J. Parallel Distrib. Comput. 106 (2017) 31–49

Contents lists available at ScienceDirect

J. Parallel Distrib. Comput.

journal homepage: www.elsevier.com/locate/jpdc

A novel hardware support for heterogeneous multi-core memory system

Tassadaq Hussain *

UCERD: Unal Color of Education Research and Development, Islamabad, Pakistan
Riphah International University, Islamabad, Pakistan
Microsoft Research Center and Barcelona Supercomputing Center, Spain

HIGHLIGHTS

- Support for both static and dynamic data-structures and memory access patterns.
- Specialized scratchpad memory is integrated to map complex access patterns.
- The data manager accesses, reuses and feeds complex patterns to the processing core.
- Complex patterns are managed at run-time, without the support of a master core.
- Can be integrated with soft/hard soft processor core.
- Support trace driven simulation and FPGA based real prototyping environment.

ARTICLE INFO

Article history:
Received 22 July 2015
Received in revised form
20 January 2017
Accepted 21 February 2017
Available online 16 March 2017

Keywords:
Heterogeneous
Controller
HPC

ABSTRACT

Memory technology is one of the cornerstones of heterogeneous multi-core system efficiency. Many memory techniques are developed to give good performance within the lowest possible energy budget. These technologies open new opportunities for the memory system architecture that serves as the primary means for data storage and data sharing between multiple heterogeneous cores. In this paper, we study existing state of the art memories, discuss a conventional memory system and propose a novel hardware mechanism for heterogeneous multi-core memory system called Pattern Aware Memory System (*PAMS*). The PAMS supports static and dynamic data structures using descriptors and specialized scratchpad memory. In order to prove the proposed memory system, we implemented and tested it on real-prototype and simulator environments. The benchmarking results on real-prototype hardware show that PAMS achieves a maximum speedup of 12.83x and 1.23x for static and dynamic data structures respectively. When compared to the *Baseline System*, the PAMS consumes up to 4.8 times less program memory for static and dynamic data structures respectively. The PAMS consumes 4.6% and 1.6 times less dynamic power and energy respectively. The results of simulator environment show that the PAMS transfers data-structures up to 5.12x faster than the baseline system.

1. Introduction

For a long era, speed was one and only source which was used for estimating the performance of any High Performance Computing (HPC) systems. Both, raw computing performance and performance per watt are equally important. Nowadays, power consumption and power density determine the system performance [18]. Besides, supercomputers, servers, and data centers also have the power constraints. To achieve the performance, the computer architectures use heterogeneous accelerator cores. The system architects are targeting low power and high-performance HPC systems having general purpose processing cores [49] and application specific accelerators [61].

As the amount of on-chip gates increases, there is a dramatic increase in size and architecture of local memories such as caches. The concept of scratchpad memory [3] is an important architectural consideration in modern HPC systems, where advanced technologies have made it possible to combine with DRAM. With the unveiling of on-chip memories such as memristors [25] and embedded DRAMs [47] the size of on-chip memory is getting a dramatic increase. Embedded DRAM (eDRAM) has the advantage of having much higher density and lower power consump-

* Correspondence to: UCERD: Unal Color of Education Research and Development, Islamabad, Pakistan.
E-mail address: tassadaq@ucerd.com.

http://dx.doi.org/10.1016/j.jpdc.2017.02.008
0743-7315/

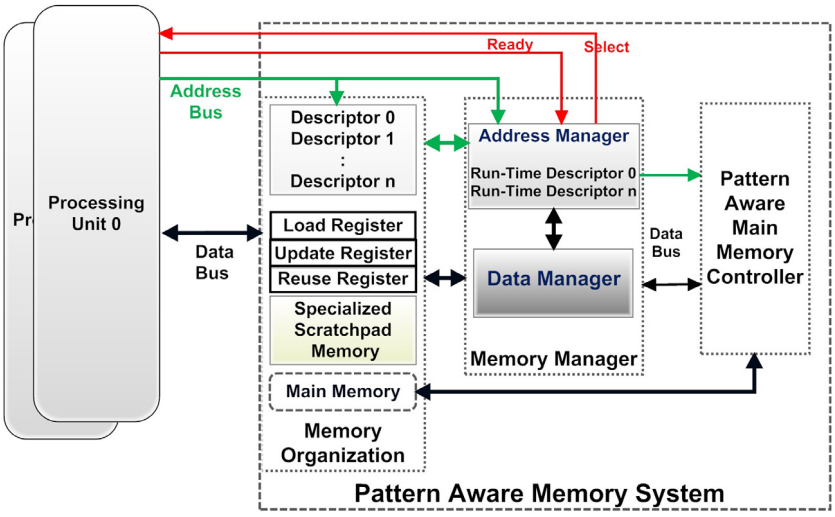

Fig. 1. Architecture of Pattern Aware Memory System.

tion than the traditional SRAM technology used for local memory. Integrating eDRAM on the same silicon chip with the processing logic holds the promise of greater bandwidth access than through external interfaces. Although the data read and write latency remains to be a bottleneck for new memory technologies. Having huge *Local Memory* as shared memory still requires a memory system in hardware and/or software that hides the on-chip communication mechanism between the applications.

Integrating intelligent registers [69,42] inside application specific processors improves the performance of the memory hierarchy. The register memory enhances the performance for applications having data locality but does not support applications with large and dynamic data-structures. Since different applications have different memory access patterns and data-structures, finding one topology that fits well for all applications is difficult. Integrating more memory controllers on system platform can increase bandwidth, but require many Input/output pins that consume power. Therefore, a memory system requires an intelligent memory controller that manages and schedules the data accesses.

Increasing the number of memories and placing them near the processing core is not the only way to address and improve the performance of applications. We present a technique to minimize the on-chip data transfer execution time of shared/distributed systems by a careful partitioning of the local memory, access patterns, and schedule and manage them in hardware. In this work, we propose the Pattern Aware Memory System (PAMS), a memory system for heterogeneous multi-core architectures. PAMS accelerates both static and dynamic data structures and their access patterns by arranging memory accesses to minimize access latency based on the information provided by pattern descriptors. PAMS operates independently from the master core at run-time. PAMS keeps data-structures and access pattern descriptors in a separate memory and prefetches the entire data structure into a special scratchpad memory. Data-structures and memory accesses are arranged in the pattern descriptors at program time and PAMS manages multiple patterns at run-time to reduce access latency. PAMS controls data movement between the *Main Memory* and the specialized scratchpad memory; data present in the *Local Memory* is reused and/or updated when accessed by several patterns. The significant contributions of the proposed PAMS architecture are:

- Support for both static and dynamic data structures and memory access patterns using the memory pattern descriptors thus reducing the impact of memory latency.
- A specialized scratchpad memory that is tailored for the *Local Memory* organization and maps complex access patterns.
- A data manager that efficiently accesses, reuses and feeds data to the computing unit.
- Data management and handling of complex memory access at run-time, without the support of a processor or the operating system.
- When compared to the *ARM based Multi-core System*, PAMS achieves between 1.35× to 12.83× and 1.12× to 1.23× of speedup for applications having static and dynamic data structures respectively. The PAMS consumes 4.6% and 1.6 times less dynamic power and energy respectively.
- In a many-core trace-driven simulation environment, the PAMS transfers data-structures up to 5.12× faster than the baseline system.

