to 52x and 1.4x to 2.9x of speedup for regular and irregular applications respectively.

8.1 APMC Architecture

The APMC supports complex regular and irregular memory access patterns and reduces run-time overhead of address management. It includes an on-chip specialized memory unit that efficiently accesses, reuses and feeds data to the computing unit. The

APMC reduces memory access latency by arranging memory addresses at compile-time using descriptors and run-time in hardware.

The main units of APMC are shown in Figure 8.1. The *Compute Unit* executes the application and can be either a processor or an accelerator. The *Front-End Interface* provides a link between APMC and the *Compute Unit*. It includes two distinct links, the *Program Line* and the *Data Line*. The *Program Line* of the *front-end interface* is used to program the *Descriptor Memory*. The *Data Line* is used to move data patterns between the compute units and the *Specialized Scratchpad Memory* (see Section 2.1.2.3). The *Local Memory System* stores both data and the access pattern descriptors, in the *SSM* and *Descriptor Memory* respectively. The *Data Manager* takes a single or multiple descriptors and transfers data patterns. It is divided into the *Load Unit*, the *Reuse Unit*, and the *Update Unit*. The *Main Memory System* is responsible for transferring data to/from main memory (SDRAM). The *Memory Access System* provides a high-speed source-synchronous interface and transfers data on both edges of the clock cycle. The units of APMC operate independently in parallel with different descriptors/requests.

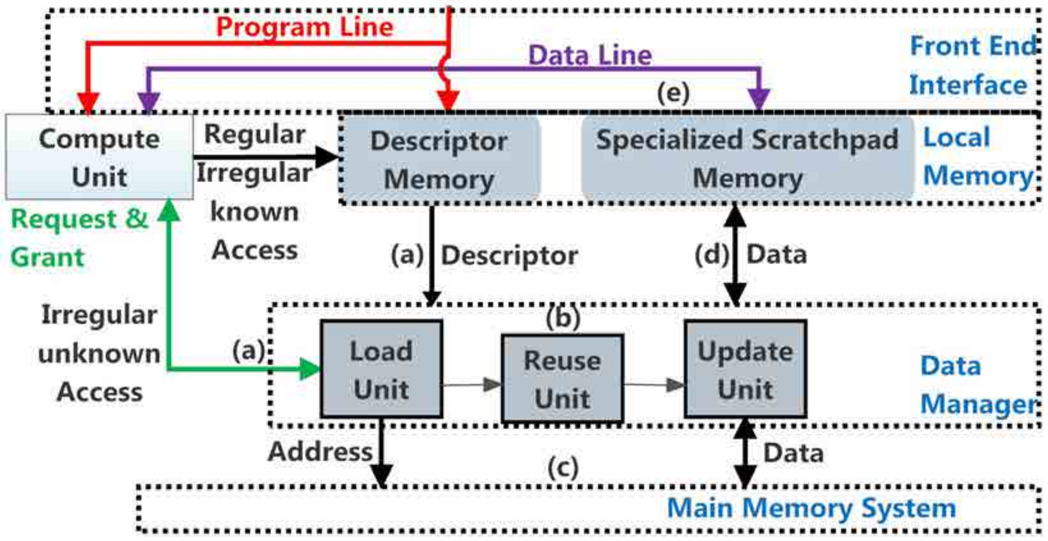


Figure 8.1: APMC: Block Diagram With Data Flow

8.1.1 APMC Working Operation

Figure 8.1 presents the control- and data-flow between units of APMC, divided into run-time steps from (a) to (e). The *Program Line* transfers the APMC binary file to the *Descriptor Memory* during initialization time.

(a) The *Load Unit* of the *Data Manager* reads complex and irregular memory patterns from the *Descriptor Memory*. Depending upon the access pattern the *Data Manager* takes a single or multiple descriptors from the *Descriptor Memory* and reads/writes data to/from the *Specialized Scratchpad Memory* and the *Main Memory System*.

(b) The *Reuse Unit* holds the information of accessed data and requests formerly un accessed data.

(c) The *Main Memory System* takes memory addresses from the *Load Unit* and transfers data between the *Update Unit* and SDRAM.

(d) The *Data Manager* reuses or updates data in the *Specialized Scratchpad Memory* and when possible skips (c).

(e) The compute unit uses the bus link to access the *Specialized Scratchpad Memory* (*SSP*).

In our current evaluation on Xilinx Virtex-5, the *SSM* has 32 banks and each bank holds 128 $\times$ 128 Bytes (32 $\times$ 32 words) (row $\times$ column). Each bank uses a single BRAM (`1x 36 Kb`), which is controlled by a separate BRAM controller and has a different base address. In the current architecture, the number of banks is fixed but row and column size can be changed depending upon the dimensions of the data set. A single or two dimension data sets are placed in a single bank and can use single or multiple BRAM/s. For a 3D data set, depending upon the 3rd dimension, up to 32 banks can be used. Depending upon the dimensions of data set, the *SSM* can be arranged by re-programming the APMC *Descriptor Memory*. The APMC accesses and places data in tiles if the data set is larger than the *SSM* structure.

The APMC uses *Regular/Irregular Descriptor Memory* to transfer complex and irregular access patterns. For each memory access descriptor, the *Data Manager* (shown

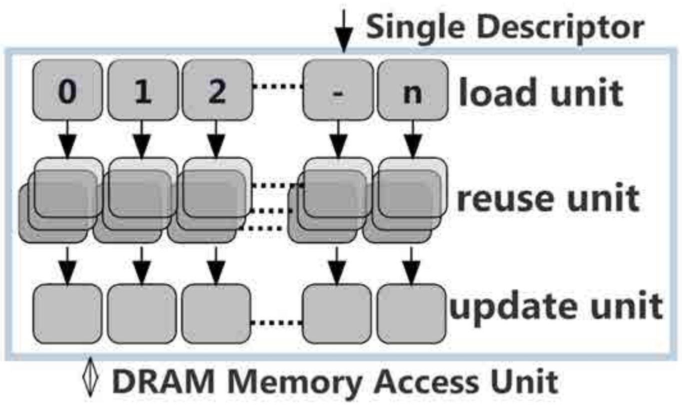
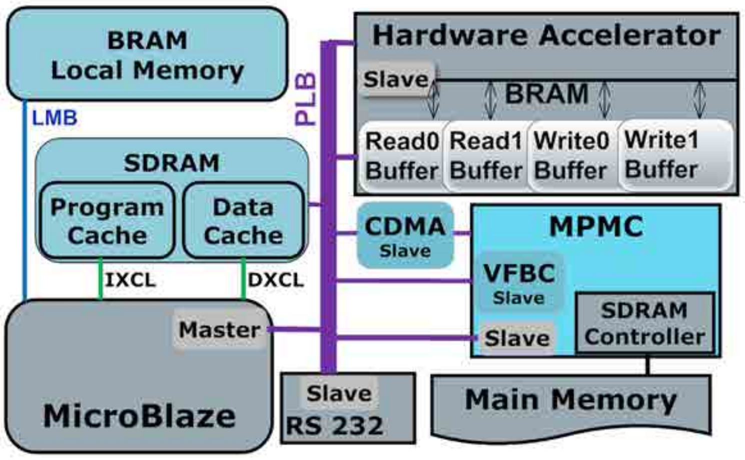


Figure 8.2: APMC: Data Manager: (a) Load, Update & Reuse Units

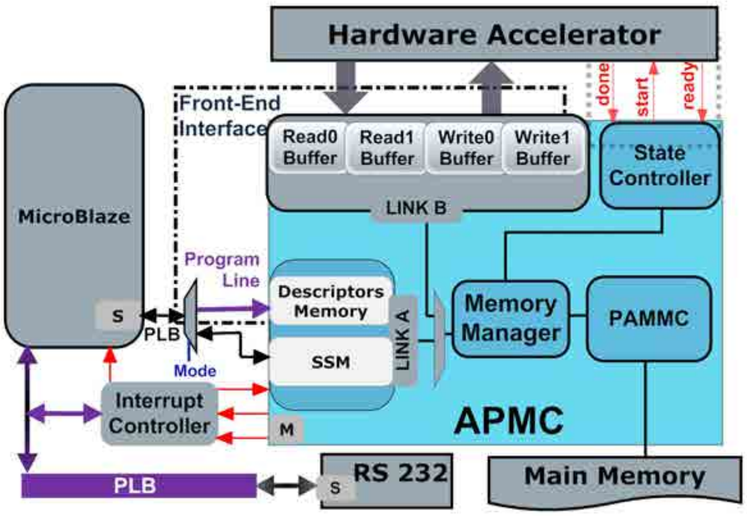
in Figure 8.2 compares the requested elements with the elements placed in the *reuse unit*. If the elements are found in the *reuse unit*, the *Data Manager* uses them again and requests the rest. The *update unit* transfers the elements addresses which are not present in the *reuse unit* to the *Main Memory System*. The *update unit* of the *Data Manager* rearranges available elements in the *SSM* and updates new loaded elements.

8.2 Experimental Setup

In this section, we describe the APMC based system and the applications with regular and irregular memory access patterns used to evaluate it. In order to evaluate the performance of the APMC system, the results are compared with a baseline system having the same compute (e.g. MicroBlaze and Hardware Accelerators) and bus units but a different memory access controller. The Xilinx Integrated Software Environment and Xilinx Platform Studio are used to design the systems. Xilinx Software Development Kit and Xilinx Power Estimator are used to implement the software application and analyze the system on-chip power respectively. A serial terminal (RS232) is used to debug, initialize data sets and display clock cycles consumed by the systems. We use a Xilinx Virtex-5 ML505 evaluation FPGA board to test the APMC system. To compare the bandwidth and power of the APMC system, an Altera Scatter-Gather DMA (SGDMA) based system is used. The SGDMA system is tested on an Altera Stratix-IV FPGA based development kit.



(a)



(b)

Figure 8.3: (a) MPMC based System Architecture (b) APMC based System Architecture

8.2.1 Baseline MPMC System

A Xilinx FPGA based state of the art High Performance Multi-Port Memory Controller (MPMC) system is used (Figure 8.3 (a)) as a baseline. The architecture has 64 kB and 4 kB of data and instruction cache respectively. The design uses Xilinx Cache Links (IXCL/DXCL) for I-Cache and D-Cache memory accesses. MPMC is a fully parameterizable memory controller, providing an efficient interfacing between the processor

	Regs	LUTs	BRAMs
MPMC			
Cache Manager & Memory	1804	1679	42
SDRAM Controller (PLBV46)	3488	2092	5
CDMA (Optional)	566	768	
VFBC (Optional)	1940	1670	
MicroBlaze Core	1400	1360	
Timer	358	287	
RS232	144	123	
Total (without CDMA & VFBC)	7069	5220	47
APMC			
Data Manager & Specialized Memory	1292	1013	36
DRAM Memory Access Unit	2679	1870	-
MicroBlaze Core	1400	1360	
Timer	358	287	
RS232	144	123	
Total	5873	4653	36

Figure 8.4: MPMC & APMC Systems Resource Utilization

and SDRAM using the IBM CoreConnect Processor Local Bus (PLB) bus interface. By default, MPMC uses the PLB interface to transfer data from/to SDRAM. PLB supports fixed-burst data transfers, and 8-word cache line read/write transfers. The MPMC controller (like other memory controllers) takes data transfer instructions from the MicroBlaze processor and performs memory operations. A modular DDR2 SDRAM [48] controller (with the PLBv46 Wrapper) is used with the MPMC system to access main memory. The DDR2 controller has a peak main memory bandwidth of 800MB/s as it has a clock frequency of 100MHz, and a data bus width of 64 bits. Figure 8.4 shows the resources of the MPMC system. The MPMC system can also use Video Frame Buffer Controller (VFBC) and Central Direct Memory Access (CDMA) direct memory access controllers. The VFBC and CDMA are optional and are used to improve the performance of the DDR2 SDRAM controller by managing complex patterns in hardware. For each application, the controller (VFBC or CDMA) that better suits the application access pattern is included in the design.

8.2.2 APMC based System

The APMC based system is shown in Figure 8.3 (b). Memory components are similar to the baseline system. The major difference between APMC and the baseline system is that the APMC uses the specialized memory and manages data transfers without support of the processor core. MicroBlaze works as a slave in the APMC system and does not have local memory nor cache memories. It is used only to initialize and program the *Descriptor Memory*. The working operation of the APMC system is subdivided into two modes using the *Mode* select line: the *Program mode* and the *Execution mode*. During the *Program mode* the APMC is attached only to the MicroBlaze processor. MicroBlaze via the *Program Line* initializes the *Descriptor Memory* of the APMC system. During the *Execution mode*, APMC is connected with the compute unit and performs memory management. The computation part is executed either by

Table 8.1: Application kernels with Regular access patterns

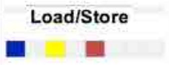
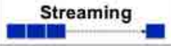
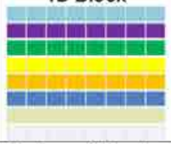
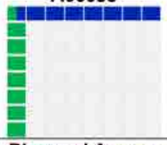
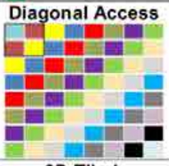
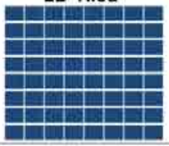
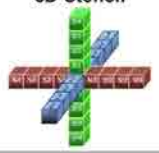
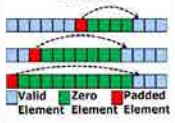
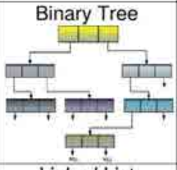
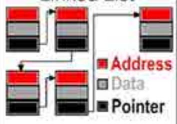
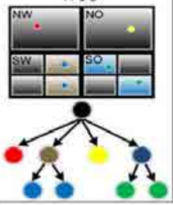
Kernel	Description	Access Pattern	Reg, LUT	GFLOPS
Rad_Con	Radian Converter converts degree into radian	Load/Store 	68, 67	0.375
Thresh	Thresholding is an application of image segmentation, which takes streaming 8-bit pixel data and generates binary output.		2289, 2339	0.125
FIR	Finite Impulse Response calculates the weighted sum of the current and past inputs.	Streaming 	3953, 2960	3.875
FFT	Fast Fourier Transform is used for transferring a time-domain signal into corresponding frequency-domain signal.	1D Block 	4977, 2567	6.0
Mat_Mul	Matrix Multiplication takes pair of tiled data and produce Output tile. Output= Row[Vector] × Column[Vector] X=Y×Z	Column & Vector Access 	2925, 1719	7.750
Smith_W	Smith-Waterman determines the optimal local alignments between nucleotide or protein sequences.	Diagonal Access 	3380, 2616	1.125
Lapl	Laplacian kernel applies discrete convolution filter that can approximate the second order derivatives.	2D Tiled 	6977, 5567	2.125
3D-Sten	3D-Stencil algorithm averages nearest neighbor points (size 8x9x8) in 3D.	3D Stencil 	6205, 2853	4.625

Table 8.2: Application kernels with Irregular access patterns

Kernel	Description	Access Pattern	Regular %	Irregular % known	Irregular % unknown
CRG	A compression algorithm, hides zero in a descriptor block	Pointer 		28	72
Huffman	Huffman is an entropy coding technique. Allocate codes to symbols, using frequency of occurrence for each symbol.	Binary Tree 		25	75
In_Rem	A Linked List Buffer	Linked List 	5	55	40
N-Body	The 3D-Hermite algorithm used to compute movement of bodies using the newtonian gravitational force.	Tree 	20	40	40

(b)

MicroBlaze or a hardware accelerator. APMC allocates a separate *Descriptor Memory* for each *compute unit*. At run-time, APMC takes memory requests from the *compute unit* and transfers data patterns to its *SSM*. Separate data links (Link A (PBI) and Link B (SSSI)) are used to transfer data of the processor and the accelerator. The resource utilization of the APMC system is shown in Figure 8.4.

8.2.3 Test Applications

The application kernels executed with regular access patterns are shown in Table 8.1. *Access pattern* presents the application's memory accesses. Each color represents a different memory access sequence and pattern. *Reg, LUT* describes the slices utilized by hardware accelerators over Virtex-5 devices. The application kernels having irregular access patterns are shown in Table 8.2. *Regular & Irregular* describes the percentage of data access and class of the application kernels. The application kernels with regular and irregular access patterns are executed on separate *ASHA* generated by ROCCC and MicroBlaze *SSP* respectively. In our current designs, *ASHA* are executed at 100 Mhz of clock frequency and the highest bandwidth required is 400 MB for *c/a=1*.

8.3 Results and Discussion

This section analyzes the results of different experiments conducted on the APMC and MPMC systems. The experiments are classified into three subsections: *Application Performance*, *Bandwidth* and *Power*.

8.3.1 Application Performance

All application kernels are executed on the MPMC and APMC based systems. Figure 8.5 shows the execution time (clock cycles) for regular and irregular application kernels. The X and Y axis represent application kernels and clock cycles, respectively. Regular and irregular application kernels have different vertical logarithmic scales. Each bar represents the application kernel's computation time and memory access time (lower is better). The APMC memory access bar is further divided into two segments: one shows the results of the default APMC with the *Data Manager* and the other shows the additional cycles spent when the *Data Manager* is disabled.