The rest of this paper is organized as follows: Sections 2 and 3 describe PAMS and Programming Model respectively. Sections 4 and 5 present the evaluation architectures and results respectively. Finally, Section 6 discusses the related work and Section 7 provides the conclusions.

2. Pattern Aware Memory System (PAMS)

The proposed PAMS (shown in Fig. 1) performs isolation and management of data-structures using *Specialized Scratchpad Memory* and improves data transfers by arranging access requests using *Descriptor Memory*. The *Descriptor Memory* manages data structures using single or multiple descriptor blocks and organizes data access requests in the form of patterns that reduce the run-time address generation and management overhead and avoids request grant time. The *Descriptor Memory* manages compile-time as well as run-time generated memory accesses of the applications. The PAMS *Memory Manager* handles *Specialized Scratchpad Memory* data and performs load, reuse and update data operations that avoid accessing the same data multiple times. At run-time, the PAMS schedules data accesses according to the application needs and applies fair data transfer scheme. The PAMS *Pattern Aware Main Memory Controller* transfers data between the *Main Memory* and *Local Memory* by maximum utilizing multiple DRAM banks. This section is further divided into following subsections, the *Memory Organization*, the *Data-Structures and Access Description*, the *Memory Manager* and the *Pattern Aware Main Memory Controller*.

2.1. Memory organization

To provide isolation and improve data locality the PAMS memory is subdivided into four sections that are: the *Descriptor Memory*, *Buffer Memory*, *Specialized Scratchpad Memory* and *Main Memory*.

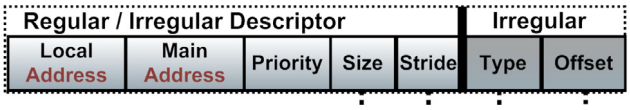

Fig. 2. Descriptor memory structure: Regular and irregular access patterns.

2.1.1. Descriptor Memory

The PAMS uses pattern descriptors to organize the *Regular* and *Irregular* access patterns [30]. These pattern descriptors reduce the overhead of run-time address generation and data management. The PAMS *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 Fig. 2) 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 processed. *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 memory addresses of a stream. The PAMS handles strides between two consecutive accesses.

The parameters for *Irregular Descriptor Memory* [30] accesses are also presented in Fig. 2. The *Type* and *Offset* register fields define the category of irregular pattern and the location of the next element respectively. 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 next access pattern by adding it to the *Main Address*. 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 the *Descriptor Memory* and signifies that PAMS 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 PAMS to gather the next address at run-time from the processing core. The *dependent unknown* patterns are not programmed at compile-time, and the PAMS address manager handles these transfers at run-time and manages them at run-time *Descriptor Memory*. PAMS bus system [28] uses separate (*request* and *grant*) signals (shown in Fig. 1) to read addresses from the compute unit. C/C++ calls such as `SEND()` and `SEND_IRREGULAR()` are used to program the descriptor blocks.

2.1.2. 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 and uses the realignment feature.
- In-order data delivery. In cooperation with the Memory Manager (see Section 2.3) that prefetches data, it ensures that the data of one pattern is sent in-order to the processing core.

The *Buffer Memory* holds three buffers which are the *load buffer*, the *update buffer* and the *reuse buffer*. The *Buffer Memory* is managed by the *Memory Manager*. 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* data (see Section 2.1.3). 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.

2.1.3. Specialized Scratchpad Memory

The system provides a *Specialized* on-chip memory that improves system performance by prefetching complete patterns [29]. As the conventional *Local Memory* system, the *Specialized Scratchpad Memory* (*SSM*) accesses the whole block as a cache line and temporarily holds data to speed up later accesses. Unlike the *Local 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 PAMS allocates separate *Descriptor* block for each processing core. Unlike the *Local Memory*, system which transfers an aligned block of data for each data miss, the PAMS transfers only the missing data by scattering/gathering operation at run-time and transfers irregular blocks of data.

The PAMS *SSM* is directly connected to the compute unit and provides single cycle data access. The *SSM* temporarily holds data to speed up later accesses. 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 data access operation. To exploit parallelism better, the banks of *SSM* are organized physically 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 Fig. 3(a). 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 one central point. A 3D *SSM* architecture that accommodates the stencil is shown in Fig. 3(b). 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 a 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, PAMS 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 can be placed on the 3D *SSM*.

In the current architecture, the number of banks is parameterizable that can be changed before the synthesis process. The row and column size is programmable and can be changed depending upon the dimensions of a 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 PAMS *Descriptor Memory*. The PAMS accesses and places data in tiles if the data set is larger than the *SSM* structure.

2.1.4. Main Memory

The *Main Memory* is the slowest type of memory having DRAM and is accessible by the whole system. Even though, PAMS has an efficient way of accessing the *Main Memory* [26] that best utilizes the bandwidth, it still has latency with respect to local memory.

2.2. Data-structures and access description

Just like the cache memory hierarchy, the PAMS supports two types of data-structure classes for memory allocation that are: the

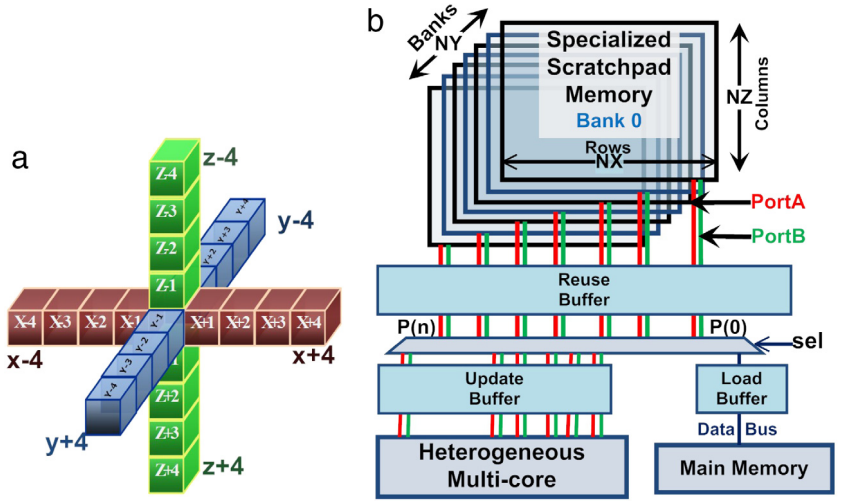

Fig. 3. (a) 3D-Stencil. (b) Buffer memory organization and Specialized Scratchpad Memory structure.

Static Data-structure and the *Dynamic Data-structure*. The *Static* and *Dynamic Data-structures* are initialized at compile-time and run-time respectively. The PAMS *Static Data-structure* method allocates the memory that, usually, has a dense data set with predictable and aligned access patterns. The *Dynamic Data-structure* has a data set with compile time unpredictable and unaligned data access requests.