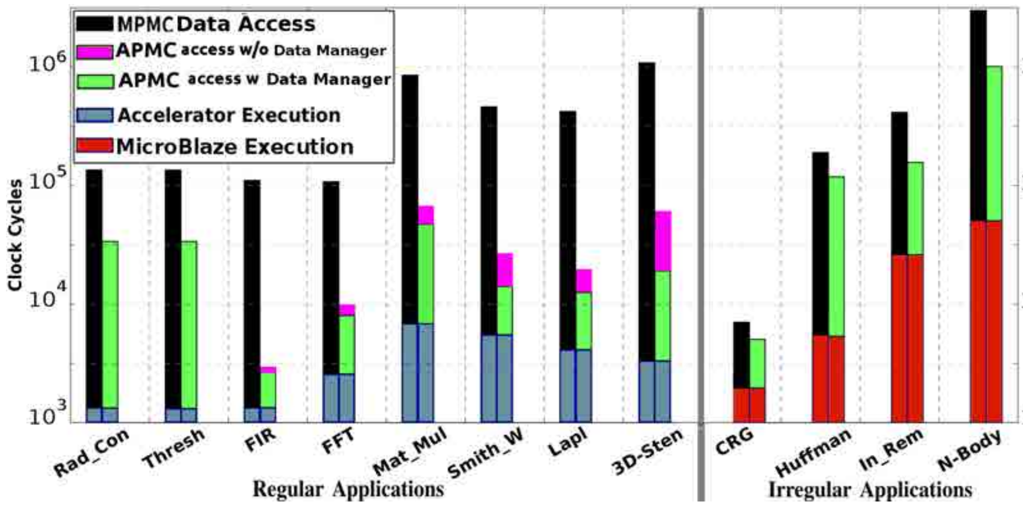


Figure 8.5: Application Execution Time

By using the APMC system, the results show the *Rad_Con*, and *Thresh* applications achieve 3.5x of speed-up over the MPMC based system. These application kernels have several memory access requests with no data locality which requires multiple descriptors and no use of on-chip *SSM*, and achieve less speed-up compared to other application kernels. *FIR* has a streaming data access pattern and achieves 26.5x of speed-up. The APMC requires only one descriptor to access a stream. This reduces the address generation/management time and on-chip request/grant time. *FFT* reads a 1D block of data, processes it and writes it back to main memory and achieves 11.9x of speed-up. The *Mat_Mul* kernel accesses row and column vectors and attains 14x speed-up. The APMC system manages addresses of row and column vectors in hardware. The *Smith_W* and *Lapl* applications take 2D blocks of data and achieve 38x and 36x of speed-up respectively. The *3D-Stencil* data decomposition achieves 52x of speed-up. The APMC takes 2D and 3D block descriptors and manages them in hardware. The speed-ups are possible because APMC can manage complex access patterns with a single descriptor. The *FIR* and *FFT* applications require a single descriptor for data transfer whereas *Mat_Mul*, *Smith_W* and *3D-Stencil* take more than one descriptor. The MPMC system uses CDMA for *FIR*, *FFT* and *Mat_mul* applications, which allows full-duplex, high-bandwidth, bus interfaces into memory. The VFBC is used to transfer a complete tile/frame for *Smith_W* and *3D-Stencil* applications to/from main memory. VFBC is a two-dimensional DMA core that has high latency but high throughput operation for very long bursts. At run-time, APMC takes several of these descriptors independently and manages them in parallel, whereas the MPMC is dependent on the MicroBlaze processor that feeds data transfer instructions. The stand-alone working operation of APMC removes the overhead of processor/memory system request/grant delay.

The *c/a* ratio improvement due to the APMC *Data Manager* unit is presented in Table 8.3. Not all the applications improve *c/a* ratio due to data irregularity or locality issues. Due to the load/store memory accesses with no data locality the performance of *Rad_Con* and *Thresh* applications do not benefit from the *Data Manager*, as Figure 8.5 shows. While executing *FIR*, *FFT*, *Mat_Mul*, *Smith_W*, *Lapl* and *3D-Stencil* applications on the APMC system, the system achieves 1.15x, 1.11x, 1.25x, 1.28x, 1.23x, and 1.97x of additional speed-ups respectively from the *Data Manager*. For regular access patterns both cache and APMC data manager improve *c/a* ratio. The data manager also

Table 8.3: APMC: $\text{Computed}_{Point}/\text{Accessed}_{Point}$ Ratio

c/a Ratio	FIR	Mat_Mul	Lapl	3D-Sten
APMC Without Data Manager	0.0078	0.01538	0.1	0.04
APMC with Data Manager	1	0.031	1	0.125

improves *c/a* ratio for applications with complex and dense access pattern such as the 3D-Stencil, column vector or diagonal vector access patterns.

The *CRG*, *Huffman*, *In_Rem* and *N-Body* applications have irregular memory patterns; therefore, these applications are executed on the MicroBlaze soft processor. The *CRG* and *Huffman* applications have long *unknown* memory access patterns with no data locality. While executing these applications on the APMC system, the system achieves 1.4x, 1.5x of speed-up respectively over the MPMC system. The *In_Rem* application has both *known* and *unknown* memory access patterns with no data locality hence achieves 1.9x of speed-up. The *N-Body* application includes irregular data patterns with data locality. While running on the APMC system, it achieves 2.9x of speed-up over the MPMC system. The APMC system utilizes the *SSM* and the *Data Manager* that reduces processor execution and access time (shown in Figure 8.5).

One reason irregular applications achieve less speed-up than regular applications on the APMC system is that there are still on-chip communication delays due to the use of the microprocessor and compute unit. We measure the impact of this delay by integrating a dummy hardware accelerator which removes on-chip communication delay. The accelerator sends random memory accesses. Each memory access is fully dependent on a previously loaded value which is *unknown* at compute-time (last accessed data defines the next address, `val=array[val]`). The experiments show that as a result of random patterns executed by the hardware accelerator and the MicroBlaze processor on the APMC system, the APMC system achieves 5.7x and 1.1x of speed-up respectively compared to the MPMC system using the MicroBlaze processor. The speed-up can further be improved by executing real application kernels on hardware accelerators with non-random accesses, which helps to apply effectively memory access scheduling and prefetching.

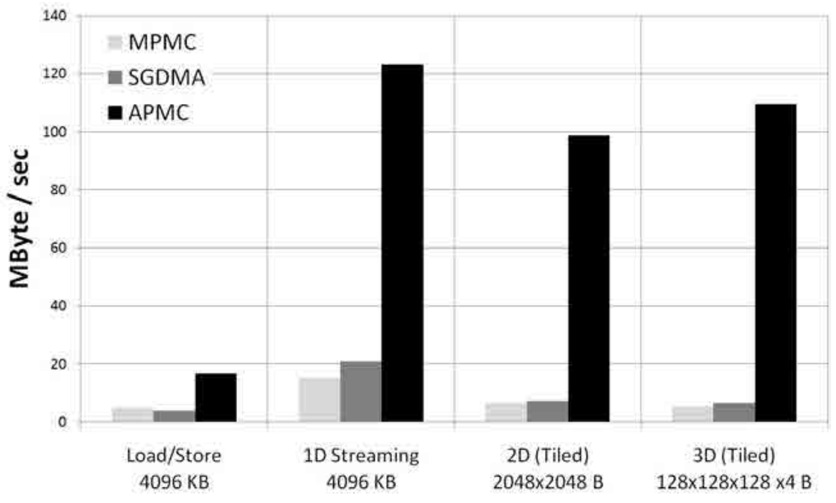


Figure 8.6: Dataset Read/Write Bandwidth

8.3.2 Bandwidth

In this section we compare the APMC, SGDMA and MPMC system's bandwidth. We use Xilinx Virtex-5 to implement APMC, MPMC systems and Altera Stratix-IV FPGAs to implement the SGDMA system as one of the most advanced memory controller available for FPGA. Both systems are connected to a 256 MByte (32M x 16) of DDR2 SDRAM having SODIMM I/O module operating at 100 MHz clock. Figure 8.6 shows data transfer bandwidth for APMC-, SGDMA- and MPMC-based systems. The X-axis presents random load/store, streaming, 2D and 3D tiled data sets of 32 MByte that are read and written from/to the main memory. The Y-axis shows bandwidth in MBytes per second (higher is better). The load/store access pattern reads and writes 4 Bytes from random locations. A single streaming access pattern accesses 1KByte of stream and 2D access pattern reads/writes 2D blocks with row and column size of 1KByte. The 3D tiled reads 128x128x32 Bytes of 3D tile (rows, column and plane) and wrote it back to main memory. The results show that APMC random load/store and streaming data transfers are 3.4x and 8.3x times faster respectively than the MPMC system. APMC gets improvement by utilizing on-chip and off-chip bus interconnects and controls data transfer without the support of a master processor. The data transfer rate of APMC system is further improved up to 15.3x and 20.4x while transferring 2D and 3D tiled data respectively. The APMC improves the memory bandwidth by transferring descriptors to the memory controllers, rather than individual references that utilize the open banks of SDRAM device. While comparing with SGDMA results show that APMC achieves 4.3x, 5.9x, 13.3x and 16.4x of speed-ups for load-store, 1D, 2D, and 3D data accesses respectively. However, SGDMA data transfer uses a microprocessor to fill the descriptors and forces to follow bus protocol. The APMC manages complex patterns independently and uses few descriptors that reduce run-time address generation and address request/grant delay. The APMC 3D tiled transfer is faster than 2D because 3D tiled requires data from multiple banks and the APMC accesses the data from multiple banks in parallel.

8.3.3 Power

On-chip static power in a Xilinx Virtex-5 device dissipates 2.75 watts while running the MPMC based system. The APMC system draws 2.1 watts of on-chip static power on a V5-Lx110T device. While comparing APMC and MPMC systems with the MicroBlaze processor and without accelerators, results also indicate that the APMC system consumes 17% fewer slices and 32% less on-chip power than the MPMC system. APMC provides low-power and simple control characteristics by rearranging data accesses and utilizing hardware units efficiently.

8.4 Conclusion

HPC applications with irregular memory access patterns suffer from limited performance on FPGA based systems. In this work we have presented PMC single core system that supports both regular and irregular memory access patterns, called APMC. The APMC system uses an on-chip memory unit (for 1D, 2D and 3D data sets), efficient memory management and a memory access policy to access DRAM. The APMC *Scheduler* handles single core (ASHA or SSP) one at a time using *FIFO* scheduling policy. The *Address Manager* improves system performance by accessing regular and irregular access patterns and allocating them in *SSP* without processor intervention. Memory patterns are arranged in *descriptors* at program-time, at run-time APMC accesses them without adding memory request and address generation delay and places them in on-chip *SSM*. The *Address Manager* handles *dependent unknown irregular* accesses at run-time using *Control Bus*. The *Data Manager* efficiently access, reuse, align and feed data to single core system. In order to prove the effectiveness of the proposed controller, we implemented and tested it on a Xilinx ML505 FPGA board. The experimental evaluations based on the Xilinx MPMC coupled with the MicroBlaze processor demonstrate that the APMC system efficiently accesses irregular memory patterns in addition to regular access patterns. In order to prove that our controller is efficient in a variety of scenarios, we used several benchmarks with different memory access patterns. The benchmarking results show that our controller consumes 17% less hardware resources, 32% less static power and achieves a maximum speedup of 52x and 2.9x for regular and irregular applications respectively.

Part V

Multi-core Memory System for Irregular Data Pattern

9

AMMC: Advanced Multi-core Memory Controller

Latest multi-core architectures require both programmability and performance and combine different types of cores, becoming heterogeneous systems. To get programmability, a part of the program is executed on general-purpose cores. To achieve performance and to increase power efficiency, compute intensive tasks are mapped into separate hardware accelerators or application-specific processors. The dedicated application specific accelerator cores have low footprint and low power consumption and feature high performance [92].

Many data intensive applications running on heterogeneous multi-core systems are by nature irregular. They may present irregular data structures, irregular control flow or irregular communication. Current hardware accelerator-based multi-core systems are designed to access regular streaming data. Executing irregular applications on them requires a separate control unit (a master processor) which accesses irregular data, and feed it to processing cores in regular format. The master processor core performs initial data processing and data management before the compute-intensive tasks are off-loaded. This needs substantial effort, and often leads to poor performance.

To overcome the memory wall and to reduce the system power, a memory system is needed that supports low frequency, low complexity cores, has efficient local memory and data management, with an intelligent scheduler while supporting a programming model that manages memory accesses in software so that hardware can best utilize

[1] Chapter 9 is based on the publication : [44].

them. In this work, we integrated PMC with a heterogeneous multi-core system having *ASHA* and *SSP*, that we term AMMC (Advanced Multi-core Memory Controller).

Some salient features of the proposed AMMC architecture are given below:

- The AMMC based system handles heterogeneous (*SSP* and *ASHA*) cores using *Symmetric* and *Asymmetric* scheduling policies, without the support of a master core and operating system.
- Regular and irregular access patterns of heterogeneous multi-cores are described using a separate *Descriptor Memory*, which reduces the on-chip communication time and run-time address generation overhead.
- The AMMC *Address Manager* and *Scheduler* handles regular and irregular pattern requests of a heterogeneous multi-core system, provides precise timing and allows scheduling mode to be changed at runtime.

9.1 AMMC Architecture

In this section, we describe the Advanced Multi-core Memory Controller (AMMC) system. The architecture (shown in Figure 9.1) is divided into five units: the *Bus System* (A), the *Local Memory Unit* (B), the *Memory Manager* (C), the *Scheduler* (D) and the *SDRAM Controller* (E).

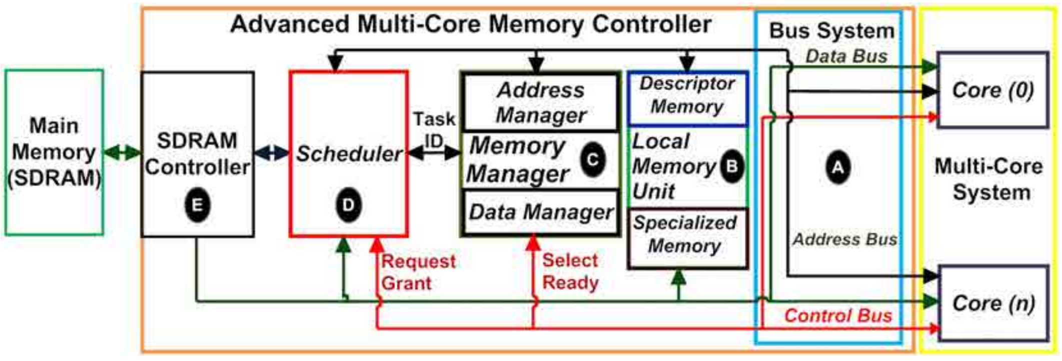
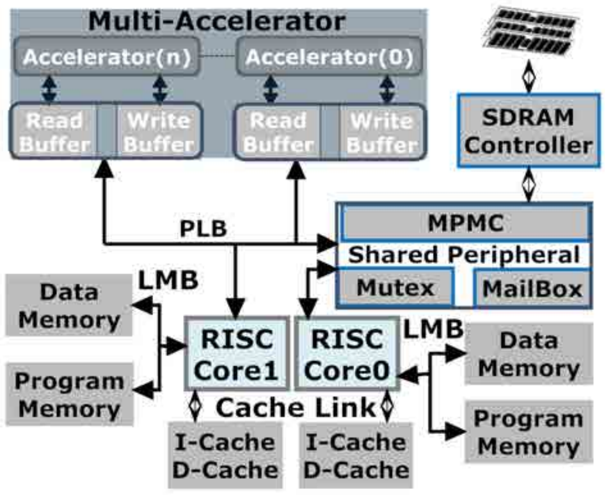


Figure 9.1: Architecture of Advanced Multi-core Memory Controller

Overview of AMMC: The regular and irregular access patterns are the same as described in Chapters 3 and 8. The basic structure of the multi-core system is based on what is described in Chapters 7 and 6, with the difference the current architecture support both *ASHA* and general purpose *SSP*, for execution of irregular kernels. The main units of AMMC are shown in Figure 9.1, as well as the *Multi-core System*, that executes the applications. The *Multi-core System* can have general purpose *SSP*, *ASHA* cores or a combination of both types. The *Bus System* provides *PBI* and *SSSI* links between AMMC and the *Multi-core System* having *ASHA* and *SSP* cores. The AMMC *Memory Unit* stores both data and the access pattern descriptors in *SSM* and *Descriptor Memory* respectively. Each processing core has separate *SSM* and *Descriptor Memory* blocks. The descriptors are programmed at compile-time with memory access patterns and scheduling priority. At run-time, the AMMC *scheduler* receives multiple memory read/write requests from the *Multi-core System* and selects a processing core, depending upon its priority level and scheduling policy. The *Scheduler* implements both `Symmetric` and `Asymmetric` scheduling policies for *ASHA* and *SSP* cores. The *scheduler* forwards the memory request to the *Memory Manager*. It is divided into the *Address Manager* and the *Data Manager*. The *Memory Manager* takes single or multiple descriptors and reads/writes the data pattern. The *Address Manager* takes *Local Memory* address (*Task ID*) of *ASHA SSP* from the *scheduler* and fetches its *Descriptor Memory*. Depending on the access pattern the address manager uses single or multiple descriptors, maps and rearranges addresses in hardware. The *Address Manager* saves mapped addresses into its *Address Buffer* for further reuse. The *Main Memory System* is responsible for transferring data between main memory and the specialized memory. The *Main Memory System* take memory addresses from the *Memory Manager*, performs the address mapping from physical address to the memory address and reads/writes data to/from the *Main Memory*. The memory address holds the memory bank, the row address, column address and chip select. The AMMC *Main Memory System* gathers multiple memory requests, manages SDRAM banks with respect to physical (SDRAM) addresses and maximizes the reuse of open SDRAM banks.



(a)

MicroBlaze System	Flip Flops	LUTs	BRAMs
Master Core	2606	2664	14
Timer	358	287	
Mutex Core	145	95	
MailBox Core	350	317	3
Bus System	145	342	
SDRAM Controller	4162	2719	13
Total	7766	6424	30

(b)

Figure 9.2: MicroBlaze-based multi-core system: (a) Block Diagram (b) Resource Utilization

9.2 Experimental Framework

In this section, we describe the MicroBlaze- and AMMC-based *Multi-core Systems* and the rest of our experimental setup. A Xilinx ML505 evaluation FPGA board is used to test the *Multi-core Systems*. The Xilinx Integrated Software Environment and Xilinx Platform Studio are used to design the *Multi-core Systems*. Xilinx Power Estimator does the power analysis. The section is divided into two subsections: the *MicroBlaze based Multi-Core System* and the *AMMC based Multi-Core System*.

There are two type of cores in our heterogeneous system: *ASHA* cores execute application kernels with regular memory accesses while *SSP* MicroBlaze cores execute application kernels with irregular memory access patterns. As has been described in Section 2.1.3.3, a state controller is used to manage multiple buffers (see Figure 2.11 (a)). Most cores integrate a *state controller* to manage multiple buffers. Tables 8.18.2 list all applications used in our experiments. The exception is when applications running on them use a single buffer (*Rad_Con*, *Thresh*, *CRG* and *In_Rem*). In our experiments, *CRG* and *In_Rem* are executed in the same MicroBlaze core, which does not have a state controller.