Multiple unaligned and random data access requests of memory lead to a high degree of data transfer delay due to the control selection of processing core, bus unit and DRAM memory. The PAMS overcomes this problem by organizing *Static* and *Dynamic* data-structures requests in descriptors. The PAMS uses two types of descriptors: the *Regular* and the *Irregular Descriptors* for *Static* and *Dynamic Data-structures* respectively. The *Regular Descriptor Memory* holds information of *Static Data-structures* (1D/2D/3D arrays) and their access patterns. The *Static Data-structures* are aligned concerning memory addressing and manages data structures that have compile-time predictable and aligned data accesses. The *Irregular Descriptor Memory* holds information about unaligned *Dynamic Data-structures* (tree-based) and their access patterns. The *Irregular Descriptor Memory* allocates as much memory as necessary during run-time. The size of the allocated memory can vary between executions of the program.

The PAMS categorizes access patterns into three sections the *compile-time predictable*, the *run-time predictable* and the *run-time unpredictable*.

Compile-time predictable. The compile time predictable accesses are managed in *Descriptor Memory* before execution. At run-time, PAMS reads the descriptor information, tailors it before execution and transfers the variables to the *Specialized Scratchpad Memory*.

- The aligned access patterns having regular strides are managed at compile-time using size and *Stride* parameter of a regular descriptor block.
- The unaligned accesses are managed in *Irregular Descriptor Memory* using *Offset* parameters.

Run-time predictable. The run-time predictable accesses are managed in *Descriptor Memory* at run-time in parallel with execution using *Address Manager* (see Section 2.3).

- For data access having aligned data and regular strides, PAMS gathers requests from processing core and manages requests in a *Regular Descriptor Memory* using stride and size parameters.
- For unaligned and irregular strided data access, PAMS organize the data access requests in *Irregular Descriptor Memory* using an offset parameter.

Run-time unpredictable. For access patterns with run-time unpredictable accesses, PAMS uses separate control (*select & ready*) signals to communicate with the processing unit at run-time.

2.3. Memory Manager

The PAMS *Memory Manager* (shown in Fig. 4) organizes and rearranges multiple noncontiguous memory accesses simultaneously that reduces read/write delay due to the control selection of DRAM memory. The PAMS *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. PAMS keeps the knowledge of memory as to whether or not an individual memory area is in the *SSM*. This knowledge allows the PAMS to manage the placement of memory, as well as reuses and shares already accessed memory. The *Memory Manager* controls the *Local Memory* system (descriptor and *SSM*). It handles a *Descriptor Memory* Pointer (DMP) and a *SSM* Pointer (SSMP) as shown in Fig. 4. 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-accelerator), the *SSP* is divided into multiple blocks. The SSMP takes the source address (*Base Address*) from the *Descriptor Memory* and with single cycle latency starts incrementing in it. The SSMP stops incrementing addresses, when the `ack` signal is granted by the *Pattern Aware Main Memory Controller* (see Section 2.5). The *Memory Manager* is further divided into three sections which are: the *Address Manager*, the *Data Manager* and the *Scheduler*.

2.3.1. Address Manager

The *Address Manager* fetches single or multiple descriptors depending on the access pattern, translates/reorders in hardware, in parallel with PAMS Read/Write operations and access data patterns of a *Processor Core*. The *Address Manager* uses one or multiple descriptors at run-time to describe the data access. Unlike the cache that transfers an aligned block of data for each data miss, the PAMS *Address Manager* accesses only the missed data for scratchpad by gathering address requests at run-time and transfers irregular blocks of data. The *Address Manager* reads compile time generated addresses from *Descriptor Memory* and accesses complex data pattern.

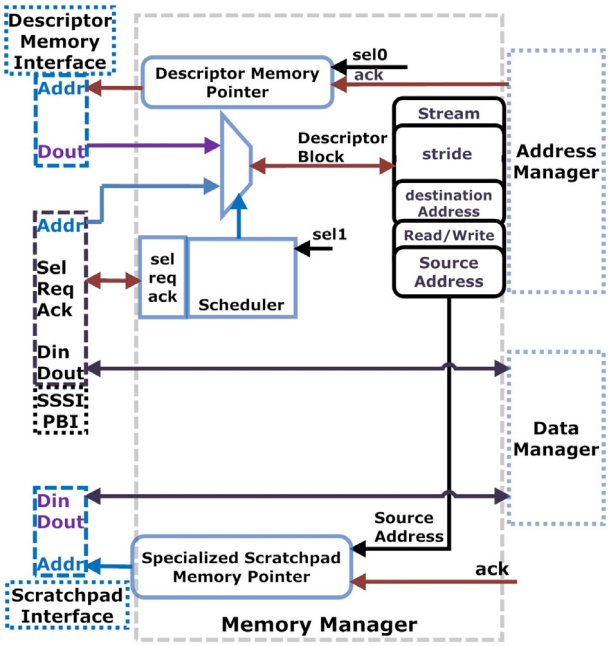

Fig. 4. PAMS: Memory Manager.

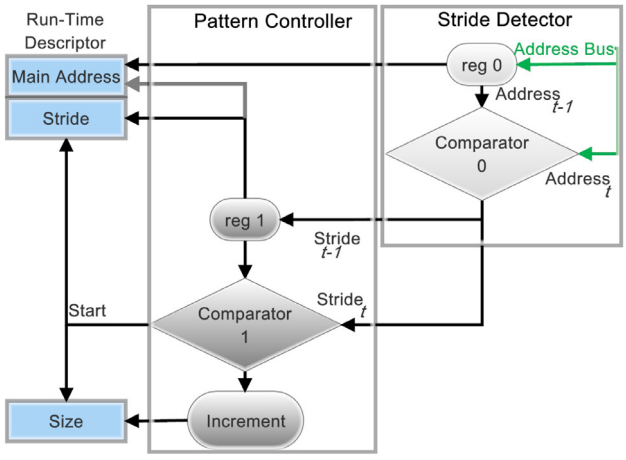

Fig. 5. Run time managed descriptor block.

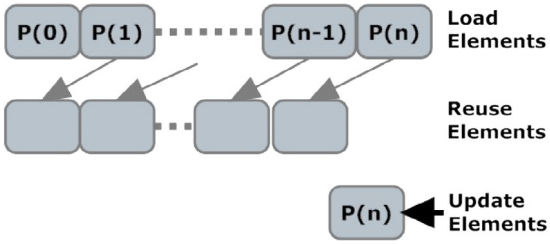

Fig. 6. Load, update and reuse of FIR access pattern.

The *Address Manager* also manages run-time unpredictable memory accesses and places them in *Descriptor Memory*. The *Address Manager* manages descriptor blocks by taking 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 Fig. 5. The *Address Manager* 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 generate stride which is given to *reg* 1 and *comparator* 1. The *comparator* 1 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* 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.