9.2.1 MicroBlaze-based Multi-Core System

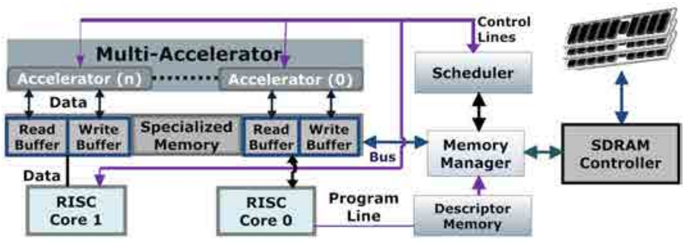
The MicroBlaze-based *Multi-core System* is used as baseline (Figure 9.2 (a)). The detailed design summary of the MicroBlaze based multi-core system is shown in Figure 9.2 (b). Each general purpose core has 16KB and 32KB of instruction and data cache respectively, and that is implemented using BRAM. MicroBlaze instruction prefetcher improves the system performance by using the instruction prefetch buffer and instruction cache streams. A MicroBlaze *SSP* (Core 0, Figure 9.2 (a)) is the master core and is used to schedule the memory requests and to manage data transfers between multi-cores and main memory (SDRAM). We use IBM CoreConnect Processor Local Bus (PLB) [20] configured for a single master and multiple slaves to connect computation units and shared peripherals. It provides maximum of 2 GByte of bandwidth while operating at 125MHz and 128-bit width. The Mutex core (shown in Figure 9.2 (a)) is used to provide synchronization when accessing shared resources. The core has a configurable number of mutexes and has a write to lock scheme. The Mailbox core uses interrupt line to pass messages between *Multi-core System*. The core handle interrupts in FIFO order.

To access data from main memory *MPMC* is employed. The *MPMC* connects the SDRAM with the MicroBlaze processors using PLB. An SDRAM (DDR2) controller is used with *MPMC* to access data from (SDRAM) main memory. The supported DDR2 memory has a peak bandwidth of 1 GByte/s as it has a clock frequency of 125MHz, and a data bus width of 64 bits.

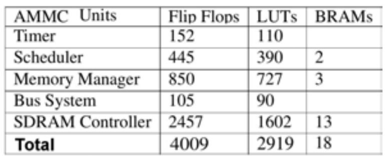
The MicroBlaze cores use Xilkernel [77] that performs scheduling, inter-process communication and synchronization with *POSIX* threads (pthreads). From the main function, application spawns into multiple statically declared threads using the pthread library. Each thread controls a single application kernel and manages its memory patterns. We use the Xilinx SDK to compile the application kernels using a library generator (libgen) and a MicroBlaze-specific gcc/g++ compiler and generate the final object file.

9.2.2 AMMC based Multi-Core System

Figure 9.3 (a) shows the implementation of an AMMC-based *Multi-core System*. General purpose cores do not integrate any cache but use the local memory provided by



(a)



AMMC Units	Flip Flops	LUTs	BRAMs
Timer	152	110	
Scheduler	445	390	2
Memory Manager	850	727	3
Bus System	105	90	
SDRAM Controller	2457	1602	13
Total	4009	2919	18

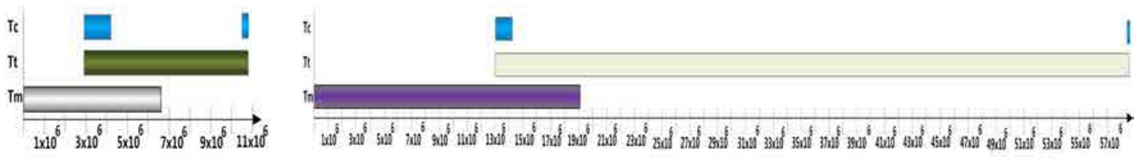
(b)

Figure 9.3: AMMC-based multi-core system: (a) Block Diagram (b) Resource Utilization

AMMC. Similarly, there is no need for an RTOS like Xilkernel. In the current implementation of AMMC, on a Xilinx ML505 evaluation FPGA board, a 256 MByte (32M x 16) of DDR2 memory having SODIMM I/O module is connected with AMMC *Main Memory System*. The resource consumed by each AMMC unit is shown in Figure 9.3 (b). The *Main Memory System* has a peak bandwidth of 1 G Byte/s since it has a clock frequency of 125 MHz, a data rate of 2 words per clock cycle, and a data bus width of 32 bits.

9.3 Results and Discussion

This section analyzes the results of experiments conducted on AMMC and MicroBlaze based systems. The experiments are characterized into three subsections:*System Performance*, and *Area & Power*.



(a) (b)

Figure 9.4: Symmetric System Performance: (a) AMMC (b) MicroBlaze Pipeline and Overlap Time Period

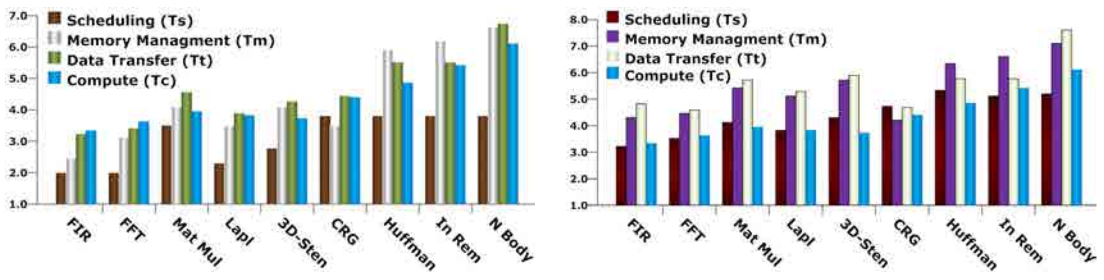
Table 9.1: Asymmetric Scheduling Priority Policies

Kernels	FIR	FFT	Mat_Mul	Lapl	3D-Sten	CRG	Huffman	In_Rem	N-Body	Speed-ups
Symmetric	I	1	1	1	1	1	1	1	1	5.47x
Asymmetric										
Group I	1	4	5	3	2	6	7	8	9	6.84x
Group II	2	3	4	5	1	8	6	9	7	5.83x
Group III	9	6	5	4	8	4	3	2	1	3.45x
Architecture	1	1	1	1	1	2	2	2	2	5.42x

9.3.1 Multi-Core System Performance

The system performance is measured by executing application kernels simultaneously using different scheduling policies, on AMMC and MicroBlaze based systems. Due to the confined FPGA resources, 5 hardware accelerator and 2 processor cores are integrated with the *Multi-core System*. The execution time of both systems is categorized into four factors: scheduling time (*Ts*), memory management time (*Tm*), data transfer time (*Tt*) and computation time (*Tc*). *Ts* holds the arbitration (request, grant and wait) time among the on-chip scheduling. *Tm* defines the address generation and data management time. *Tt* presents the data access time from external memory. It includes address mapping from physical address space to SDRAM address space, interface timing and synchronization. *Tc* holds the computation time of the application kernels. To measure the overlap and processing time, each application kernel is assigned four timers which count *Ts*, *Tm*, *Tt* and *Tc* clock cycles. While counting the number of cycles, each timer counts the other working timers that are used to measure pipeline and overlap time.

In the *Symmetric Scheduling* policy, the requests are treated with the FIFO method, which removes the scheduling time. Figures 9.4 (a) and (b) present the overlapped / pipelined time of AMMC and MicroBlaze systems respectively. While running the *Multi-core System* using symmetric scheduling, the results show that the AMMC system achieves 5.47x of speed-up. The current *Multi-core System* contains application kernels with different access patterns. The *Symmetric Scheduling* policy gives higher priority to application kernels with many memory requests. These requests add on-chip bus and memory access delays therefore, AMMC system is not fully overlap *Tm* & *Tt*. These delays can be decreased by executing *Multi-core System* with *Asymmetric Scheduling* policy.



(a) (b)

Figure 9.5: Asymmetric System Applications Performance: (a) AMMC (b) MicroBlaze

To operate the *Multi-core System* simultaneously, we categorize the *Asymmetric Scheduling* policy into two types; the memory access (*Mem Acc*) based asymmetric policy and the architecture based asymmetric policy (shown in Table 9.1). The memory access based asymmetric policy assigns priorities to the application kernels with respect to their access patterns and is further categorized into three groups. In Group *I*, the highest priorities (`1`) are allocated to application kernels having fewer memory requests and dense access patterns. For example, the applications having multiple read/write requests are given low priorities. To check the sensitivity of *Asymmetric Scheduling* execution, the priorities of Group *I* are slightly varied in Group *II*. In Group *III*, the priorities are assigned to check the critical performance of asymmetric execution. For example, the highest priority (`1`) is allocated to application kernels having maximum memory requests. Like MicroBlaze Xilkernel scheduling model, the AMMC scheduling policies and memory accesses are configured statically at program-time and are executed by hardware at run-time. The memory access based asymmetric policy performs load balancing reduces on-chip communication and memory management delay.

Figures 9.5 (a) & (b) present clock cycles, while executing application kernels simultaneously using memory access based *Asymmetric Scheduling* policy. X (logarithmic scale) and Y axis present clock cycles and application kernels, respectively. Each bar represents *Ts*, *Tm*, *Tt* and *Tc*. While running all application kernel together using the asymmetric scheduling, the results show that the scheduling, memory manager and memory transfer of AMMC based system are 21x, 2.9x and 7.1x faster respectively, compared to the MicroBlaze based system. The computation units execution time (*Tc*) remains the same for both systems. Figures 9.6 (a) and (b) present the overlapped/pipelined time of AMMC and MicroBlaze systems respectively. The *Tc* of all application kernels is overlapped (shown in Figure 9.6 (a) & (b)). In the AMMC system, *Tt* and *Tm* are dominant for the regular and irregular application kernels respectively. As all AMMC units operate in parallel, AMMC overlaps all other units under the unit that consumes more time. For example for regular application kernels *Ts*, *Tm* and *Tc* are overlapped under *Tt*. The MicroBlaze based system overlaps *Tc* & *Ts* completely and partially overlaps *Tm* and *Tt* (shown in Figure 9.6 (b)). While running all application kernel together using the *Asymmetric Scheduling* with priorities of Group *I*, the results show that the AMMC based system achieves 6.84x of speed-up compared to MicroBlaze based system. While executing application kernels with priories of Group *II* and Group *III*, the AMMC based system achieves 5.83 and 3.45x of speed-ups respectively. The AMMC *Asymmetric Scheduling* policy manages system resources (Application code, On-Chip Off-Chip Memory) of *Multi-core system* without the support of the operating system.

In architecture based asymmetric policy, the processing cores are assigned priorities depending upon their instruction set architecture, execution and communication (request/grant) speed. The architecture based asymmetric priorities are shown in Table 9.1. All the cores of one type get the same priority. The priority `1` executes hardware accelerator core requests with higher priority. Requests having same priorities are executed in FIFO order. While running *Multi-core Systems* using the architecture based *Asymmetric Scheduling* policy, the results show that the AMMC based system achieves 5.42x of speed-up compared to MicroBlaze based system. For performance evaluations, we analyzed that the priority based scheduling has the potential for sup-

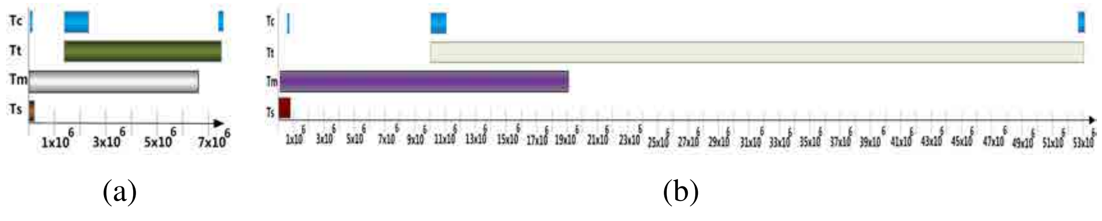


Figure 9.6: Asymmetric System: (a) and (b) AMMC and MicroBlaze Systems Pipeline and Overlap Time Period

porting scalability and load balancing and improve performance while assigning priorities according to the characteristics of memory access patterns.

9.3.2 Area & Power

In comparison, on-chip power in a Xilinx V5-Lx110T device dissipates 3.15 watts while running the MicroBlaze based system. The AMMC system draws 2.27 watts of on-chip static power on a V5-Lx110T device. While comparing the AMMC and MicroBlaze systems without slave units (accelerators and processor), results show that AMMC system consumes 21% fewer slices and 27.9% less on-chip static power than the MicroBlaze system. The AMMC provides low-power and simple control characteristics by rearranging data accesses and utilizing hardware units efficiently.

9.4 Conclusion

In this work, we have proposed a version of PMC controller called AMMC, that schedules multi-core operations while taking processing, scheduling, memory management and memory transfer into account. AMMC has been coupled with *ASHA* and *SSP* based heterogeneous system. The AMMC *Descriptor Memory* and *Address Manager* improves the system performance by reducing address management time of regular and irregular patterns. The AMMC system holds information of each processing core's access patterns in a separate *Descriptor Memory*. The AMMC *Address Manager* supports *regular*, *irregular known*, *independent unknown irregular* and *dependent unknown irregular* memory accesses that eliminates the overhead of arranging and gathering address/data by the master cores (i.e. microprocessor). The *Scheduler* manages complex memory patterns using *Symmetric* and *Asymmetric* scheduling policies. The AMMC system is implemented and tested on a Xilinx ML505 evaluation FPGA board. The performance of the system is compared with a microprocessor based system that has been integrated with the Xilkernel operating system. Results show that the AMMC based multi-core system consumes 21% fewer hardware resources, 27.9% less on-chip power and achieves 6.8x of speed-up compared to the MicroBlaze-based multi-core system.

10

MAPC: Memory Access Pattern based Controller

The shared communication buses of heterogeneous systems are problematic both in latency and bandwidth. A shared bus has long electrical wires, and if there are several potential slave units – in a multi-core processor all cores and the memory sub-system act as a master and slave respectively – the load makes the bus even slower. Furthermore, the fact that several units share the bus fundamentally limits the bandwidth seen by each core. When the number of components attached to the bus increases, the physical capacitance on the bus wires grows, and, as a result, its wiring delay grows even further. The multi-core bus unit also suffers from on-chip and off-chip bus interconnects and handles data transfers that require a complex bus matrix architecture, direct memory controllers and master core which manages data movements. The bus faces delays such as master/slave arbitration and bus switching time.

The latencies generated by the memory operations are determined by the memory access time, which is smaller than the processor clock cycle time. As the number of processor memory requests increases the latency of memory accesses also increases. With the increase of memory capacity, the memory access time also increases due to address decoding, internal delays in driving long bit lines, selection logic and the need to use a small amount of charge per bit. Integrating more memory controllers [73] on a system platform can increase bandwidth but requires an intelligent controller that manages and schedules the data accesses. Results have shown [35] that management

[1] Chapter 10 is based on the publication : [45] and [46].

of data transfers according to the application specified patterns reduces the bus delays and memory access latencies and improves the system performance.

In this chapter, we evaluate the PMC multi-cores with a run-time scheduling policy and run-time pattern manager called Memory Access Pattern based Controller (MAPC). The MAPC *Address Manager* (Section 2.1.3.1) handles multi-core irregular unknown access patterns and arranges them at run-time in the *Descriptor Memory*. MAPC uses access pattern descriptors to program, manage and execute data transfers and provides all features described in Chapter 2. MAPC is based on three major approaches:

- Compile- and run-time data pattern management.
- Run-time rearrangement and prioritization of patterns.
- Management of SDRAM rows/banks based on access patterns.

The MAPC *Pattern Descriptor Unit* manages complex memory patterns into its *Descriptor Memory* at compile- and run- time. The *Pattern Descriptor Unit* improves the bandwidth by transferring access pattern descriptors to the memory controller, rather than individual references. The MAPC *Pattern Scheduler* applies a run-time data access prioritizing policy that rearranges access patterns according to data transfer request and size. The *Pattern Aware Main Memory Controller* decodes access pattern descriptors and manages DRAM open banks and rows with respect to the access pattern.

The main contribution of this chapter is a mechanism that reduces the impact of regular and irregular memory access management and transfer time for a multi-core system by applying run-time scheduling policy. To achieve this, we also propose a *Pattern Descriptor Unit* that organizes data accesses into patterns at compile- and run-time. Moreover, we propose a *Pattern Scheduler* that applies fair data transfer policy on applications having huge transfer requests or size and improve fairness for data dependent and time critical execution. The *Pattern Scheduler* prioritizes access patterns, helping MAPC to utilize SDRAM open banks and rows. The design uses a *Pattern Aware Main Memory Controller* that efficiently accesses data from SDRAM single- or multi- banks. The experimental results show that MAPC transfers different data sets up to 1.95x faster than the baseline system with a conventional but state of the

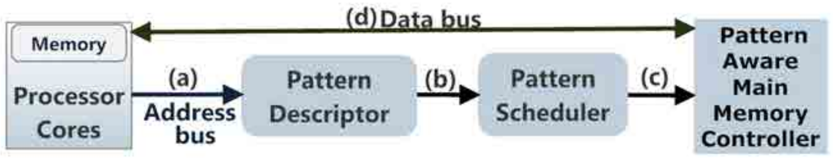


Figure 10.1: MAPC: Block Diagram With Data Flow

art Scatter Gather Direct Memory Access (SGDMA). When compared to the baseline system, the MAPC system achieves between 1.13x to 3.6x of speedup for different applications and consumes 28% fewer hardware resources, 13% less dynamic power. While running applications concurrently in a multi-core environment, the MAPC system achieves up to 5.34x of speedup.

10.1 MAPC Architecture

Main features of MAPC is already described in Chapter 2. MAPC supports and schedule complex memory access patterns to reduce run-time overhead of data access management and data transfer. MAPC reduces the memory access latency and improves fairness by arranging the memory accesses in patterns, by scheduling patterns with respect to data transfer requests and size and by using adaptive SDRAM banks management.

The main units of MAPC are shown in Figure 10.1, as well as the *Processor Cores*, which execute the applications. The *Address* bus is used to program the *Pattern Descriptor Unit*. The *Data Bus* is used to transfer data patterns to the memory of the *Processor Cores*. The *Pattern Scheduler* takes a single or multiple descriptors and prioritizes data accesses with respect to data transfer size and requests. The *Pattern Aware Main Memory Controller* is responsible for data transfers to/from SDRAM. The units of MAPC operate independently in parallel with different data access patterns.

10.1.1 MAPC Working Operation

Figure 10.1 presents control- and data-flow between units of MAPC, divided into run-time steps from (a) to (d).