2.3.2. Data Manager

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 Fig. 3). The *reuse buffers* temporarily holds loaded elements for further reuse. For each 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 Section 2.5). 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 Fig. 6) 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 the *c/a* ratio is $1/16 = 0.062$. The *Data Manager* improves *c/a* ratio to 1 by reusing accessed points.

2.4. Scheduling

The PAMS *scheduler* manages and controls the run-time memory requests depending on the access pattern transfer size and program priorities. Each data structure is assigned a *priority* value along with the *Specialized Scratchpad Memory* address, which are placed in the registers of the scheduler. The PAMS *Scheduler* supports programmed and automatic scheduling policy [27]. The scheduling policies parameters are programmed statically at program time and are executed by hardware at run-time. The program strategy emphasizes on priority and incoming requests of the processing cores.

2.5. Pattern Aware Main Memory Controller

The *Pattern Aware Main Memory Controller* (PAMMC) (Fig. 7) accesses data to/from DRAM memory and aims to improve memory accesses throughput and reduces memory access latency by applying multi-bank management policy that reduces the decoding time of DRAM bank and the row address. The time required by the *PAMMC* to access data to/from a *DRAM* memory is divided into three categories.

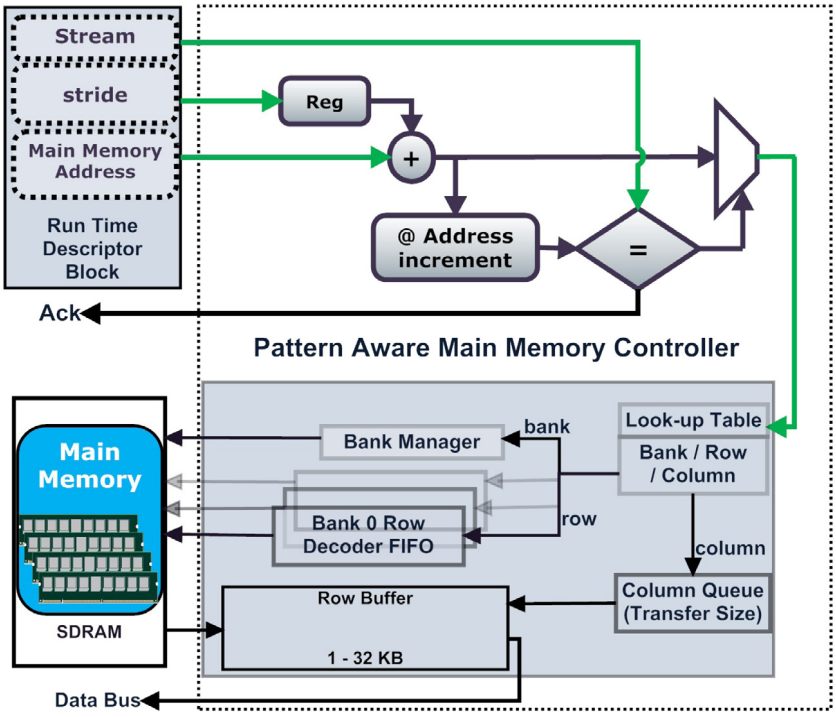

Fig. 7. Pattern Aware Main Memory System.

- Row hit: When the data is available in a row buffer, the *PAMMC* issues only a read (RD) or write (WR) command to the DRAM bank.
- bank hit: When a row is not open in the row-buffer and *PAMMC* needs to first issue an activate (ACKT) command to open the required row, then a read/write command.
- Bank conflict: When data access requires data from a bank that is not open. This requires *PAMMC* 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.

PAMMC takes memory addresses from *Memory Manager*, performs the address mapping from physical address to DRAM address and read/write data to/from DRAMs. The PAMS *PAMMC* can integrate multiple DRAMs that can increase the memory bandwidth. The memory address holds the DRAM bank, the row address, column address and chip select. *PAMMC* supports two possible modes of operations for bank management and to manage the row-buffer: 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 maintains multiple banks opened 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 accesses data from multiple banks in parallel.

PAMMC uses descriptors for access patterns that improve the memory bandwidth by transferring descriptors to the memory controllers, rather than specific references and by obtaining data from the DRAM device. Unlike a conventional DRAM 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 keep precharged contiguous rows. The *Compile time Predictable* access patterns are very common in the *PAMS*, because addresses are described in descriptors and occurs for around 70% of all memory patterns. The data transfers are organized in descriptors that manage DRAM open banks and rows. Depending upon the complexity of access patterns the *PAMMC* selects a single- or multi-bank mode of operation at run-time. For example, if an access pattern (i.e. 3D Stencil) requests data from multiple banks then the *PAMMC* operates it with the multi-bank mode. The *PAMMC* increases data access locality and ensures that the *row buffer* of a bank that was left open by the previous access pattern will make the best possible use of the next data access. Such kind of access requires only RD/WR commands. The *PAMMC* applies an adaptive multi-bank management policy that handles the row buffers of each bank independently.

The *PAMMC* uses a policy that schedules the memory patterns and dynamically combines them whenever it makes sense to do so. The DRAM addresses are provided to the *Address Look-up Table* of the *PAMMC*. The *Address Look-up Table* arranges incoming access patterns with respect to bank, row and column. For the arrangement of access patterns, the highest priority is given to the pattern that belongs to the same row and bank. This reads/writes next columns without causing contention on a shared system bus and reduces the PRE and ACKT commands overhead.

3. Programming model

Many applications exhibit different data access patterns, forcing the hardware designers to facilitate software programming by holding a general interface among the memory management unit and the processing unit. This ends in a settlement on the achievable performance because of the generic process of calling data transactions. An alternative to achieve higher performance is to avoid the generic interface and manually arrange memory accesses. PAMS programming model aims to remove the programmer effort of manually organizing memory accesses and meet the performance requirements of HPC applications. This facilitates the programmer,

simplify hardware related programming and configuration constraints while using the architectural model. The section is further categorized into *Descriptors and Data Structures* and *PAMS Programming*.

3.1. Descriptors and Data Structures

PAMS supports regular data access patterns such as strided vector and 1D/2D/3D auto tiling. PAMS prefetches multiple regular streams, arranges them according to the predefined patterns and buffers them in *SSM* to perform the access in a single cycle. PAMS *Data Manager* handles these accesses in parallel with computation. A single descriptor holds complex access pattern information (row, column, diagonal vector, etc.). For example, a 3D stencil memory requires 3 descriptors for accessing row, column and bank vectors, each descriptor with stride of one, *row_size* and *row_size* × *column_size* length respectively. In this section we elaborate the functionality of PAMS to support irregular data structures (pointer, linked-list, tree etc.) that are mainly used to represent application's data containing a hierarchical relationship between elements. The overall structure of a Linked List is described by using the *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 PAMS Linked List data structure allocates space for each data element separately using the *Irregular Descriptor Memory*. The *Type* 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. For *n*-body simulation a tree.h file is provided which has specifications for the data structures and its access pattern used in tree construction. A similar tree-code in software is used by Joshua E. Barnes.

3.2. PAMS programming model

The proposed PAMS provides comprehensive support for the C and C++ languages. Fig. 8 shows the C structure that is used to describe the PAMS *SSM* and the Main Memory data set. The structure (Fig. 8(a)) is used to define the local buffer and the data set of an application kernel. The access patterns in PAMS are managed with functions such as `SEND()` and `SEND_IRREGULAR()`, which uses single descriptor. Specialized function calls (e.g. `MEM_CPY()`, `STENCIL_VECTOR()`, `etc.`) are provided for complex and dense access patterns. These function calls use more than one descriptor. 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. *Part I* of Fig. 8(a) defines a 3D *SSM* (`SCRATCHPAD`) of 32 × 32 × 3. *Part II* identifies the *Main Memory* data set information. *Part III* shows a function call that initializes a 3D data transfer between the *Main Memory* and the *SSM*. The section is further categorized into *Programming PAMS*, *Direct Memory Access*, *3D-Stencil Data Transfer*, and *Multi-core Programming*.

3.2.1. Programming PAMS

The PAMS program flow is shown in Fig. 8(b). The PAMS program code is compiled using the GCC compiler and Intel Pin tool [41] for real-prototype and trace-based simulation environments respectively. PAMS uses a wrapper library that offers function calls to describe the application complex access patterns. The memory access information is included in the PAMS header file and provides function calls (e.g. `STENCIL()`, `MEM_CPY()`, etc.) that require basic information of local memory and data set. The PAMS library takes the data transfer information and fills the PAMS *Regular Descriptor Memory*. In case of unknown memory patterns the PAMS library uses Pin trace tool which gathers memory traces and places them in *Irregular Descriptor Memory*. The programmer only needs to identify the relevant call to the library function call (e.g. `STENCIL()`, etc.) and the PAMS system automatically transfers the data pattern to the local memory of the compute unit. The function `MEM_CPY()` takes local memory and the *Main Memory* descriptions as input and transfers the memory pattern. The size of a memory access pattern is dependent on the local memory size. For example a `MEM_CPY()` generates 32 `SEND` or `SEND_IRREGULAR` calls at compilation time, to transfer 32 × 32 size of 2D local memory. The function calls improve the performance of application kernels by marshaling data according to the application needs. For the complex memory patterns, single or multiple descriptors are used at compile-time to reshape and unfold data patterns. At run-time these descriptors help PAMS to transfer memory patterns between the processing cores and the *Main Memory* with minimum delay. PAMS decouples data accesses from execution, prefetches complex data structure and prevents processing unit stalls.

3.2.2. Direct Memory Access

Fig. 9 shows complex data transfer examples in multi-core systems using a conventional memory system and PAMS. The conventional system transfers data between the *Main Memory* and the local memory of a processing core using Direct Memory Access (DMA). DMA transfers are specified using the `Dma_Transfer` function call. Fig. 9(a) shows how it is used to transfer 1024 bytes of data stream from local buffer *SrcBuffer* to the *Main Memory DestBuffer*. To transfer a noncontiguous block of data, such a strided memory access pattern, the conventional memory system uses multiple `Dma_Transfer` calls. In PAMS, the `SEND` function call is used to specify descriptors that will be placed in the *Descriptor Memory*. Fig. 9(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, PAMS will access 1024 consecutive bytes. The *PAMS_Buffer* and *PAMS_Dataset* parameters hold the start address of local (data) memory and the *Main Memory* (*SDRAM*) data set respectively. The parameters *Bytesize* and *Offset* define the type of memory access. The PAMS `SEND` function call transfers 1024 bytes having *stride* between two consecutive memory locations. The PAMS `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 Fig. 9(b), *Offset* and *Type* indicates that the next transfer is unknown until run-time. For more complex memory accesses, the PAMS provides `SEND_IRREGULAR` function call that can have multiple *Offset* registers. In both systems the programmer has to describe the *Main Memory* data sets and the local buffer size of application kernels.

3.2.3. 3D-Stencil Data Transfer

A conventional example of 3D stencil access pattern is shown in Fig. 10(a). A 3D stencil memory access has three (row, column and bank) vectors. PAMS removes the programmer effort of manually arrangement of complex memory accesses. A PAMS example of a 3D stencil access is shown in Fig. 10(b). A single stencil can be accessed by using the `STENCIL()` function call. The function requires *SSM* and *Main Memory* data set information. The PAMS uses three descriptors that transfer the three stencil vectors: row, column and bank. PAMS reduces address management by initializing different strides for each vector access. PAMS improves performance and reduces main loop computation by access stencil in the form of vectors. A `STENCIL_VECTOR()` call is used to transfer multiple stencil vectors by using multiple `STENCIL()` function calls. This function call requires 3D-*SSM* and *Main Memory* data set information.

a

```
//SSM Structure
typedef struct PAMS_DATA_MEMORY
    {
    int SP_ADD; // SSP ADDRESS
    int SP_WID; // WIDTH
    int SP_HEI; // HEIGHT
    int SP_BAN; // BANKS
    } SSP_MEMORY
// Main Memory (SDRAM) Structure
typedef struct PAMS_MAIN_MEMORY
    {
    int EXT_ADD; // MAIN MEMORY ADDRESS
    int EXT_WID; // WIDTH
    int EXT_HEI; // HEIGHT
    int EXT_BAN; // BANKS
    } MAIN_MEMORY
```

```
//  Main Program
SSP_MEMORY SSP;
MAIN_MEMORY DATASET;
//Local Memory (Buffer): Part I
SSP.SP_ADD=0X00000000;
SSP.SP_WID=32;
SSP.SP_HEI=32;
SSP.SP_BAN=3;
//Main Memory (Dataset): Part II
DATASET.EXT_ADD=0X00001000;
DATASET.EXT_WID=128;
DATASET.EXT_HEI=128;
DATASET.EXT_BAN=3;
//ACCESS DESCRIPTION : Part III
MEM_CPY(SSP, DATASET,1)
Computation(SSP);
```

b

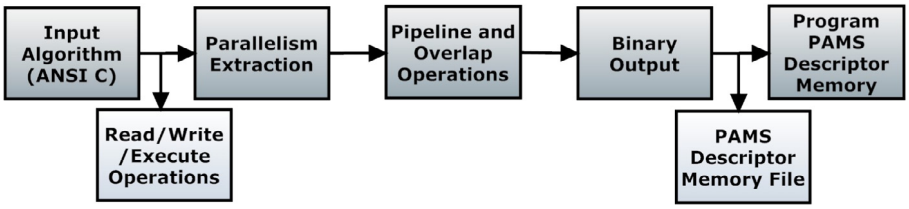

Fig. 8. Examples of: (a) SSM and main memory program structure (b) Program flow.