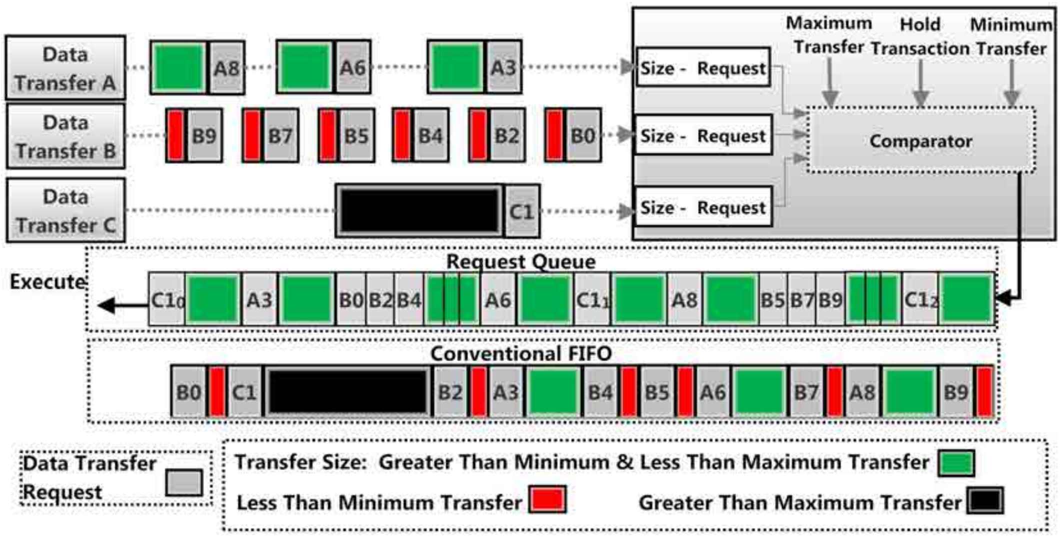


Figure 10.2: MAPC: Request, Hold and Grant Policy

(a) The *Pattern Descriptor Unit* manages compile and run -time generated data transfer requests of the *Processing Cores* and organizes them in the form of descriptors. MAPC uses the *Pattern Descriptor Unit* (*PDU*) to hold the information of complex memory access patterns in descriptors. The *PDU* uses one or multiple descriptor blocks to describe the access data patterns of a *Processor Core*. The *PDU* also manages run-time memory accesses as described in the *descriptor* block. At run-time the *PDU* allocates a single *descriptor* block for each processing core. The *PDU* takes memory requests from a *Processor Core*, buffers them and compares consecutive requests. If the addresses of consecutive memory requests have constant strides, the *PDU* allocates a *descriptor* block by defining *stride* and *size* parameters. If the request has variable strides then the *PDU* uses the *offset* parameter of the *descriptor* that points a random location (shown in Figure 10.3).

(b) The *Pattern Scheduler* reads multiple descriptors from the *Pattern Descriptor Unit* and prioritizes data patterns at run-time with respect to transfer size and requests. The *Pattern Scheduler* applies a **Request, Hold and Grant** (*RHG*) policy to process the memory requests, Figure 10.2 shows an example. The *RHG* depends upon the number of data access requests and transfer sizes. The *RHG* uses three parameters for scheduling which are the *Minimum Transfer*, the *Maximum Transfer* and the *Hold Transaction*. The *transfer* parameters specify the minimum and

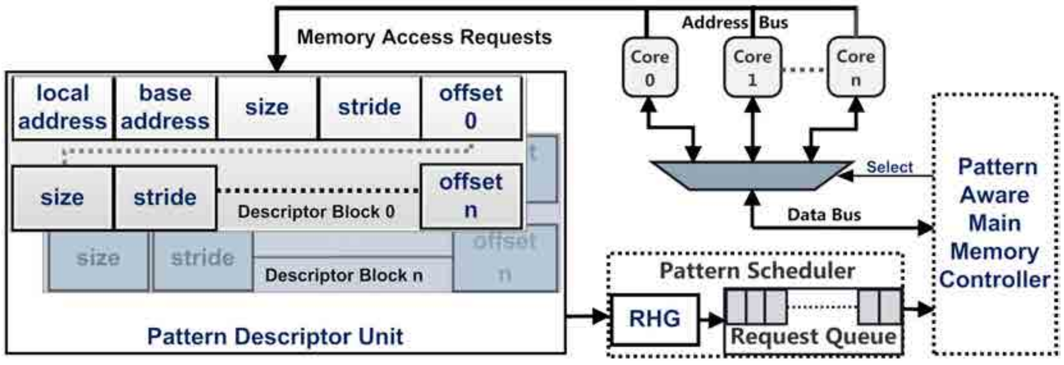


Figure 10.3: MAPC:: Pattern Descriptor and Pattern Scheduler

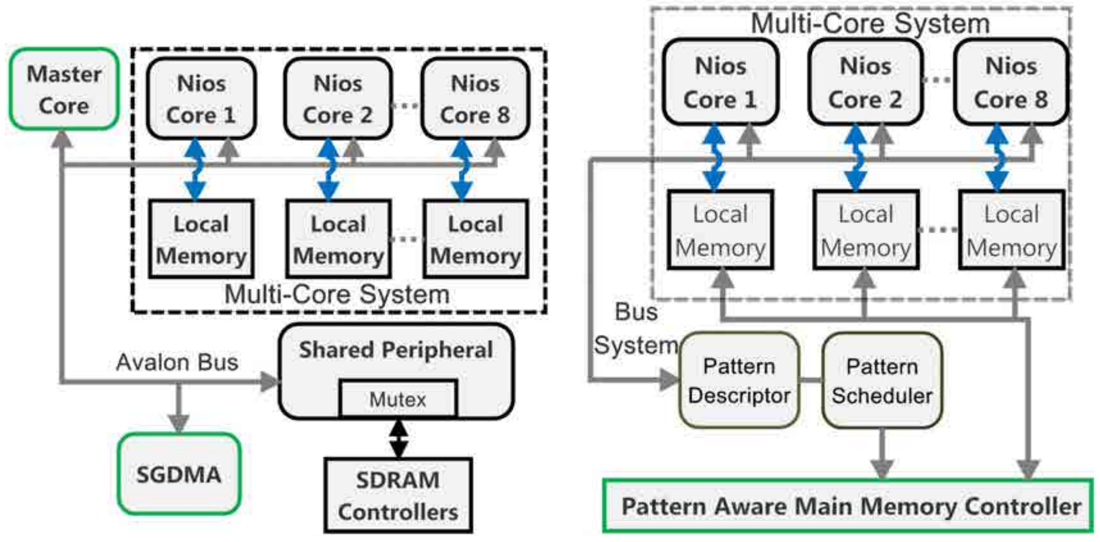
maximum size of the data transfer request on which the *Pattern Scheduler* applies the *RHG* policy. The *Pattern Scheduler* immediately grants all requests which are greater than *Minimum Transfer* and less than *Maximum Transfer*. It splits the transfers longer than *Maximum Transfer* and holds transfer requests of size less than *Minimum Transfer*, merging several of them into a single larger request. The *Hold Transaction* is the number of requests for which the *Pattern Scheduler* holds a request. 128B, 4KB and 8 are the default values of *Minimum Transfer*, *Maximum Transfer*, and *Hold Transaction* respectively. These parameters are programmed at compile-time.

(c) The *Main Memory System* uses *PAMMC* takes pattern requests from the *Pattern Scheduler*, decodes the data access patterns and manages SDRAM banks, rows and columns.

(d) The *PAMMC* transfers data between the *Processing Cores* and SDRAM.

10.2 Experimental Framework

To evaluate the proposed hardware, in this section, we describe the Nios- and MAPC-based *Multi-core Systems*. The Altera Quartus II version 13.0 and the Nios II Integrated Development Environment (IDE) are used to develop the systems. An Altera DE4 FPGA board is used to test the systems. In the current implementation on an Altera DE4 evaluation FPGA board, a DDR2 memory having SODIMM I/O module



(a) (b)

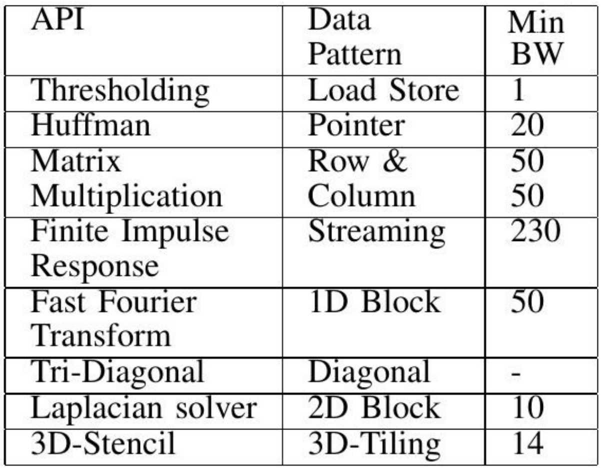
Figure 10.4: Multi-Core Systems: (a) Nios (b) MAPC

is connected. DDR2 SSDRAM device is used. The SDRAM memory has a capacity of 1GB and a peak bandwidth of 1.6GB since it has a clock frequency of 100MHz, a data rate of 2 words per clock cycle, row buffer of 32KB and a data bus width of 64 bits. This section is divided into three subsections: the *Multi-core and Applications*, the *Nios Multi-core System*, and the *MAPC Multi-core System*.

10.2.1 Multi-core System and Applications

We propose a Nios processor based *Multi-core system* having eight Nios II/f cores (shown in Figure 10.4 (a) and (b)). The Nios II/f is an RISC soft processor architecture, optimized and implemented with FPGA resources. The Nios II/f has a high performance barrel shifter with hardware multipliers and branch prediction. Each core has 16KB of Data and 8KB Instruction cache memory.

Figure 10.5 shows the application kernels which are executed on *Multi-core*. Each processor core executes a single application kernel, thus evaluating the system with a wide range of patterns. Each application kernel has different memory access patterns. They have been selected to measure the behaviour and performance of data management and data transfer in a variety of scenarios. Column *Min BW* (*minimum band-*



API	Data Pattern	Min BW
Thresholding	Load Store	1
Huffman	Pointer	20
Matrix Multiplication	Row & Column	50 50
Finite Impulse Response	Streaming	230
Fast Fourier Transform	1D Block	50
Tri-Diagonal	Diagonal	-
Laplacian solver	2D Block	10
3D-Stencil	3D-Tiling	14

Figure 10.5: Brief description of application kernels

width) presents data required (in bytes) for the processor core after each computation without applying techniques such as pipelining, overlapping, etc. The Tri-Diagonal application has a diagonal access pattern with unknown variable transfer size, therefore, its *minimum bandwidth* requirement is not predefined. While operating at 100Mhz the minimum required bandwidth is calculated with Equation 10.1. The operating frequency is the clock at which the processing core operates on accessed elements. The consume cycles are the clock cycles taken by a processing core to generate a single output element. When executing independent kernels in each of the core that access

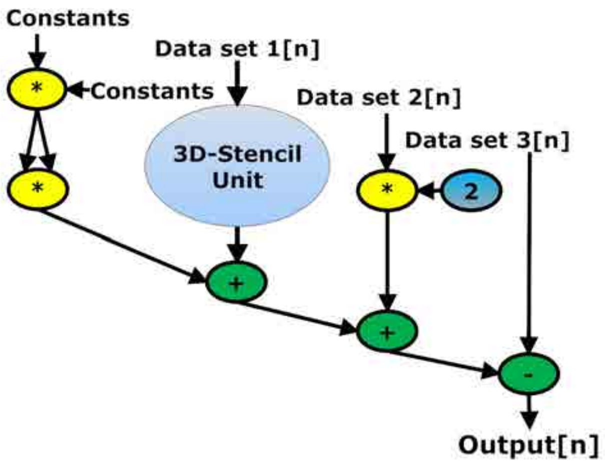


Figure 10.6: Control Data Flow Graph of RTM

its own data set, the behavior of the memory system can be very different from that observed when multiple cores execute different parts of the same application and share data, at least partially. In order to consider this scenario, we also execute MPAC with a Reverse Time Migration (RTM) application that runs on multiple cores. The control data flow graph of the RTM application is shown in Figure 10.6. The application kernel uses three data sets with different data transfer sizes, and the output of each data set is dependent on each other. For each output the RTM application needs 108 Bytes (27 elements), 100 Bytes from *data set 1* and 4 Bytes each from *data set 2* and *data set 3*.

$$MinimumBandwith = \frac{Operating\ Frequency \times Access\ Elements}{Consume\ Cycles} \tag{10.1}$$

10.2.2 Nios Multi-core System

The *Nios Multi-core System* shown in Figure 10.4 is used as baseline. The *Nios Multi-core System* uses a separate additional Nios II/f as master core that manages on-chip bus transactions, memory read/write requests between the multi-core system and SDRAM memory. An Altera Scatter-Gather DMA (SGDMA) is used that handles multiple data transfers efficiently. The *Nios Multi-core System* uses a hardware mutex core to share resources of multiple cores and prevent system conflicts. Before accessing shared peripherals each Nios core checks that the mutex is available, if it is free the processor acquires the right to use shared resources and locks the mutex core so that no other processor can access peripherals. The Altera mutex core has two 32-bit memory mapped mutexes and reset registers. *Nios Multi-core System* uses Avalon bus that manages data movement between the *Multi-core System* and SDRAMs. The Avalon bus is programmed to work at 400MB while using a 100MHz clock speed and a 32-bit data bus. The *Nios Multi-core System* uses 22,784 flip-flops, 22,416 ALUTs and 877,672 memory bits.

10.2.3 MAPC Multi-core System

Figure 10.4 (b) shows the implementation of a MAPC based multi-core system. The major difference between *Nios Multi-core System* and this is that at run-time MAPC

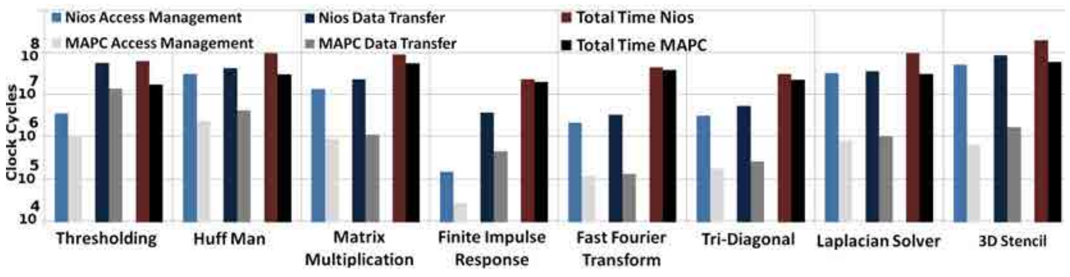


Figure 10.7: Single-Core Results: Execution Time

manages multi-core memory transfers and SDRAM memory without support of a separate Nios processor, complex on-chip bus controller and high performance DMA controller, as explained in previous Sections. The *MAPC Multi-core System* uses 16,904 flip-flops, 15,308 ALUTs and 767,672 memory bits.

10.3 Results and Discussion

This section analyzes the results of different experiments conducted on the MAPC and *Nios Multi-core System*s.

10.3.1 Single-Core Performance

In this section, we measure the performance of test applications shown in Figure 10.5 using MAPC and *Nios Single-core System*s. Either MAPC or the master core handles address management and data transfers for the application. To measure performance, a single application is executed at a time on a Nios II/f core. Multiple hardware timers are added to measure the *Computation*, *Access Management* and *Data Transfer* times for each application kernel. The *Total Time* shown in Figure 10.7 (a) is the sum of *Access Management*, *Data Transfer* and *Computation* time of each application. *Access Management* refers to the time needed to manage and schedule access pattern and *Data Transfer* to the time required to read/write data to/from SDRAM. The X and Y axis represent application kernels and number of clock cycles, respectively. The Y axis has logarithmic scale (lower is better).

Results show that MAPC improves *Access Management* time for *Thresholding*, *Huffman* and *Matrix Multiplication* up to 3.2x, 12.8x and 15.38x respectively over the *Nios Single-core System*. These applications have access patterns with variable and unknown strides that the MAPC *PDU* manages at run-time. The *Finite Impulse Response* (*FIR*) application has a streaming data transfer pattern and achieves 5.66x of speedup. The MAPC *PDU* requires only one descriptor to access the *FIR* pattern. This reduces the address generation/management time and on-chip request/grant time. The *Fast Fourier Transform* (*FFT*) application kernel reads a 1D block of data pattern, processes it and writes it back to SDRAM. This application achieves 18x of speedup. The *Tri-Diagonal* application has diagonal 1D data access and achieves 17.5x of speedup. *Laplacian Solver* takes 2D block of data and achieves 39.4x of speedup. The *3D-Stencil* application achieves 78.16x of speedup with the MAPC *PDU* and *Pattern Scheduler* that take 2D and 3D block descriptors and manage them in hardware.

For *Thresholding*, *Huffman* and *Matrix Multiplication* applications the MAPC PAMMC improves *Data Transfer* time up to 4.07x, 9.95x and 20.34x over the *Nios Single-core System*. These application kernels have several memory access requests with fewer data locality, use less open banks and rows of *SDRAM memory*. *FIR* requires a contiguous SDRAM Data Transfer pattern and allows *PAMMC* to access data from SDRAM open *row-buffers*, achieving 8.14x of speedup. *FFT* kernel requires multiple data patterns of 1D blocks. Each 1D block of data uses different row buffers. A single 1D access pattern utilizes one open row-buffer, and multiple data blocks require to open multiple row-buffers. The MAPC *PAMMC* improves 25.02x *Data Transfer* time. The *Tri-Diagonal* application achieves 20.14x of speedup. The *Laplacian Solver* application takes a 2D block of data and achieves 34x of *Data Transfer* speedup. The *3D-Stencil* application achieves 50.34x of speedup. The *3D-Stencil* requests data from multiple banks, the MAPC *PAMMC* applies multi-bank policies and keep multi-row buffer open which improves *Data Transfer* time. The *PAMMC* also processes access patterns in parallel with respect to available DRAM banks.