a

```
/* Instance of the DMA driver */
static DMA DmaInstance;

/* Pointer to local buffer */
int      *SrcBuffer;
SrcBuffer = (int *)  0x00000000;
/* Pointer to Dataset  */
int      *DestBuffer;
DestBuffer = (int *)  0x000010000;
/* Transfer Size in Bytes */
int       Bytesize=1024;
/* Single Transfer */
Dma_Transfer (DmaInstance,
     SrcBuffer, DestBuffer,
     Bytesize);
```

b

```
/*Pointer to Local Memory */
SSP_MEMORY PAMS_Buffer      ;
PAMS_Buffer = (char *)  0x00000000;
/* Pointer to Dataset  */
char      *PAMS_Dataset;
PAMS_Dataset = (char *)  0x000010000;
int       Bytesize=1024;
/* Stride between Elements */
char       stride=0x04;
/* Pointer to Offset Transfer */
int       Offset=0x20000000;
char    Type=0x00; // NULL
/* PAMS Transfer */
SEND( PAMS_Buffer, PAMS_Dataset,
Bytesize, stride, Offset, Type);
```

Fig. 9. DMA transfer function calls: (a) Conventional memory system (b) PAMS.

a

```
// 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];}
}
```

b

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

// Main Program

SSP_MEMORY STENCIL;
SSP_MEMORY SSM_3D;
MAIN_MEMORY DATASET_3D;
// Part I : Local SSM
// Single Stencil Buffer
STENCIL.SP_ADD=0X10000000;
STENCIL.SP_WID=9;
STENCIL.SP_HEI=3;
STENCIL.SP_BAN=1;
// 3D 32x32x32 SSM
SSM_3D.SP_ADD=0X11000000;
SSM_3D.SP_WID=32;
SSM_3D.SP.HEI=32;
SSM_3D.SP_BAN=32;
// Part II : Main Memory
// 3D-Data set
DATASET_EXT_ADD=0X00100000;
DATASET_EXT_WID=128;
DATASET_EXT_HEI=128;
DATASET_EXT_BAN=128;
//PART III : DATA TRANSFER
STENCIL (STENCIL, DATASET_3D, PRIORITY1);
STENCIL_VECTOR (SSM_3D, DATASET_3D,
      PRIORITY2);
```

Fig. 10. 3D Stencil access pattern: (a) Conventional load/store (b) PAMS.

3.2.4. Multi-core programming

A multi-core programming example of PAMS is shown in Fig. 11. The multi-cores are initialized with the data structures and the scheduling policy priorities [31]. Like Pthread scheduling policies the PAMS, scheduler initializes priorities at compile time. Unlike Pthread, the PAMS executes requests at run-time in hardware. If the same priority is assigned to multiple cores, the PAMS scheduler processes the cores in symmetric mode. The PAMS automatically