Results show that MAPC improves the overall performance of application kernels over the *Nios Single-core System*, including also *Computation* time into account. The *Thresholding* and *Huffman* achieve 3.6x and 3.1x of overall speedup respectively. The *Matrix Multiplication*, *Finite Impulse Response*, *Fast Fourier Transform*, *Tri-Diagonal* and *Laplacian* and *3D-Stencil* applications achieve 1.74x, 2.8x, 1.17x, 1.37x, 3.13x and 3.2x of speedup respectively. The *Matrix Multiplication*, *Finite Impulse Response*,

Table 10.1: MAPC System Speedups, Power and Energy for different Scheduling Policies

Scheduling Policies	min Tx	max Tx	Hold Tx	Speed up	Power	Energy	Imprrovement
NIOS - FIFO					5.75	25.47	1
MAPC- FIFO	-	-	1	2.45	4.97	9.54	2.66
RHG-32:4K:8	32	4K	8	2.71	5.15	10.1	2.51
RHG-128:4K:8	128	4K	8	3.25	5.1	7.1	3.63
RHG-256:4K:8	256	4K	8	3.6	5.05	6.1	4.20
RHG-512:4K:8	512	4K	8	3.8	5.03	5.8	4.35
RHG-128:1K:8	128	1K	8	2.62	5.22	10.3	2.46
RHG-128:2K:8	128	2K	8	2.81	5.19	9.1	2.79
RHG-128:8K:8	128	8K	8	4.01	5.1	6.04	4.21
RHG-128:16K:8	128	16K	8	4.45	4.9	5.22	4.87
RHG-128:4K:2	128	4K	2	2.53	5.01	9.39	2.71
RHG-128:4K:4	128	4K	4	2.74	5.08	9.14	2.78
RHG-128:4K:16	128	4K	16	3.20	5.13	8.02	3.17
RHG-128:4K:32	128	4K	32	3.22	5.16	7.95	3.20
RHG-16:1K:8	16	1K	8	2.41	5.08	9.3	2.73
RHG-1K:16K:4	1K	16K	4	4.29	5.19	5.35	4.75
RHG-1K:16K:8	1K	16K	8	5.34	5.28	4.37	5.82
RHG-1K:16K:16	1K	16K	16	5.22	5.26	4.46	5.70
RHG-2K:16K:8	2K	16K	8	5.14	5.2	4.47	5.69
RHG-1K:32K:8	1K	32K	8	5.28	5.16	4.32	5.88
RHG-2K:32K:8	2K	32K	8	4.98	5.14	4.56	5.57

Fast Fourier Transform and *Tri-Diagonal* applications are CPU bound, therefore benefit less from MAPC *Access Management* and *Data Transfer*.

10.3.2 Multi-Core Performance

In this section, we execute applications simultaneously using different scheduling policies on the multi-core systems. In order to isolate the effect of the RHG policy, first the applications are evaluated using a *FIFO* policy which serve the data transfer requests with the order of arrival time. While executing application kernels using *FIFO* policy, results show that MAPC system achieves 2.45x of speedup over *Nios Multi-core System*. The applications with a high number of requests cause bus arbitration delays. The *FIFO* policy gives them high priority, thus affecting the system fairness and starving other applications for long time periods. In order to determine the fairness of the MAPC system, the applications are executed using different *RHG* data transfer and transaction parameters (shown in Table 10.1). Fairness is discussed in Section 10.3.4, columns *Min Tx*, *Max Tx* and *Hold Tx* present *Minimum*, *Maximum Transfer Size* and *Hold Transactions* respectively. Column *Speedup* shows MAPC speedups having different data access scheduling policies over Nios *FIFO* data transfer policy.

MAPC achieves a minimum of 2.41x and a maximum of 5.34x speedup while using *RHG-16:1K:8* and *RHG-1K:16K:8* scheduling policies respectively. The *RHG-16:1K:8* policy holds 16B and 1KB values for the *Minimum* and *Maximum Transfer* parameters respectively. The *RHG-16:1K:8* executes data transfer requests greater than 16B and splits data accesses into multiple 1KB transfers having size greater than 1KB. The *RHG-1K:16K:8* has high values for the *Minimum* and *Maximum Transfer* parameters. For a *Minimum Transfer* of 1KB size, the MAPC *RHG-1K:16K:8* policy gathers multiple requests and the *PDU* tailors it in single descriptor block. This avoids multiple request grant time and allows MAPC to transfer the data in one transaction. Having *Maximum Transfer* size of 16KB avoids MAPC to split data transfers, and the *PAMMC* maximize the utilization of SDRAM open row buffer. Increasing *Minimum Transfer* beyond 1K affects the performance. *RHG-2K:16K:8* scheduling policy holds input data transfer requests until accumulates a transfer size of 2KB. The scheduling forces applications to wait which affects the fairness. Results show that MAPC RHG scheduling policy improves system performance and handles data more efficiently. The *Nios Multi-core System* uses multiple load/store or Scatter/Gather DMA calls to access complex patterns. The speedups are possible because MAPC can manage complex access patterns in descriptors and execute them without the support of extra hardware such as a microprocessor, bus controller or DMA. At run-time, MAPC takes information of complex access patterns from *PDU* independently and manages them in parallel, whereas the *Nios Multi-core System* is dependent on the master processor that

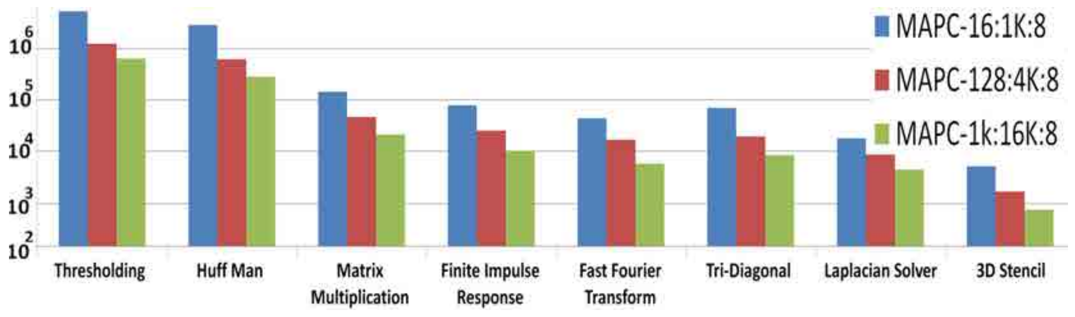


Figure 10.8: Fairness: Single-Core Hold Time

feeds data transfer instructions. The stand-alone working operation of MAPC removes the overhead of processor/memory system request/grant delay.

10.3.3 Resource and Total Power

While comparing the *MAPC Multi-core System* with the *Nios Multi-core System*, results show that the MAPC system consumes 28% fewer logic elements. To measure the total dynamic power, the DE4 board provides a resistor to sense current/voltage and 8-channel differential 24 bit analogue to digital converter. Table 10.1 also presents dynamic power and energy of the *MAPC* and *Nios Multi-core System*s executing applications concurrently at 100Mhz clock frequency, and each application processes a 2MB data set. Column *Power* presents measured dynamic power. Columns *Energy* and *Impr.* show respectively measured energy and improvement over the baseline, i.e. reduction of energy consumption over NIOS-FIFO. While using different scheduling policies results show that MAPC draws between 8.1% to 13.5% less power and 2.6x and 5.8x times less energy than the *Nios Multi-core System* respectively.

10.3.4 Fairness

To check the fairness of MAPC, we measure the hold time of each application for *RHG-128:4K:8*, *RHG-16:1K:8* and *RHG-1K:16K:8* scheduling policies having minimum, average and maximum system performance respectively (shown in Figure 10.8). The hold time includes the request initialization and management time for which the application core and MAPC remain stalled. The results show that the MAPC scheduling policies fairly prioritize applications having both small and large data transfers and avoid applications to starve. While increasing the minimum and maximum data transfer sizes for *RHG* scheduling policies results show that MAPC improves the performance as well as decreases the hold time of the applications. Results show that the *RHG-1K:16K:8* policy provides the best fairness as well as system throughput.

In the experiments explained above all core execute independent kernels. In order to evaluate the impact of *RHG* in speedup and fairness when cores are not independent, we use the RTM application. Two processor cores are used to execute one instance of RTM, the 3D-Stencil unit is executed on core1, and the rest of data processing is done on core2. 3D-Stencil kernel is programmed to process a 24x24x24 (55296B) 3D data block. For the *RHG* policy, the MAPC system uses 128B and 55296B for *Minimum* and *Maximum Transfer* respectively. While executing a single instance of RTM using *FIFO* and *RHG* policies, results show that the MAPC system achieves 4.53x and 4.89x of speedup respectively over a *Nios Multi-core System*. Four instances of RTM having 12 different data sets are executed on MAPC and *Nios Multi-core System*s. While executing four RTM applications together, the performance of MAPC improves up to 10.1x and 12.5x. This shows that MAPC improves performance of time-critical applications that contain computational steps that must be implemented efficiently.

10.3.5 Bandwidth

In this section, we measure the bandwidth of MAPC and *Nios Multi-core System*s for different number of cores by reading and writing two types of patterns. The X-axis

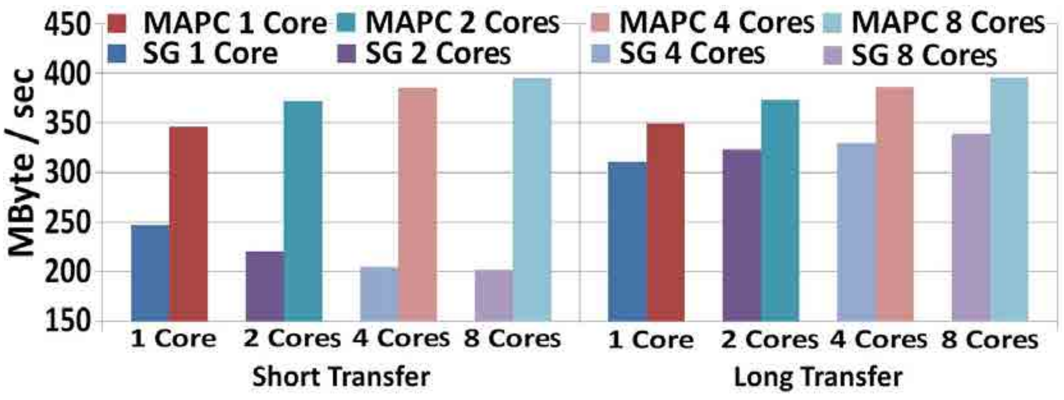


Figure 10.9: MAPC and *Nios Multi-core System*s: Memory Bandwidth

(shown in Figure 10.9) presents two types of data transfers and number of cores. Each data transfer reads and writes a data set of 2MB from/to the SDRAM. The type *Short Transfer* contains data transfers that have a maximum transfer size of 128B and the type *Long Transfer* has a transfer size of 4KB. Therefore, a single core has 32768 and 1024 read after write requests of *Short Transfer*s and *Long Transfer*s respectively. The requests increase with the number of processor cores. While using 1, 2, 4 and 8 processor cores for *Short Transfer* type, results show that the *MAPC Multi-core System* transfers data 1.40x, 1.68x, 1.88x and 1.95x times faster respectively than the *Nios Multi-core System*. While transferring data with the *Long Transfer* type, the *MAPC Multi-core System* improves bandwidth 1.07x, 1.13x, 1.16x and 1.17x times. Results show that the *Nios Multi-core System* reduces the aggregated bandwidth for *Short Transfer*s when increasing the number of cores. The *Nios Multi-core System* uses the SGDMA controller that forces to follow the bus protocol and requires a processor that provides data transfer instructions. For multiple cores, the *Nios Multi-core System* uses multiple instructions to initialize SGDMA. SGDMA can begin a new transfer before the previous data transfer completes with a delay called pipeline latency. The pipeline latency increases with the number of data transfers. Each Data Transfer requires bus arbitration, address generation and SDRAM bank/row management. The MAPC *Short Transfer* type uses few descriptors that reduce run-time address generation and address request/-grant delay and improve bandwidth by managing addresses at compile-time and by accessing data from multi-DRAM devices and multi-banks in parallel.

The most recent microprocessor and DRAM subsystems (e.g. DDR3, DDR4, etc.) give more bandwidth by running the system at higher clock speed but generate higher latency. The idea of MAPC is to reduce address management by organizing accesses into patterns, add fairness by scheduling patterns with respect to transfer size and utilize SDRAM banks and rows with respect to access patterns.

10.4 Conclusion

This chapter shows results for a system called MAPC, which implements all the features of PMC, as described in Chapter 2. MAPC enhances the application's performance by organising run-time complex access patterns in descriptors, schedules these access patterns with respect to access transfer size and requests, manages SDRAM open banks/rows and executes access patterns without processor intervention. The Nios Multi-core System is used as the baseline. The baseline system uses a separate additional Nios II/f as master core that manages on-chip bus transactions, memory read/write requests between the multi-core system and SDRAM memory. The system uses an Altera Scatter-Gather (see Section 11.3.2) DMA (SGDMA) is used that handles multiple data transfers efficiently. The experimental evaluations based on the Altera FPGA coupled with the *Nios Multi-core System* demonstrate that an MAPC based multi-core system efficiently manages complex memory patterns at compile time as well as at run-time. When compared to the baseline multi-core system, the MAPC based system achieves between 2.41x to 5.34x of speedup for different applications. While using different scheduling policies results show that MAPC draws between 8.1% to 13.5% less power and 2.6x and 5.8x times less energy than the *Nios Multi-core System* respectively.

Part VI

Related Work, Conclusions and Future Research Directions

11

Related Work

A number of memory systems have been proposed by research groups of academia and industry. A Memory System is selected by considering application's data access pattern and required performance. While executing the real applications having complex access patterns, a Memory System confronts difficulties while generating addresses and aligning streams for complex access patterns. A conventional memory system is categorized with respect to their usage, which are the *Local Memory System*, the *Memory Management* and the *Main Memory System*.

11.1 Local Memory System

11.1.0.1 Scratchpad

Scratchpad is a low latency memory that is tightly coupled to the CPU [93]. Therefore, it is a popular choice for on-chip storage in real-time embedded systems. The allocation of code/data to scratchpad memory is performed at compile time leading to predictable memory access latencies. Panda et al. [94] developed a complete allocation strategy for scratchpad memory to improve the average-case program performance. The strategy assumes that the access patterns are known at compile time. Suhendra et al. [95] aims at optimizing memory access tasks worst-case performance. However, in that study, scratch-pad allocation is static having static and predictable access patterns that do not change at run-time, raising performance issue when the amount of code/data is much larger than scratchpad size. Dynamic data structure management using scratchpad techniques are more effective in general because they may keep the working set in scratchpad. This is done by copying objects at predetermined points in the program in response to execution [96]. Dynamic data structure management requires a dynamic scratchpad allocation algorithm to decide where copy operations should be carried out. A time-predictable dynamic scratchpad allocation algorithm has been described by Deverge and Puaut [96]. The program is divided into regions, each with a different set of objects loaded into the scratchpad. Each region supports only static data structures. This restriction ensures that every program instruction can be trivially linked to the variables it might use. Udayakumaran et al. [97] proposed a dynamic scratchpad allocation algorithm that supports dynamic data structures. It uses a form of data access shape analysis to determine which instructions can access which data structures, and thus ensures that accesses to any particular object type can only occur during the regions where that object type is loaded into the scratchpad. However, the technique is not time-predictable, because objects are spilled into external memory when insufficient scratchpad space is available. PMC address manager arranges unknown memory access at run-time in the form of pattern descriptors. PMC performs data management and handles complex memory accesses at run-time using 1D/2D/3D *Scratchpad Memory*.

11.1.1 Cache

Numerous research groups in academia and industry have studied how to improve local memory address/data management and speed of irregular memory accesses. G. Stitt et al. [66] presented a traversal data cache structure that dynamically serializes pointer-based traversal data structure into the FPGA local memory and feeds the corresponding data to FPGA accelerators in a streaming fashion. This idea is based on exploiting the opportunity of repeated traversals for a branch of a tree with the assistance from a microprocessor. For traversing the tree data, especially for the case of Barnes-Hut tree, Coole et al. [98] extended the idea of traversal data caches [66] by prefetching multiple branches of data into the FPGA and keeping them accessible by the compute block thus exploiting repeated traversals and parallel execution of multiple traversals whenever possible. For large data sets, this implementation requires a slightly larger address space due to the complexity of the traversal data cache framework. The support of specialized memory and data access in hardware allows PMC to manage pointer-based data structures without a microprocessor core. This PMC reduced memory address space by transforming large and irregular address spaces into a few descriptor blocks and map them to the cache framework.

Mellor-Crummey et al. [99] studied the impact of reordering on data reuse at different levels in the memory hierarchy, and introduced an architecture independent multi-level blocking approach for irregular applications which performs data and computation reordering. Diniz et al. [100] described the mapping of traversals for Sparse-mesh and Quad-tree data structures to FPGA-based smart memory engines. This design supports the relocation of data in memory for improved locality. The work suggests that reconfigurable logic when combined with data reorganization can lead to significant performance improvements. The work focuses over spatial pointer-based data structures for specific data structures (Sparse-mesh and Quad-tree). PMC on-chip specialized memory unit maps and allocates different types of applications with irregular memory patterns. PMC on-chip memory unit improves performance by prefetching irregular patterns and providing them to the compute unit as a regular pattern, which hides latency and maximizes reuse ratio.

Application specific hardware accelerators with different data transfer modes use combination of scratchpad and cache memories. Yu et al. propose VIPERS [101], a vector architecture that consists of a scalar core to manage data transfers, a vector core for processing data, an address generation logic, and a memory crossbar to control data movement. Chou et al. present the VEGAS [31] vector architecture with a scratchpad to read and write data and a crossbar network to shuffle vector operations. VENICE [102] is an updated version of VEGAS, with scratchpad and DMA that reduces data redundancy. VENICE has limitations of rearranging complex data with scatter/gather support. Yiannacouras et al. propose the VESPA [30] processor that uses a configurable cache and hardware prefetching of a constant number of cache lines to improve the memory system performance. The VESPA system uses wide processor buses that match the system cache line sizes. VIPERS and VEGAS require a scalar Nios processor that transfers data between the scratchpad and the main memory. A crossbar network is used to align and arrange on-chip data. PMC eliminates the crossbar network and the limitation of using a scalar processor for data transfer. PMC manages addresses in hardware with the pattern descriptors and accesses data from main memory without support of a scalar processor core. The PMC data manager re-arranges on-chip data using the buffer memory without a complex crossbar network, which allows the vector processor to operate at higher clock rates.

11.2 Memory Manager

A *Memory Manager* lies in between the *Local Memory System* and the *Main Memory System* that manages on-chip data and controls off-chip accesses.

The snoop control unit (SCU) maintains the coherence of the *Local Memory System* in the multi-core systems. The SCU is responsible for managing the interconnect arbitration, communication, *Local Memory System* and *Main Memory System* data transfers and cache coherence for the multi-core system. The SCU communicates with each of the processor core through a *Local Memory System* coherency bus and manages the coherency between the L1 and the L2 caches. The block implements duplicated 4-way associative tag RAMs acting as a local directory that lists coherent cache lines held in the *Memory System* L1 data caches. The directory allows the SCU to check if data is in the L1 data caches with great speed and without interrupting the processors. Also, accesses can be filtered only to the processor that is sharing the data. The SCU can also copy clean data from one processor cache to another and eliminate the need for *Main Memory System* data accesses to perform this task. S. Zhuravlev et al. [103] presented an intelligent scheduler that is aware of underlying caches and schedules applications with respect to memory access demands. The proposed scheduler generates the hardware caches contentions between threads. The PMC system overcomes this problem by scheduling task operations while taking into account the programmed priorities, and on-chip specialized memories and run-time memory accesses patterns.