```
// Main Program
SSM_MEMORY SSM_2D;
SSM_MEMORY 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.ADD =0X00000000;
SSM_2D.WID =32;
SSM_2D.HEI =32;
SSM_2D.BAN =1;
// 3D 32x32x32 SSM
SSM_3D.ADD=0X10000000;
SSM_3D.WID=32;
SSM_3D.HEI=32;
SSM_3D.BAN=32;
// Part II : Main Memory
// 2D-Data set
DATASET_2D.ADD=0X00001000;
DATASET_2D.WID=256;
DATASET_2D.HEI=256;
DATASET_2D.BAN=1;
// 3D-Data set
DATASET_3D.ADD=0X00100000;
DATASET_3D.WID=128;
DATASET_3D.HEI=128;
DATASET_3D.BAN=128;
//PART III : DATA TRANSFER
MEM_CPY (SSM_2D,
DATASET_2D, PRIORITY1);
3D_STENCIL_VECTOR (SSM_3D,
DATASET_3D, PRIORITY2);
// PART IV:
//PROCESSING MULTI-CORES
LAPLACIAN (SSM_2D);
STENCIL (SSM_3D);
```

Fig. 11. Multi-core: PAMS programming example of 2D and 3D applications.

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. The program initializes two *ASHA*s and their 2D and 3D tiled data pattern. *Part I* of Fig. 11 defines a 2D and 3D *SSM* of 32×32 and $32 \times 32 \times 32$ elements respectively. *Part II* identifies main memory data sets information. *Part III* initializes 2D and 3D data transfers. The PAMS uses `MEM_CPY` function call to read/write dense data structure and access pattern. The minimum requirement of each access pattern using `MEM_CPY` 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 Stencil processing cores. The Laplacian and Stencil kernels require 2D and complex 3D data patterns respectively. The PAMS supports different processing cores having RISC, CISC and ASIP architectures.

4. Experimental framework

In this section, we describe and evaluate the PAMS on a real-prototype and simulation environments. The section is divided into three subsections: the *Application Kernels and Processing Cores*, the *ARM Cortex-A9 Evaluation System* and the *Tasksim Simulator*

4.1. Application Kernels and Processing Cores

Fig. 12 shows the application kernels that are used in the design. Column *Pattern* presents access patterns of the application kernels. Each color represents a separate memory transfer pattern. The column *Size Byte* of Fig. 12(b) describes the program memory size in bytes.

The Reduced Instruction Set Computer (RISC) and Application-Specific Hardware Accelerator (*ASHA*) based cores are used to process application kernels shown in Fig. 12. The RISC processor core executes the applications having general purpose instructions and irregular access patterns. The processor core provides programmability and high-efficiency by using dual-issue superscalar, out-of-order and dynamic length pipeline stages.

The *ASHA* is generated by ROCCC [34]. ROCCC is a tool that creates streaming hardware accelerators from a subset of C. The hardware generated by ROCCC speeds up the applications by replacing critical regions in software with a dedicated hardware component. These dedicated accelerators have a low footprint and low power consumption and provide high performance. The applications having dynamic data-structures are executed on the RISC processor, and the application kernels with static data structures are executed on *ASHA* cores.

4.2. ARM Cortex-A9 Evaluation System

To test the PAMS in a real-time prototyping environment, an *ARM Cortex-A9 Evaluation System* is used. The *ARM Cortex-A9 Evaluation System* uses a dual-core ARM Cortex-A9 processor with Xilinx 7-series field programmable gate array (FPGA) logic. An ARM processor is a 32-bit hard core processor [12] supports ARMv7-A architecture. The processor has configurable L1 caches, 128-bit SIMD (Single Instruction, Multiple Data) architecture and floating point extensions. The *ARM Cortex-A9 Evaluation System* architecture combines the features of a Streaming Multi-Processor SMP and reconfigurable architecture. To efficiently utilize the technology advancements and chip resource capabilities, the *ARM Cortex-A9 Evaluation System* platform provides Data, Instruction, and Task-level parallelism. The Vivado Design Suite is used to design the system architecture. A state-of-the-art, high-performance, low-power, 28 nm ZYBO board with ZYNQ 7000 series FPGA family is used to test the system. The evaluation is further categorized into two subsections the *ARM based Generic Memory System* and the *ARM based Pattern Aware Memory System*.

4.2.1. ARM based Generic Memory System

The *ARM based Generic Memory System* (ARM-GMS) architecture (shown in Fig. 13(a)) integrates a dual-core ARM Cortex-A9 processor operating at 650 MHz clock in the processing system. Each ARM processor has its own Single Instruction Multiple Data (SIMD) media processing engines, memory management unit (MMU), and separate 32 kB level-one (L1) instruction and data caches. Each Cortex-A9 processor provides two 64-bit AXI master interfaces for independent instruction and data transactions to the Snoop Control Unit (SCU). The *Memory System* allocates 200 kB *Scratchpad* and 20 kB *Program* memories by using fast FPGA block RAM, 512 MB of DDR3 *Main Memory* with 1 Gbps bandwidth.

To manage on-chip data for computationally intensive window (loop) operations, the *ASHA* uses smart buffers in *ARM-GMS*. The smart buffer helps to minimize the accesses to the *Main Memory* bandwidth for programs that operate on static data structures and to perform loop operations over arrays. The smart buffer is a part of *ASHA* and uses FPGA resources. These smart buffers reuse data through loop transformation-based program restructuring. It stores the input and processed data for future iterations and removes the old data if it is not required in the future.

a

Application Kernel	Description	Access Pattern	GFLOPS
Radian Converter	Converts Degree into radian	Load/Store	0.375
Thresholding	An application of image segmentation, which takes streaming 8-bit pixel data and generates binary output.		0.125
FIR Finite Impulse Response	Calculates the weighted sum of the current and past inputs.	Streaming	3.875
FFT Fast Fourier Transform	Used for transferring a time-domain signal into corresponding frequency-domain signal.	1D Block	6.0
Matrix Multiplication	Output= Row[Vector] × Column[Vector] X=Y×Z	Column & Vector Access	7.750