To solve on-chip bus bandwidth bottleneck, there have been several types of high-performance on-chip buses proposed. The multi-layer AHB (ML-AHB) bus-matrix proposed by ARM [104] has been used in many SoC designs due to its simplicity, simple architecture and low power. The ML-AHB bus-matrix interconnection scheme provides parallel access paths between multiple masters and slaves in a system. The PLB crossbar switch (CBS) from IBM [105] allows communication between masters on one PLB and slaves on the other. The CBS supports concurrent data transfers on multiple PLB buses along with a prioritization method to hold multiple requests to a generic slave port. Like other on-chip Bus Units, AHB (ML-AHB) and PLB (CBS) use a master core that manages on-chip bus transactions.

Marchand et al. [106] have developed software and hardware implementations of the Priority Ceiling Protocol that control the multiple-unit resources in a uniprocessor environment. Yan et al. [107] has designed a hardware scheduler to assist the synergistic processor cores task scheduling on heterogeneous multi-core architecture. The scheduler supports first come first service (FCFS) and dynamic priority scheduling strategies. It acts as helper engine for separate threads working on the active cores. The idea of scouting hardware threads [108; 109] was developed by Sun as part of the design of their latest processor called Rock (canceled since). The scout thread idea clearly targeted the conventional memory wall problem, trying to mask the latency of *Main Memory System* accesses. Helper threads also used this algorithm that improves the efficiency of *Local Memory System* usage. A separate core (hardware thread) is used that monitors the memory traffic between a specific core, records memory access patterns. By using this information whenever the same data access is observed again the helper core begin fetching data from the *Local Memory System*. If the data is already in the *Local Memory System*, the helper core makes sure that it stays there, and no unnecessary write-backs would occur. This method tends to reduce both latency, but also optimize memory bandwidth usage: if the prediction is correct, valuable memory traffic is prioritized, and unimportant one can be avoided. Thread level speculation [110] support in hardware is quite similar to the scouting thread concept: the processor is capable of performing run-ahead execution on certain branches, using private copies of data and registers, at the end either validating or discarding the result. The PMC holds information of memory patterns in the form of *Descriptor Memory*. Currently, accessed patterns are placed in the address manager of PMC. The PMC monitors the access patterns without using a separate core and reuses these patterns for multiple cores if required.

Several DRAM access scheduling techniques [111; 112] have been proposed and evaluated to optimize throughput, reduce the average memory latency and improve bandwidth utilization for streaming applications as well as general-purpose applications. Kim et al. [113] proposed DRAM scheduling algorithms for multiple memory controllers that perform thread prioritization decisions by tracking long term memory intensity of threads and utilize this information to reduce bandwidth limitations, memories contention, and enforce bank/port/channel/bus conflicts. Nazm et al. [114] presented a programmable memory controller (PARDIS) in hardware that maps existing proposed DRAM scheduling algorithms through dedicated command logic. PARDIS takes memory requests from the last-level cache and generates commands to arrange data transfers between the processor and main memory system. APEX [115] uses a library that first extracts the most active patterns exhibited by the application when accessing data structures, and explores the memory module configurations to match the needs of these access patterns. The PMC *Pattern Descriptor Unit* not only determines access patterns at compile-time but also manages at run-time complex patterns which are difficult to predict.

Corbal et al. [116] proposed the Command Vector Memory System (CVMS) which decreases the processor to memory address bandwidth usage by transferring commands (descriptors) to the memory controllers, rather than individual references. A CVMS descriptor includes a base and a stride which is extended into the suitable sequence of references by each off-chip memory bank controller. The bank controllers in the CVMS utilize a row/closed scheduling policy among commands to improve the bandwidth and latency of the SDRAM. PMC improves on-chip communication bandwidth by managing both compile- and run- time generated access pattern descriptors. The PMC descriptor block has uses offset parameter that align complex transfers, each transfer with variable stride. The PMC *Pattern Aware Main Memory Controller* manages SDRAM by using a bank management policy selected by run-time access patterns.

11.3 Main Memory System

Increasing the *SDRAM Controller* bus width increases the bandwidth but it is primarily limited up to a set limit by the pin counts, as well as the area requirements. Increasing clock frequency of the system bus is also very difficult due to the long, heavily capacitative PCB traces. Complex and irregular memory accesses generate latencies to initialize rows and banks of main memories. To overcome the latency of the *Main Memory System* a number of memory access techniques are proposed.

11.3.1 Prefetching

Prefetching is one way to overlap/interleave memory access management under other instruction this reduce application execution time. A lot of research has been done in the past to build efficient prefetchers. The ever increasing disparity between the processor and memory speeds continually add to the importance of latency hiding techniques such as software data prefetching. Software prefetching is useful for multiprocessor systems that can issue many memory requests. It is an attractive strategy for reducing the effect of long memory latencies without notably increasing the bandwidth required to support traffic between main memory and cache. Porterfield et al. [117; 118] presents a compiler algorithm for inserting prefetches. The technique is implemented as a preprocessing that introduced prefetching into the source code. Previous proposals prefetch all array references in inner loops one step before. Porterfield presents that such scheme was issuing too many superfluous prefetches and offered a smarter scheme supported by dependence vectors and overflow iterations. Since the simulation happened at a fairly conceptual level, the prefetching overhead is predictable rather than presented. Porterfield [117] also presents software prefetching approach that reduces cache missing latencies. By providing a non-blocking prefetch instruction that accesses data from specific memory address to be brought into the cache, the compiler overlaps the memory latency with other computation. Klaiber et al. [119] extended the Porterfield's work by recognizing the requirement to prefetch additional to a single iteration ahead. They incorporated multiple memory system parameters in the equation for how much iteration ahead to prefetch, and placed prefetches by hand at the assembly code level. They proposed prefetching into a separate fetch buffer rather than directly into the cache. Their results have confirmed that prefetching directly into the cache can provide significant speedups, and without the drawback of cache size reduction to house a fetch buffer. Gornish et al. [120; 121] presented an algorithm for determining the initial instance when it is safe to prefetch shared data in a multiprocessor by software controlled cache coherency. This work is focusing on a block prefetch instruction, rather than a single line prefetches. Doshi et al. [7] introduce a software data prefetching technique that improves the performance of programs that suffer many cache misses at several levels of the memory hierarchy. This technique utilizes register rotation and prediction method to hide latency such as software data prefetching and it delivers a performance improvement due to optimized prefetch scheduling. PMC manages software prefetching by using known *Descriptor Memory*. The PMC *Descriptor Memory* manages memory accesses at compile-time and prefetch them at run-time before computation. The PMC has ability to access chain of complex and irregular data transfers where each data transfer can have variable stride.

As software controlled prefetching schemes need support from both hardware and software, various methods have been proposed that are strictly following hardware based prefetching. Hardware based prefetching have better dynamic information, and therefore can recognize things for instance cache conflicts that are difficult to predict in the software based schemes during compile time. Hardware based prefetching do have overhead to initialize hardware block at run-time. Porterfield [118] evaluated several cache line based hardware prefetching schemes. In a few cases (known memory accesses), they were fairly efficient at reducing miss rates, but at the for unknown accesses, they offer a significant increase in memory traffic. Lee et al. [122] proposed a complex lookahead method for prefetching in a multiprocessor system where shared data is unachievable. They found that the efficiency of the method is dependent on branch prediction and synchronization. Baer and Chen [6] projected a scheme that uses a history buffer to sense strides. In their scheme, a *lookahead PC'* theoretically scans through the program ahead of the usual PC having branch prediction. When the look ahead PC finds a similar stride entry in the table, it issues a prefetch. Ganusov et al. [123] proposed the Efficient Emulation of Hardware Prefetchers via Event Driven Helper Threading (EDHT) that discovers the idea of using accessible general purpose cores in a chip multiprocessor environment as helper engines for separate threads working on the active cores. The EDHT framework uses lightweight hardware support for efficient event communication. Extra cores are used to execute prefetching threads that emulate the behavior of complex outcome prediction based prefetching algorithms using the EDHT framework. The EDHT framework is used for efficient event driven software emulation of complex hardware accelerators and describes the implementation of the EDHT framework for a range of prefetching techniques that reduced contention for shared resources. Gornish et al. [124] presented integration of data prefetching scheme that tries to improve software and hardware prefetching. Software schemes calculate address calculation instructions and a prefetch instruction for each cache line that needs to be prefetched. Hardware schemes can detect data access streams and strides by using complex hardware. In the integrated scheme, the compiler calculates the values for the prefetching offsets and the number of prefetches to issue for each access stream. Simple hardware is then provided to handle the bulk of the remaining accesses. Roth et al. [125] introduced a prefetching scheme that collects pointer loads along with the dependency relations. A separate prefetch engine takes the access description and executes load accesses in parallel with the original program. However, finding dependencies for linked data structures is easy compared to sparse matrices and index tree based data structures. The address flow of pointer access is irregular and predictable that remains unchanged through registers and transfers to/from memory. Dynamic prefetching [126; 127] is introduced when the microprocessor is designed to run a wide variety of workloads of which it is agnostic during the design. In comparison, the PMC *Memory Manager* dynamically initializes the *Descriptor Memory* for known and unknown memory access patterns. PMC also schedules the sparse vector, linked list, tree, etc. based access patterns at compile or runtime and prefetch them dynamically.

11.3.2 Scatter Gather Controllers

A scatter gather controller takes noncontiguous data requests reorders them and transfers data between main memory and local memory [128].

Wen et al. [129] explain FT64 and Multi-FT64 based system for High Performance Computing with streams. FT64 is a programmable 64bit stream processor, working as coprocessor of an Itanium2 processor to accelerate numerical code. The work describes a multiprocessor (multi-FT64) system architecture having the Network Interface on each FT64 to connect stream register file to other FT64, using the program Stream-LUCAS as an example. Wen et al. [130] present an FT64 based on chip memory sub system that combines software/hardware managed memory structure. Chai et al. [131] present a configurable stream unit for providing streaming data to hardware accelerators. The stream unit is situated in the system bus, and it prefetches and align data based on streams descriptors. The descriptor unit presents a method to let the programmer specify data movement explicitly by describing their memory access patterns. These descriptors also define data shape and location of stream. As the stream unit installed inside on chip bus unit. Internal and external bus delays remain present while transferring data. DC.Gou et al. [132] employ the extended Single Affiliation Multiple Stride (SAMS) synchronous memory scheme at an appropriate level in the memory hierarchy. SAM memory provides both Arrays of Structures (AoS) and Structure of Arrays (SoA) views for the structured data to the processor, appearing to have maintained multiple layouts for the same data. This memory hierarchy is used to achieve best SIMDization. PMC handles complex and irregular streams for HPC heterogeneous system. The type of transfer (e.g. row, column, diagonal, etc.) can be managed by defining the size of the stride. The PMC uses *Specialized Scratchpad* memory which handles complex data transfers and feed them to processing cores.

The XPS Channelized DMA Controller [90] provides simple Direct Memory Access (DMA) services to peripherals and memory devices on the Processor Local Bus (PLB). Lattice Semiconductor Scatter/Gather Direct Memory Access Controller IP [133] and ALTERA Scatter/Gather DMA Controller core [134] provide data transfers from noncontiguous block of memory to another by means of a series of smaller contiguous transfers. Both Scatter/Gather cores read a series of descriptors that specify the data to be transferred. Each Data transfer contains a unit stride that is not suitable for access complex unknown memory patterns. These DMA controllers are forced to follow microprocessor instructions and bus protocol. This introduces onchip/offchip bus delay as well as delay caused by microprocessor during address generation, management and arrangement. The data transfer of these controllers is regular and is managed/-controlled by a microprocessor (Master core) using a bus protocol. PMC extends this model by enabling the memory controller to access complex (regular and irregular) memory patterns and by working stand-alone in microprocessor or accelerator environment.

McKee et al. [135] introduce a Stream Memory Controller (SMC) system that detects and combines streams together at program-time and at run-time prefetches read-streams, buffers write-streams, and reorders the accesses to use the maximum available memory bandwidth. The SMC system describes the policies that reorder streams with a fixed stride between consecutive elements. McKee et al. also proposed the Impulse memory controller [8; 136] that supports application-specific optimizations through configurable physical address remapping. The Impulse memory controller [8] supports application-specific optimizations through configurable physical address remapping. By remapping physical addresses, applications can control the data to be accessed and cached. The Impulse controller works under the command of the operating system and relies on cache for local memory management. Impulse performs physical address remapping in software, which may not always be suitable for HPC applications using hardware accelerators. PMC remaps and produces physical addresses in the hardware unit without the overhead of operating system intervention. Based on its C/C++ language support, PMC can be used with any operating system that supports the C/C++ stack.

12

Conclusions and Future Research Directions

This chapter presents the conclusions of the research pursued during this thesis work. Moreover, it also throws some light on subsequent future research.

12.1 Conclusions

The ever-increasing complexity of high-performance computing applications limits the performance due to memory constraints in heterogeneous multi-core systems. In this thesis, we propose a novel access pattern-based multi-core memory architecture called PMC that improves the processor-memory performance gap. PMC supports both regular and irregular memory access patterns for heterogeneous multi-core system using memory pattern descriptors to reduce the impact of memory latency. Regular and irregular access patterns of multi-cores are described using a separate *Descriptor Memory*, which reduces the on-chip communication time and run-time address generation overhead. The memory architecture uses the *Specialized Scratchpad Memory* that tailors local memory organization and maps complex access patterns and uses a memory manager that efficiently accesses, reuses and feeds data access patterns to the multi-core system. Furthermore, to improve the on-chip data access a *Memory Manager* is integrated that efficiently accesses, reuses, aligns and feeds data to the multi-core heterogeneous system. The *Memory Manager* organizes and rearranges multiple non-contiguous memory accesses simultaneously which reduces the read/write delay due to the control selection of SDRAM memory. The memory architecture applies a run-time data access prioritizing policy by using the *Scheduler* which rearranges access patterns according to data transfer request and size. The *Pattern Aware Main Memory Controller* decodes access pattern descriptors and manages DRAM open banks and rows with respect to the access pattern.

A conventional memory system accesses data using singled or a multiple load/store and DMA calls. These data transfer calls transfer single or block of regular data elements. If the access patterns are complex and irregular then the task of programming these accesses becomes significantly more difficult. The PMC memory system accesses data in the form of patterns/shapes based on application requirements using specialized data transfer calls. These data transfers not only reduce the programmers effort of manually arranging memory accesses, but meet the performance requirements of HPC applications with complex and irregular patterns. A conventional *Local Memory System* faces issues while handling complex and irregular data with no data locality. Unlike the *Local Memory* structure, the PMC *Specialized Scratchpad Memory* has a parameterizable 3D structure that arranges data of noncontiguous memory locations and deliberately places them at a known location, rather than on fixed 2D structure according to a fixed hardware/software policy. The *Specialized Scratchpad Memory* accesses the whole data pattern as a cache line and temporarily holds data to speedup later accesses. Unlike a scratchpad memory system which uses a processor core to handle data transfers and accesses an aligned block of data for each data miss, the PMC *Address Manager* handles complex patterns and if there is a data miss it accesses only the missed data pattern. The PMC *Data Manager* along with the *Address Manager* efficiently reuse data patterns already available in *Specialized Scratchpad Memory*. A conventional on-chip/off-chip bus system uses multiple requests for complex data transfers. The PMC *Memory Manager* improves on-chip/off-chip bus bandwidth by organizing data transfer requests in descriptor commands that reduce bus switching and improve address bus bandwidth. The *Memory Manager* arranges and aligns data patterns using the *Data Manager* and the *Buffer Memory* and feeds patterns to the multi-core system without using a complex crossbar network. The *Main Memory System* reduces external memory delays and improves the data bus performance by utilizing the already open row buffers and uses a multi-bank mode for complex patterns that enables multiple row buffers and read/write data in parallel.

The PMC system is successfully applied in several different scenarios including vectors processors, high level synthesis accelerators, graphics system, single-core and multi-core including heterogeneous multi-cores with general purpose processor and application specific hardware accelerators. The PMC system is tested and verified on different types of FPGAs; we have targeted the Xilinx ML505 Evaluation Platform with a 65 nm Virtex-5 LX FPGA. Figure 12.1 shows maximum operating frequency of different PMC units on Virtex-5 LX FPGA. The board has 128 Block RAMs and a single DDR2 SDRAM of 256 MB capacity. On the ML505 board, the PMC system can operate at a maximum of 260 MHz. The PMC design has been evaluated also on the Xilinx Virtex-7 FPGA VC707 Evaluation Kit with XC7VX485T 28 nm FPGA. The board has 68Mb BRAM and 1GB DDR3 SDRAM. The PMC is also tested on the Altera DE4 Development board with 40 nm Stratix IV EP4SGX230 FPGA family. The focus of FGPA based PMC system is for low volume production; whereas the design can be ported to an ASIC if the demand is high. PMC is written in register transfer level (RTL) VHDL code, so we can synthesize the design to an application specific integrated circuit (ASIC) with a vendor library. The PMC design of a traditional standard cell ASIC device will involve tasks such as; placement and physical optimization, clock tree synthesis, signal integrity analysis, and routing using different EDA software tools.

On the ML505 board, the PMC-based multi-core system consumes 21% fewer hardware resources, 27.9% less on-chip power and achieves 6.8x of speed-up com-

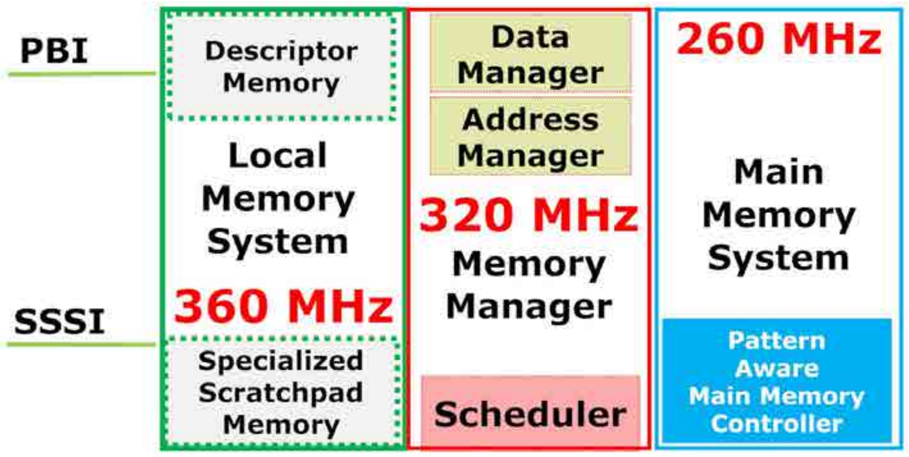


Figure 12.1: PMC : Maximum Operating Frequency of Each Unit on ML505 FPGA

pared to the MicroBlaze-based multi-core system. Using a single core, PMC transfers regular and irregular data sets up to 20.4x and 3.4x faster, respectively. When compared to the MicroBlaze based multi-core system implemented on the VC707 Kit, PMC transfers memory patterns up to 4.65x faster. While using the Altera DE4 Development board, the PMC based *Nios Multi-core system* achieves between 2.41x to 5.34x of speedup for different applications over conventional *Nios Multi-core System*. Results show that PMC draws between 8.1% to 13.5% less power and 2.6x and 5.8x times less energy than the *Nios Multi-core System* respectively. PMC transfers different data sets up to 1.40x and 1.95x faster for single core and multi-core respectively than the baseline system with a conventional but state-of-the-art Scatter Gather Direct Memory Access (SGDMA).

12.2 Future Research Directions

The most recent microprocessor and memory subsystems (e.g. DDR3, DDR4, etc.) provide higher bandwidth by running the system at higher clock speed but as a consequence, experience higher latency. The idea of our memory architecture is (i) to reduce the address management overhead by organizing memory accesses into patterns, (ii) add fairness by scheduling patterns with respect to transfer size and, (iii) utilize memory banks and rows with respect to access patterns. Recent and emerging technologies such as Ferroelectric RAM (FeRAM), Phase-Change RAM (PCRAM), Magnetic RAM (MRAM), Flash, and Memristor, have shown promise, and some of them are already being considered for implementation into emerging products. **FeRAM** is a non-volatile memory offering low power consumption but are poorly scalable [137]. **MRAMs** [138] and **PCRAMs** also have poor scalability and require large programming currents [139] during a write cycle that increases power dissipation per bit and makes voltage scaling difficult. **FLASH** memories suffer from both slow write/erase times and low endurance cycles [140]. **Memristors** [141] however, have demonstrated promising results in terms of the write operation voltage scaling. But their read/write access times are much higher than conventional SRAMs and DRAMs. These emerging memory technologies show promise for ultra-high density memories, but are not feasible for HPC heterogeneous multi-core memory systems due to their higher read/write access latencies [142]. For this reason, we expect PMC to be applicable to emerging

```
#pragma CSS task
[input ( parameters ) ] \
[output ( parameters ) ] \
[inout ( parameters ) ] \
[target device( [CELL, SMP,
   CUDA] ) ] \
[implements ( task name ) ] \
[reduction ( parameters ) ] \
[ high priority ]
```

(a)

```
#pragma PMC task
[input ( descriptors ) ] \
[output ( descriptors ) ] \
[inout ( descriptors )] \
[target device( [Heterogeneous
    Multi-core PMC] ) ] \
[implements ( Core Name ) ] \
[reduction ( parameters ) ] \
[ high priority ]
```

(b)

Figure 12.2: Programming Model Data Transfer Example: (a) StarSs (b) PMC

memory technologies. However, we consider that there is significant work to be done to fully understand how to optimize PMC to manage the actual characteristics of these new memories.

Virtualization of computer hardware resources is currently one of the main research topics of HPC. Virtualization hides the physical characteristics of the current computing platform from users and shows an abstract computing platform. Heterogeneous multi-core architectures using reconfigurable computing fabrics have shown great potential in many high-performance applications that benefit from hardware customization, while still relying on some amount of programming effort. Applications are programmed for a given fixed-size hardware. The resulting configuration cannot be reused to program a device of a different type or size. The key to overcoming this limitation is to combine hardware virtualization with the pattern-based memory architecture. By adding more features in *Descriptor Memory* with a description of the data flow (bus communication paths between different cores) and the control flow (sequencing of operators) a hardware virtualization model can be defined.

Parallel Programming Models [70; 143] permit programmers to write sequential applications, utilize the concurrency and use the heterogeneous components for automatic parallelization at run time. The idea is to replace the parallel programming model data transfer calls (e.g. StarSs in Figure 12.2 (a)) with PMC calls (see example in Figure 12.2 (b)) that can simplify the optimization and rearrangement of memory transfers by the programmers. The integration of PMC calls in a programming model will facilitate programmers to write their code without going into hardware details. The programming model can embed a set of specialized data access patterns inside PMC API (using *Descriptor Memory*) that would effectively eliminate the requirement of explicitly programming data transfers for a range of applications. Using PMC function calls, the programming model identifies the functions that will be executed as tasks on a specialized core (e.g. *VP*, *ASHA* , *SSP*, etc.) and transfers complex regular/irregular and specialized data patterns. These tasks can potentially be executed in parallel with defined inputs and outputs using PMC descriptors. The current PMC performs data transfer synchronization, load balancing, scheduling to optimize data locality, etc. of multi-cores running on a single cluster and connected to PMC using *PBI* and *SSSI*. A network interface (e.g. Fast/Gigabit ethernet) is required in the PMC architecture to communicate with multi-core machines running on a different cluster. In this scenario, the PMC can best utilize the multi-cluster hardware resources, manage and control different heterogeneous components e.g. memories, processors/accelerators, etc. Such PMC extensions to Parallel Programming Models will also make easier the programming of reconfigurable heterogeneous HPC systems and their implementation in a prototype runtime system.

Publications

I Publications

1. Reconfigurable Memory Controller with Programmable Pattern Support; Tassadaq Hussain, Miquel Pericas, Nacho Navarro, Eduard Ayguade; The 5th HiPEAC Workshop on Reconfigurable Computing, (WRC 2011).

2. Implementation of a Reverse Time Migration Kernel using the HCE High Level Synthesis Tool; Tassadaq Hussain, Miquel Pericas, Nacho Navarro, Eduard Ayguade; The 2011 International Conference on Field-Programmable Technology IIT Delhi New Delhi, India (FPT 2011).

3. PPMC : A Programmable Pattern based Memory Controller; Tassadaq Hussain,Muhammad Shafiq, Miquel Pericas, Nacho Navarro, Eduard Ayguade; The 8th International Symposium on Applied Reconfigurable Computing (ARC 2012).

4. PPMC : Hardware Scheduling and Memory Management support for Multi Hardware Accelerators; Tassadaq Hussain, Miquel Pericas, Nacho Navarro, Eduard Ayguade; The 22nd International Conference on Field Programmable Logic and Applications (FPL 2012).

5. APMC: Advanced Pattern based Memory Controller; Tassadaq Hussain, Oscar Palomar, Adrian Cristal, Osman Unsal, Eduard Ayguade, Mateo Valero and Rethinagiri Santhosh Kumar; 22nd ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA 2014).

6. Stand-alone Memory Controller for Graphics System; Tassadaq Hussain, Oscar Palomar, Adrian Cristal, Osman Unsal, Eduard Ayguade and Mateo Valero; The 10th International Symposium on Applied Reconfigurable Computing (ARC 2014).

7. Memory Controller for Vector Processor; Tassadaq Hussain, Oscar Palomar, Adrian Cristal, Osman Unsal, Eduard Ayguade and Mateo Valero; The 25th IEEE International Conference on Application-specific Systems, Architectures and Processors (ASAP 2014).

8. Advanced Pattern based Memory Controller for FPGA based Applications; Tassadaq Hussain, Oscar Palomar, Adrian Cristal, Osman Unsal, Eduard Ayguade and Mateo Valero; International Conference on High Performance Computing and Simulation (HPCS 2014).

9. PGC: a pattern-based graphics controller; Tassadaq Hussain and Amna Haider; International Journal of Circuits and Architecture Design (IJCAD).

10. MAPC: Memory Access Pattern based Controller; Tassadaq Hussain, Oscar Palomar, Adrian Cristal, Osman Unsal, Eduard Ayguade and Mateo Valero; The 24th International Conference on Field Programmable Logic and Applications (FPL 2014).

11. PMSS: A programmable memory system and scheduler for complex memory patterns; Tassadaq Hussain, Amna Haider and Eduard Ayguade; ScienceDirect: Journal of Parallel and Distributed Computing 2014.

12. AMMC: Advanced Multi-core Memory Controller; Tassadaq Hussain, Oscar Palomar, Adrian Cristal, Osman Unsal, Eduard Ayguade and Mateo Valero; 2014 International Conference on Field-Programmable Technology (FPT 2014).

13. PAMS: Pattern Aware Memory System For Embedded Systems; Tassadaq Hussain, Nehir Sonmez. Oscar Palomar, Adrian Cristal, Osman Unsal, Eduard Ayguade and Mateo Valero; 2014 International Conference on ReConFigurable Computing and FPGAs (ReConFig 2014).

14. AMC: Advanced Multi-accelerator Controller; Tassadaq Hussain, Amna Haider, Shakaib A. Gursal and Eduard Ayguade; ScienceDirect: Journal of Parallel Computing 2014.

15. Advanced Memory Controller for Vector Processor; Tassadaq Hussain, Oscar Palomar, Adrian Cristal, Osman Unsal, Eduard Ayguade and Mateo Valero; To appear in Journal of Signal Processing Systems (JSPS) 2015.

II Other Papers and Extended Abstracts

1. Streaming Scatter Gather DMA Controller for Hardware Accelerators; Tassadaq Hussain, Miquel Pericas, Nacho Navarro, Eduard Ayguade; Advanced Computer Architecture and Compilation for Embedded Systems (ACACES 2010); ISBN 978-90-382-1631-7, Terrassa, Spain, July 2010.

2. RTM Kernel Implementation on FPGA using High Level Synthesis Tool; Tassadaq Hussain, Paolo Palazzari and Koen De Bosschere; HiPEAC PhD Internship Project Report: 2012; isbn : 978-3659557651

3. PhD Internship Report; Tassadaq Hussain; Appeared in HiPEAC info (issue 29): January 2012

4. Programmable Memory Controller for Vector System-on-Chip; Tassadaq Hussain; The seventh Microsoft Research Summer School, Microsoft Research in Cambridge, U.K., from 2 July to 6 July 2012.

5. PPMC: On Chip Memory Manager and Scheduler for Vector Processor; Tassadaq Hussain; Appeared in HiPEAC info (issue 32): October 2012.

6. PGC: Programmable Graphics Controller; Tassadaq Hussain and Amna Haider; Appeared in HiPEAC info (issue 38): April 2014.

7. Supporting Scatter/Gather Tasks in Manycore architectures; Tassadaq Hussain; 1st BSC Doctoral Symposium in Barcelona Supercomputing Center: May 2014.

References

[1] Kim Jung-Sik et al., “A 1.2 V 12.8 GB/s 2 Gb Mobile Wide-I/O DRAM With 4 128 I/Os Using TSV Based Stacking,” *Solid-State Circuits, IEEE Journal of.* 4

[2] Kho, Rex et al., “A 75 nm 7 Gb/s/pin 1 Gb GDDR5 graphics memory device with bandwidth improvement techniques.” 4

[3] Monica S. Lam, et al., “The Cache Performance and Optimizations of Blocked Algorithms,” in *ASPLOS’91.* 4, 151

[4] Banakar Rajeshwari, Steinke Stefan, Lee Bo-Sik, Balakrishnan M, Marwedel Peter, “Scratchpad memory: design alternative for cache on-chip memory in embedded systems,” in *Proceedings of the 10th international symposium on Hardware/software codesign.* ACM, 2002. 4, 14

[5] Udayakumaran Sumesh, Dominguez Angel and Barua Rajeev, “Dynamic allocation for scratch-pad memory using compile-time decisions,” *ACM Transactions on Embedded Computing Systems (TECS)*, 2006. 4, 15

[6] J. Baer and T-F. Chen., “An effective on-chip preloading scheme to reduce data access penalty.” in *SC’91.* 4, 151, 204

[7] Gautam Doshi, et al., “Optimizing Software Data Prefetches with Rotating Registers,” in *PACT’01.* 4, 151, 203

[8] Zhang. Lixin et al., “The impulse memory controller,” *IEEE Transactions on Computers*, 2001. 4, 151, 206

[9] Tassadaq Hussain, Miquel Pericas, Nacho Navarro and Eduard Ayguade, “PPMC: Hardware Scheduling and Memory Management support for Multi Hardware Accelerators.” in *FPL 2012.* 4, 25, 117, 151

[10] Diniz Pedro, et al., "Data search and reorganization using FPGAs: application to spatial pointer-based data structures," in *FCCM'03*. 5, 151

[11] Xilinx Virtex-7, "Leading FPGA System Performance and Capacity," 2012. [Online]. Available: http://www.xilinx.com/products/silicon-devices/fpga/virtex-7/index.htm 5, 131

[12] Bhole Mayur, Kurude Aditya and Pawar Sagar, "3D Tri-Gate Transistor Technology and Next Generation FPGAs." 5

[13] Leiserson Charles E and Mirman Ilya B, "How to survive the multicore software revolution (or at least survive the hype)," *Cilk Arts, Cambridge*, 2008. 6

[14] Shan Yi, Hao Yuchen, Wang Wenqiang, Wang Yu, Chen Xu, Yang Huazhong, Luk Wayne, "Hardware Acceleration for an Accurate Stereo Vision System Using Mini-Census Adaptive Support Region," *ACM Transactions on Embedded Computing Systems (TECS)*, 2014. 6

[15] Yiu Ka Fai Cedric, Li Zhibao, Low Siow Yong and Nordholm Sven, "FPGA multi-filter system for speech enhancement via multi-criteria optimization," *Applied Soft Computing*, 2014. 6

[16] Tassadaq Hussain, Miquel Pericas, Nacho Navarro and Eduard Ayguade, "Implementation of a Reverse Time Migration Kernel using the HCE High Level Synthesis Tool," *FPT 2012*. 6, 25, 125, 131, 132

[17] Kuon Ian and Rose Jonathan, "Measuring the gap between FPGAs and ASICs," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 2007. 7

[18] Wen Xingzhi and Vishkin Uzi, "Fpga-based prototype of a pram-on-chip processor," in *Proceedings of the 5th conference on Computing frontiers*. ACM, 2008. 7

[19] Athanas Peter, Pnevmatikatos Dionisios and Sklavos Nicolas, *Embedded Systems Design with FPGAs*. Springer, 2013. 7

[20] Embedded Development Kit EDK 10.1i, *MicroBlaze Processor Reference Guide*. 8, 69, 88, 173

[21] "Nios II: Processor Reference Handbook," 2009. 8, 105

[22] Yiannacouras Peter et al., "The microarchitecture of FPGA-based soft processors," in *International conference on Compilers, architectures and synthesis for embedded systems 2005*. 8, 103

[23] Villarreal Jason, Park Adrian, Najjar Walid and Halstead Robert, "Designing modular hardware accelerators in C with ROCCC 2.0," in *The 18th Annual International IEEE Symposium on Field-Programmable Custom Computing Machines. FCCM2010*. 8, 11, 69, 142

[24] Feist Tom, "Vivado design suite," *Xilinx, White Paper Version*, vol. 1, 2012. 10

[25] "AutoESL High-Level Synthesis Tool." [Online]. Available: http://www.xilinx.com/tools/autoesl.htm 10

[26] "Impulse CoDeveloper Overview," 3,April 2011. [Online]. Available: http://www.impulseaccelerated.com/ 10

[27] Matthew Bowen, "Handel-C Language Reference Manual." 10

[28] Graphics, Mentor, "Catapult C synthesis overview," 2012. 10

[29] Alessandro Marongiu and Paolo Palazzari, "The HARWEST Compiling Environment: Accessing the FPGA World through ANSI-C Programs." *CUG 2008 Proceedings*, 2008. 11, 132

[30] Yiannacouras, et al., "VESPA: portable, scalable, and flexible FPGA-based vector processors," in *CASES 2008, Proceedings of international conference*. 11, 95, 96, 100, 104, 199

[31] Chou Christopher H et al., "VEGAS: soft vector processor with scratchpad memory," in *Proceedings of the international symposium on FPGA 2011*. 11, 95, 96, 199

[32] Russell Richard M, "The CRAY-1 computer system," *Communications of the ACM 1978*. 11

[33] Dominguez Angel, Udayakumaran Sumesh and Barua Rajeev, "Heap Data Allocation to Scratch-pad Memory in Embedded Systems," *J. Embedded Comput.* 15

[34] Tassadaq Hussain, Miquel Pericas, Nacho Navarro and Eduard Ayguade, "Reconfigurable Memory Controller with Programmable Pattern Support," *HiPEAC Workshop on Reconfigurable Computing*, Jan, 2011. 25, 67

[35] Tassadaq Hussain, Muhammad Shafiq, Miquel Pericas, Nacho Navarro and Eduard Ayguade, "PPMC: A Programmable Pattern based Memory Controller," in *ARC 2012*. 25, 67, 117, 179

[36] Tassadaq Hussain, Oscar Palomar, Adrian Cristal, Osman Unsal, Eduard Ayguady, Mateo Valero and Amna Haider, "Stand-alone Memory Controller for Graphics System," in *The 10th International Symposium on Applied Reconfigurable Computing (ARC 2014)*. ACM, 2014. 25, 77

[37] Tassadaq Hussain and Amna Haider, "PGC: A Pattern-Based Graphics Controller," *Int. J. Circuits and Architecture Design*, 2014. 25, 77

[38] Tassadaq Hussain, Oscar Palomar, Adrian Cristal, Osman Unsal, Eduard Ayguady and Mateo Valero, "Memory Controller for Vector Processor," in *The 25th IEEE International Conference on Application-specific Systems, Architectures and Processors*. IEEE ASAP 2014 Conference, 2014. 25, 95

[39] Tassadaq Hussain, "Programmable Memory Controller for Vector System-on-Chip," *The seventh Microsoft Research Summer School, Microsoft Research in Cambridge, U.K*, 2012. 25, 95

[40] Tassadaq Hussain, Amna Haider and Eduard Ayguade, "PMSS: A programmable memory system and scheduler for complex memory patterns," *ScienceDirect Journal of Parallel and Distributed Computing*, 2014. 25, 117

[41] Tassadaq Hussain, Amna Haider, Shakaib A. Gursal and Eduard Ayguade;, "AMC: Advanced Multi-accelerator Controller," *ScienceDirect Journal of Parallel Computing*, 2014. 25, 131

[42] Tassadaq Hussain, Oscar Palomar, Adrian Cristal, Osman Unsal, Eduard Ayguady and Mateo Valero, "Advanced Pattern based Memory Controller for FPGA based Applications," in *International Conference on High Performance Computing & Simulation*. ACM, IEEE, 2014, p. 8. 26, 151

[43] Tassadaq Hussain, Oscar Palomar, Adrian Cristal, Osman Unsal, Eduard Ayguady, Mateo Valero and Rethinagiri, Santhosh Kumar, "APMC: Advanced Pattern based Memory Controller," *22nd ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA 2014)*, 2014. 26, 151

[44] Tassadaq Hussain, Oscar Palomar, Adriyn Cristal, Osman Unsal, Eduard Ayguady and Mateo Valero, “AMMC: Advanced Multi-core Memory Controller,” in *2014 International Conference on Field-Programmable Technology (FPT 2014).* IEEE. 26, 169

[45] Tassadaq Hussain, Oscar Palomar, Adrian Cristal, Osman Unsal, Eduard Ayguady and Mateo Valero, “MAPC: Memory Access Pattern based Controller,” in *International Conference on Field Programmable Logic and Applications (FPL2014).* IEEE. 26, 179

[46] Tassadaq Hussain, Nehir Sonmez, Oscar Palomar, Adriyn Cristal, Osman Unsal, Eduard Ayguady and Mateo Valero, “PAMS: Pattern Aware Memory System for Embedded Systems,” in *International Conference on Reconfigurable Computing and FPGAs (ReConFig 2014).* IEEE, 2014. 26, 179

[47] Cong Jason, Ghodrat Mohammad Ali, Gill Michael, Grigorian Beayna and Reinman Glenn, “CHARM: a composable heterogeneous accelerator-rich microprocessor,” in *Proceedings of the 2012 ACM/IEEE international symposium on Low power electronics and design.* ACM, 2012. 67

[48] Xilinx, *Memory Interface Solutions*, December 2, 2009. 72, 157

[49] M. Shafiq, M. Pericas, and R., “Exploiting memory customization in fpga for 3d stencil computations.” 73

[50] Chen Chang-San, “Programmable multi-level bus arbitration apparatus in a data processing system,” US Patent 5,528,767. 77

[51] Hartley Richard and Zisserman Andrew, *Multiple view geometry in computer vision.* Cambridge Univ Press, 2000, vol. 2. 81

[52] “Xilinx University Program XUPV5-LX110T Development System.” [Online]. Available: http://www.xilinx.com/univ/xupv5-lx110t.htm 88, 123, 138

[53] Xilinx LogiCORE IP, *Local Memory Bus (LMB)*, December, 2009. 89

[54] Embedded Development KitEDK 10.1i, *MicroBlaze Processor Reference Guide.* 89, 140

[55] “Visual computing technology from NVIDIA,” http://www.nvidia.com/. 95

[56] Espasa Roger et al., “Vector architectures: past, present and future,” in *12th international conference on Supercomputing*, 1998. 95

[57] Kozyrakis C. et al., “Overcoming the limitations of conventional vector processors,” in *ACM SIGARCH Computer Architecture News, 2003*. 95

[58] Lee Yunsup et al., “Exploring the tradeoffs between programmability and efficiency in data-parallel accelerators,” 2011 booktitle=. 95

[59] Cheng Hui, “Vector pipelining, chaining, and speed on the IBM 3090 and cray X-MP.” 97

[60] Weiss Michael, “Strip mining on SIMD architectures,” in *Proceedings of the 5th international conference on Supercomputing*. ACM, 1991. 97

[61] X. Gu, J. Yang, X. Wu, C. H. and P. Liu, “An efficient architectural design of hardware interface for heterogeneous multi-core system,” in *Proceedings of the 8th IFIP international conference on Network and parallel computing*. Springer-Verlag, 2011. 117

[62] K. Ravindran, N. Satish, Y. Jin, and K. Keutzer, “An fpga-based soft multiprocessor system for ipv4 packet forwarding,” 2005. 117

[63] C .Boneti, R. Gioiosa, F. Cazorla, and M. Valero, “A dynamic scheduler for balancing HPC applications,” in *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, 2008. 117

[64] Cedric Augonnet, et al., “Data-Aware Task Scheduling on Multi-accelerator Based Platforms,” in *ICPADS’10*. 117

[65] Chun Liu, Anand Sivasubramaniam and Mahmut Kandemir, “Organizing the Last Line of Defense before Hitting the Memory Wall for CMPs,” *High-Performance Computer Architecture, International Symposium on*, 2004. 117

[66] Stitt, Greg, et al., “Traversal caches: a first step towards FPGA acceleration of pointer-based data structures,” in *CODES+ISSS’08*. 117, 198

[67] Krishnamurthy A., Culler D. E., Dusseau, A., Goldstein, S. C., Lumetta S., von Eicken T. and Yelick K., “Parallel programming in Split-C,” in *Proceedings of the 1993 ACM/IEEE conference on Supercomputing*, 1993. 117

[68] Hatcher P. J., Quinn M. J., Lapadula A. J., Seevers B. K., Anderson R. J. and Jones R. R., “Data-Parallel Programming on MIMD Computers,” *IEEE Trans. Parallel Distrib. Syst.* 117

[69] A. Cedric, Clet-Ortega, J. Thibault, S. Namyst and Raymond, "Data-Aware Task Scheduling on Multi-accelerator Based Platforms," in *Proceedings of the 2010 IEEE 16th International Conference on Parallel and Distributed Systems*, 2010. 117

[70] Meenderinck C. and Juurlink Ben, "A Case for Hardware Task Management Support for the StarSS Programming Model," in *Proceedings of the 2010 13th Euromicro Conference*. IEEE Computer Society, 2010. 117, 213

[71] Sally A. McKee, "Reflections on the memory wall," in *In Proceedings of the 1st conference on computing frontiers,* , 2004. 117

[72] Awasthi Manu, Nellans David W., Sudan Kshitij, Balasubramonian Rajeev and Davis Al, "Handling the problems and opportunities posed by multiple on-chip memory controllers," in *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, 2010. 118

[73] Nuvacon Corporation, *Multi-DRAM Controller*, 2004. [Online]. Available: www.nuvacon.com 118, 179

[74] "Xilinx Integrated Software Enviroment Design Suite (ISE) Version 11." [Online]. Available: http://www.xilinx.com/support/techsup/tutorials/tutorials11.htm 123, 138

[75] "Xilinx Platform Studio (XPS) Version 11." [Online]. Available: http://www.xilinx.com/support/documentation/dt_edk_edk11-1.htm 123, 138

[76] "Xilinx Power Estimator (XPE) Version 14.3." [Online]. Available: http://www.xilinx.com/ise/power_tools 123, 138

[77] Xilinx , "Xilkernel," December , 2006. [Online]. Available: www.xilinx.com/ise/embedded/edk91i_docs/xilkernel_v3_00_a.pdf 123, 141, 173

[78] Scogland, et al., "A first look at integrated GPUs for green high-performance computing," *Computer Science - Research and Development*. 128

[79] "Stratix IV GX FPGA Development Kits." [Online]. Available: http://www.altera.com/products/devkits/altera/kit-siv-gx.html 131

[80] Xilinx Artix-7, "Leading System Performance per Watt for Cost Sensitive Applications," 2012. [Online]. Available: http://www.xilinx.com/products/silicon-devices/fpga/artix-7/index.htm 131

[81] Halfhill T.R., “Tabula’s time machine,” *Microprocessor Report*, 2010. 131

[82] Kirk Saban, “Xilinx Stacked Silicon Interconnect Technology Delivers Breakthrough FPGA Capacity, Bandwidth, and Power Efficiency,” in *White Paper: Virtex-7 FPGAs*, 2011. 131

[83] Wulf Wm. A. and McKee Sally A., “Hitting the memory wall: implications of the obvious,” *SIGARCH Comput. Archit. News*, 1995. 132

[84] Sally A. McKee, “Reflections on the memory wall,” *ACM: Proceedings of the 1st conference on Computing frontiers, 2004*. 132

[85] M.C.McFarland, A.C.Parker, and R.Camposano, “The high-level synthesis of digital systems,” *Proc. IEEE, vol. 78, no. 2*, 1990. 132

[86] PLDA, *PCI Express XpressLite2 Reference Manual*, February 2010. [Online]. Available: http://www.plda.com/prodetail.php?pid=102 140

[87] Xilinx, *LogiCORE IP I/O Module*, October, 2012. 140

[88] Xilinx LogiCORE IP, *Multi-Port Memory Controller (MPMC)*, March 2011. 140

[89] “Xilinx Software Development Kit (SDK).” [Online]. Available: http://www.xilinx.com/tools/sdk.htm 141

[90] “Embedded System Tools Reference Manual EDK 13.1.” [Online]. Available: www.xilinx.com/support/documentation/sw_manuals/xilinx13_2/est_rm.pdf 141, 206

[91] “SiSoftware Sandra 2013.” 147

[92] Vajda, András and Brorsson, Mats and Corcoran, Diarmuid, *Programming many-core chips*. Springer, 2011. 169

[93] Steinke Stefan, Grunwald Nils, Wehmeyer Lars, Banakar Rajeshwari, Balakrishnan M, Marwedel Peter, “Reducing energy consumption by dynamic copying of instructions onto onchip memory,” in *System Synthesis, 2002. 15th International Symposium on*. 197

[94] Panda Preeti Ranjan, Dutt Nikil D and Nicolau Alexandru, *Memory issues in embedded systems-on-chip: optimizations and exploration*. Springer, 1999. 197

[95] Suhendra Vivy, Mitra Tulika, Roychoudhury Abhik and Chen Ting, "WCET centric data allocation to scratchpad memory," in *Real-Time Systems Symposium, 2005. RTSS 2005. 26th IEEE International*. 197

[96] Deverge J-F and Puaut Isabelle, "WCET-directed dynamic scratchpad memory allocation of data," in *Real-Time Systems, 2007. ECRTS'07. 19th Euromicro Conference on*. 198

[97] Udayakumaran Sumesh, Dominguez Angel and Barua Rajeev, "Dynamic allocation for scratch-pad memory using compile-time decisions," *ACM Transactions on Embedded Computing Systems (TECS)*. 198

[98] James Coole, John Wernsing and Greg Stitt, "A Traversal Cache Framework for FPGA Acceleration of Pointer Data Structures: A Case Study on Barnes-Hut N-body Simulation," in *ReConFig'09*. 198

[99] Mellor-Crummey et al., "Improving memory hierarchy performance for irregular applications," *In ICS '99*. 199

[100] C. Pedro and P. Joonseok, "Data Search and Reorganization Using FPGAs: Application to Spatial Pointer-based Data Structures." 199

[101] Yu, Jason et al., "Vector processing as a soft processor accelerator," *ACM Transactions on Reconfigurable Technology and Systems, 2009*. 199

[102] Severance Aaron et al., "VENICE: A compact vector processor for FPGA applications," in *International Conference on Field-Programmable Technology 2012*. 199

[103] Zhuravlev Sergey, Blagodurov Sergey and Fedorova Alexandra, "Addressing shared resource contention in multicore processors via scheduling," in *ACM SIGARCH Computer Architecture News, 2010*. 200

[104] "AMBA 4 AXI," http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ihi0022e/index.html, 2013. 200

[105] IBM CoreConnect, "PLB Crossbar Arbiter Core," 2001. 200

[106] P. Marchand and P. Sinha, "A hardware accelerator for controlling access to multiple-unit resources in safety/time-critical systems." Inderscience Publishers, April 2007. 201

[107] L. Yan, W. Hu, T. Chen, Z. Huang, "Hardware Assistant Scheduling for Synergistic Core Tasks on Embedded Heterogeneous Multi-core System," in *Journal of Information & Computational Science (2008).* 201

[108] Lu Jiwei, Das Abhinav, Hsu Wei-Chung, Nguyen Khoa and Abraham Santosh G, "Dynamic helper threaded prefetching on the Sun UltraSPARC® CMP processor," in *Microarchitecture, 2005. MICRO-38. Proceedings. 38th Annual IEEE/ACM International Symposium on.* IEEE, 2005, pp. 12–pp. 201

[109] Chaudhry Shailender et al., "Simultaneous speculative threading: a novel pipeline architecture implemented in sun's rock processor," *ACM*, 2009. 201

[110] Steffan J Greggory, Colohan Christopher B, Zhai Antonia and Mowry Todd C, *A scalable approach to thread-level speculation.* ACM, 2000. 201

[111] Rixner Scott et al., "Memory access scheduling," in *ACM SIGARCH Computer Architecture News*, 2000. 201

[112] Shao Jun et al., "A burst scheduling access reordering mechanism," in *High Performance Computer Architecture, HPCA 2007.* 201

[113] Kim, Yoongu et al., "ATLAS: A scalable and high-performance scheduling algorithm for multiple memory controllers," in *HPCA, 2010.* 201

[114] Bojnordi Mahdi Nazm and Ipek Engin, "PARDIS: a programmable memory controller for the DDRx interfacing standards." 202

[115] Grun Peter et al., "APEX: access pattern based memory architecture exploration," in *14th international symposium on Systems synthesis*, 2001. 202

[116] Jesus Corbal, et al., "Command Vector Memory Systems: High Performance at Low Cost," in *PACT'01.* 202

[117] D. Callahan, K. Kennedy, and A. Porterfield., "Software prefetching," *In Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems.* 203

[118] A. K. Porterfield., "Software Methods for Improvement of Cache Performance on Supercomputer Applications," *PhD thesis, Department of Computer Science, Rice University,.* 203, 204

[119] A. C. Klaiber and H. M. Levy., " Architecture for software-controlled data prefetching." *In Proceedings of the 18th Annual International Symposium on Computer Architecture*. 203

[120] E. H. Gornish., " Compile time analysis for data prefetching," *Master's thesis, University of Illinois at Urbana-Champaign.* 203

[121] E. Gornish, E. Granston and A. Veidenbaum., " Compiler-Directed Data Prefetching in Multiprocessors with Memory Hierarchies." *In International Conference on Supercomputing,.* 203

[122] R. L. Lee., "The Effectiveness of Caches and Data Prefetch Buffers in Large-Scale Shared Memory Multiprocessors." *PhD thesis, Department of Computer Science, University of Illinois at Urbana-Champaign*, May 1987. 204

[123] Ganusov Ilya and Burtscher Martin, "Efficient emulation of hardware prefetchers via event-driven helper threading," in *Proceedings of the 15th international conference on Parallel architectures and compilation techniques*, 2006. 204

[124] Gornish Edward H. and Veidenbaum Alexander, "An Integrated Hardware/Software Data Prefetching Scheme for Shared-Memory Multiprocessors;," *International Journal of Parallel Programming*, 1999. 204

[125] Roth, Amir and Moshovos, Andreas and Sohi, Gurindar S., "Dependence based prefetching for linked data structures," in *ASPLOS'98*. 205

[126] Keith I. Farkas, Norman P. Jouppi and Paul Chow, "How Useful Are Non-blocking Loads, Stream Buffers, and Speculative Execution in Multiple Issue Processors?" 1994. 205

[127] Norm Jouppi, "Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers," 28-31 May 1990. 205

[128] McGowan Steven, "Scatter gather emulation," 2013, US Patent 8,510,759. 205

[129] M Wen, N Wu, C Zhang, W Wu, Q Yang and C Xun, "FT64: Scientific Computing with Streams," *High Performance Computing HiPC 2007.* 205

[130] M Wen, N Wu, C Zhang, Q Yang, J Ren, Y He, W Wu, J Chai, M Guan, C Xun, "On-Chip Memory System Optimization Design for the FT64 Scientific Stream Accelerator," *Micro IEEE 2008.* 205

[131] Sek M. Chai, N. Bellas, M. Dwyer and D. Linzmeier, “Stream Memory Subsystem in Reconfigurable Platforms.” 2nd Workshop on Architecture Research using FPGA Platforms, 2006. 205

[132] Gou, Chunyang, Kuzmanov Georgi and Gaydadjiev Georgi N., “SAMS multi-layout memory: providing multiple views of data to boost SIMD performance,” 2010. 206

[133] Lattice Semiconductor Corporation, *Scatter-Gather Direct Memory Access Controller IP Core Users Guide*, October 2010. 206

[134] A. Corporation, *Scatter-Gather DMA Controller Core, Quartus II 9.1*, November 2009. 206

[135] Sally A. McKee, et al., “Dynamic Access Ordering for Streamed Computations,” *IEEE Trans. Computer. November 2000.* 206

[136] John Carter, Wilson Hsieh, Leigh Stoller, Mark Swanson, Lixin Zhang, Erik Brunvand, Al Davis,Chen-Chi Kuo, Ravindra Kuramkote,Michael Parker, Lambert Schaelicke, and Terry Tateyama, “Impulse: Building a Smarter Memory Controller,” *Fifth International Symposium on High Performance Computer Architecture (HPCA-5)*, January 1999. 206

[137] Pellizzer Fabio and Bez Roberto, “Non-volatile semiconductor memories for nano-scale technology,” in *Nanotechnology (IEEE-NANO), 2010 10th IEEE Conference on.* 212

[138] Satyamoorthy Prateeksha and Parthasarathy Sonali, “MRAM for Shared Memory in GPGPUs,” *Captured from http://www. cs. virginia. edu/˜ sp5ej/MRAM. pdf*, 2010. 212

[139] Ren Wanchun, Jing XZ, Xiang YH, Xiao HB, Zhang BC, Liu B, Song ZT, Rao F, Xu J, Wu GP et al., “(Invited) Thin Film Challenges of Phase Change Random Access Memory,” *ECS Transactions*, 2013. 212

[140] Eshraghian Kamran, Cho Kyoung-Rok, Kavehei Omid, Kang Soon-Ku, Abbott Derek and Kang Sung-Mo Steve, “Memristor MOS content addressable memory (MCAM): Hybrid architecture for future high performance search engines,” *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 2011. 212

[141] Choi Yoonsuk and Latifi Shahram, “Future prospects of DRAM: emerging alternatives,” *International Journal of High Performance Systems Architecture*, 2012. 212

[142] “European Technology Platform for High Performance Computing ,” http://www-hpc.cea.fr/docs/2013/ETP4HPC_book_singlePage.pdf, 2011. 212

[143] “Cell Superscalar (CellSs) User Manual (Barcelona Supercomputing Center),” May 2009. 213

Declaration

I herewith declare that I have produced this work without the prohibited assistance of third parties and without making use of aids other than those specified; notions taken over directly or indirectly from other sources have been identified as such. This work has not previously been presented in identical or similar form to any other Spanish or foreign examination board.

The thesis work was conducted from `September 2009` to `December 2014` under the supervision of Prof. Eduard Ayguadé, Prof. Mateo Valero, Dr. Adrian Cristal, Dr. Osman S. Ünsal and Dr. Oscar Palomar.

Tassadaq Hussain,
Barcelona, December 2014.