Smith Waterman	Determining optimal local alignments between nucleotide or protein sequences	Diagonal Access	1.125
Laplacian Solver	Applies discrete convolution filter that can approximate the second order derivatives.	2D Tiled	2.125
3D-Stencil Kernel	An algorithm that averages nearest neighbor points (size 8x9x8) in 3D.	3D Stencil	4.625

b

Kernel	Description	Access Pattern	Size Byte
CRG	A compression algorithm, hides zero in a descriptor block	Pointer (Valid Element, Zero Element, Padded Element)	6822
Huffman	Huffman is an entropy coding technique. Allocate codes to symbols, using frequency of occurrence for each symbol.	Binary Tree	12350
In_Rem	A Linked List Buffer	Linked List (Address, Data, Pointer)	32280
N-Body	The 3D-Hermite algorithm used to compute movement of bodies using the newtonian gravitational force.	Octree (NW, NO, SW, SO)	16336

Fig. 12. Application Kernels: (a) Static data-structures (b) Dynamic data strictures.

4.2.2. ARM based Pattern Aware Memory System

The *ARM based Pattern Aware Memory System* (ARM-PAMS) architecture is shown in Fig. 13(b). To balance the workload, each *ASHA* is equipped with two read/write scratchpad data memories. The PAMS processing system uses dual-core ARM processors having features similar to the *ARM-GMS*. The major difference between *ARM-PAMS* and *ARM-GMS* is that the *ARM-PAMS* operates processor cores as slaves. At run-time, the *ARM-PAMS* receives address along with control instruction from the processing system and performs data operations. In current evaluation, the L2 memory of the Cortex-A9 processor is replaced with the *Specialized Scratchpad Memory*. The *Specialized Scratchpad Memory* manages the static and dynamic descriptors and data manager that organize, reuse and transfer data to processing cores.

4.3. Tasksim simulator

A trace-driven Tasksim simulator [50] for application specific accelerator-based multi-core architectures is used to test the PAMS. The TaskSim is designed for computer architecture exploration having complex multi-/many-core chip multiprocessors. The Tasksim targets the simulation of parallel applications coded in a master–worker task offload computational model. The Tasksim architecture is configured to model a single-chip Cell system with single master processor and 12 accelerators/workers (shown in Fig. 14). The on-chip cache is disabled, and configures the interconnection and memory bandwidth. The master processors start applications at the main() subroutine of the program. The memory controller is used to transfer data between main and local memories. The system architecture uses two memory controllers to read/write data from main memory. The Tasksim based systems are categorized into *Tasksim-PAMS* and *Tasksim Generic Memory System* (Tasksim-GMS). The *Tasksim-GMS* uses a DMA for each worker which overlaps the data transfer and computation. The DMA transfers the data between local memory and the main memory using two Memory Controllers. Whereas, the *Tasksim-PAMS* does not use DMA controller or Memory Controller. It takes the data transfer traces, organizes them into descriptor memory and performs main memory data transfer by using two *Pattern Aware Main Memory Controller*s.

5. Results and discussion

This section analyzes the results of different experiments conducted on real prototype and simulation environments. The experiments are characterized into four subsections: *System Performance*, *Vector Processing*, *Program Memory*, *Memory Throughput*, and *Dynamic Power and Energy*.

5.1. Program memory

The program memory allocates memory space for the retrieval and the manipulation of memory assignments. In this section we compare the executable object files of the *ARM-PAMS* and *ARM-GMS* for each kernel using GNU GCC compiler.

Fig. 15 presents the program memory for static and dynamic data-structures generated by the *ARM-PAMS* and *ARM-GMS* respectively. The *y*-axis is in logarithmic scale, smaller is better. In the *ARM-PAMS*, the *Regular Descriptor Memory* holds the local variables and the descriptor information for static access patterns.

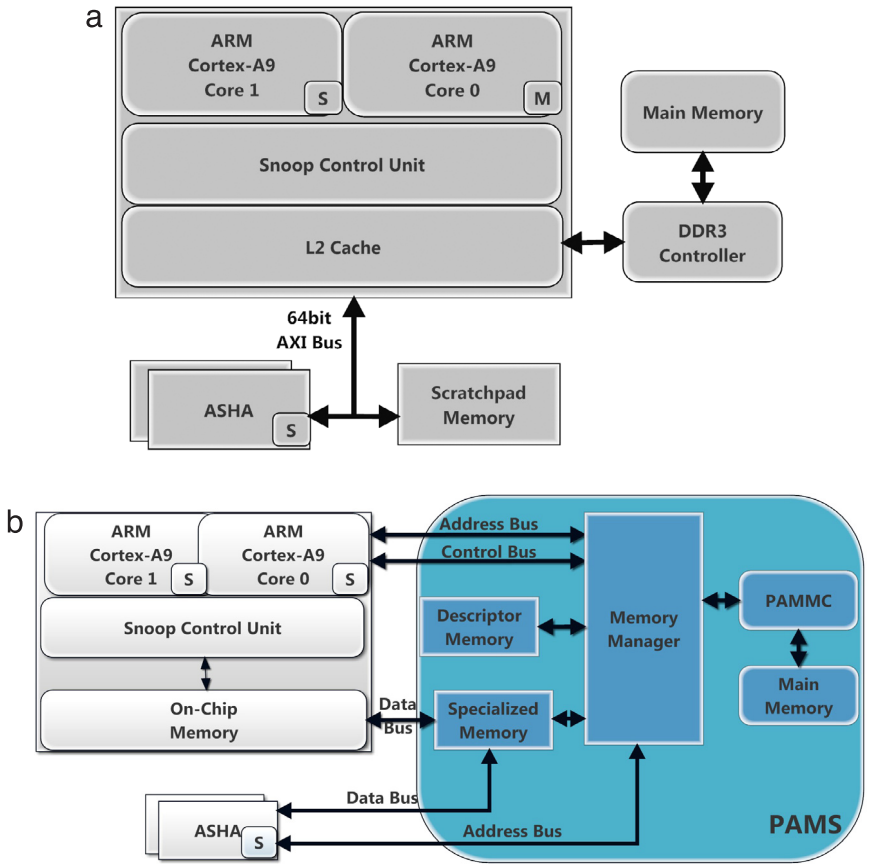

Fig. 13. Architecture: (a) Baseline memory system (b) PAMS based system.

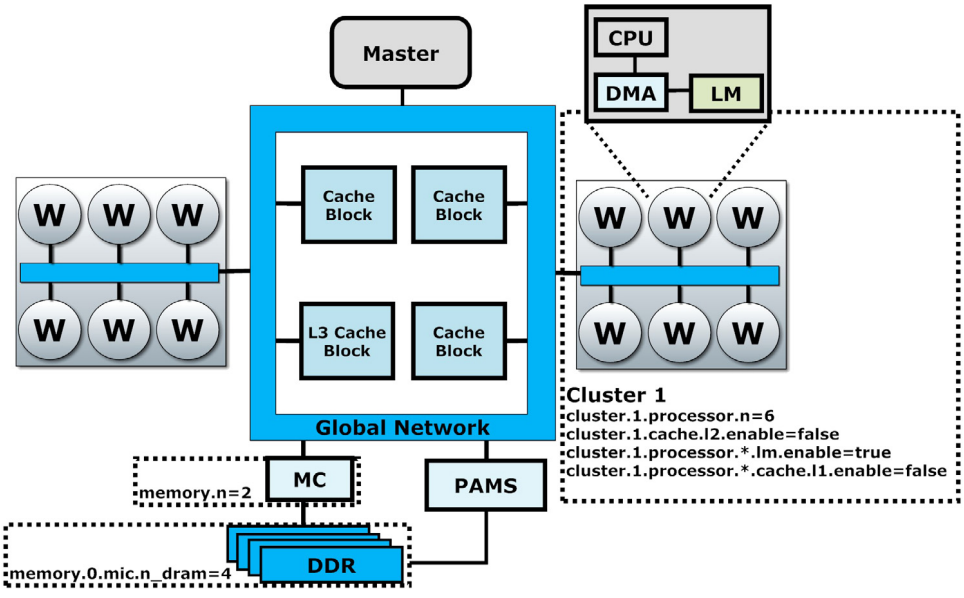

Fig. 14. Tasksim: PAMS and Memory System Architecture.

In the *ARM-GMS* implementation, the equivalent information is stored in the *stack* and *heap*. The Rad_Con and Thresh applications have statically known Load/Store accesses, which occupy 3.1 times less space in the *Regular Descriptor Memory* compared to the *ARM-GMS*. The FIR application has a streaming data pattern, which uses 3.25 times less memory. The FFT and Mat_Mul/Lapl/Smith-W kernels access 1D and 2D patterns respectively. These applications use 2.56, 2.47, 3.97, and 4.85 times less space in the *Regular Descriptor Memory* respectively. The 3D-Sten kernel accesses 3D data patterns having $32 \times 32 \times 32$ dimensions. Due to the small dimensions of the 3D-stencil data set, the *ARM-PAMS* uses two times more regular memory than the *ARM-GMS*. However across results show as a rule of thumb, the *ARM-PAMS* can handle applications having complex and dense access patterns more efficiently as their dimension size increases. We measured the heap memory usage of the *ARM-GMS* for applications having

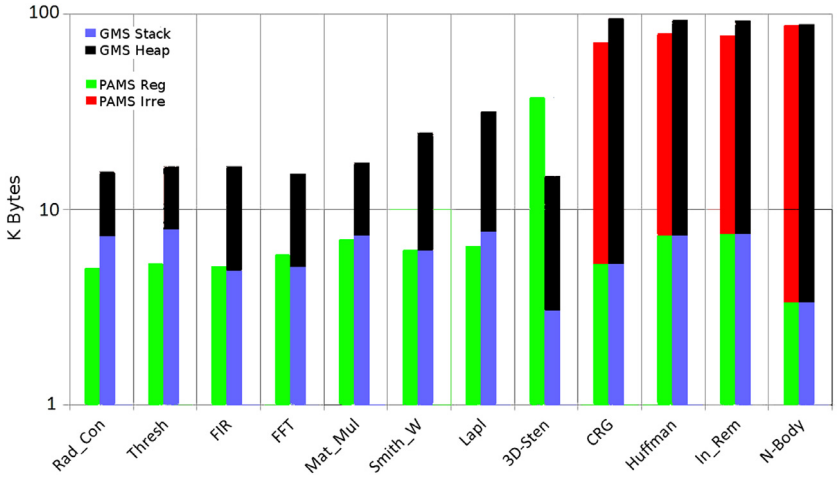

Fig. 15. Program memory: Static and dynamic data-structures.

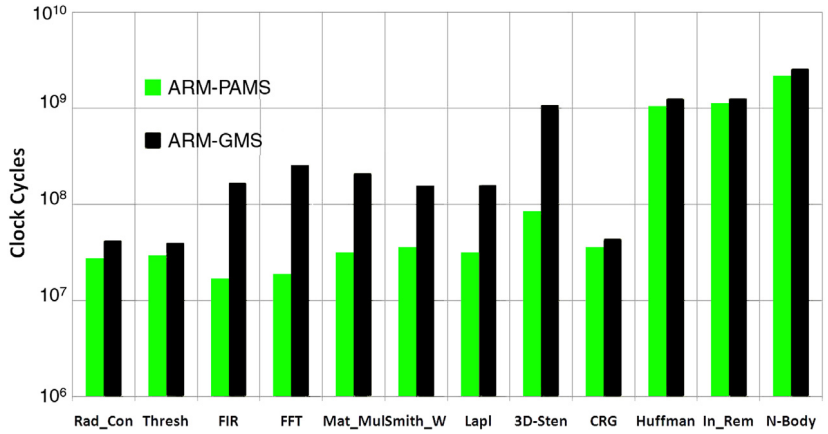

Fig. 16. Application performance.

dynamic data-structures and compared it with the *ARM-PAMS Irregular Descriptor Memory*. For CRG, Huffman, In_Rem, and N-Body, the *ARM-PAMS* requires 1.01× to 1.3× less *Irregular Descriptor Memory*, compared to the *ARM-GMS*. The *stack* and *Regular Descriptor Memory* size are the same in both systems.

5.2. System performance

In this section we compare the performance of *ARM-PAMS* and *ARM-GMS*. Fig. 16 shows the number of clock cycles taken by an application (see Section 4.1) to manage and process the 4 MB data set. *Y*-axis presents the number of clocks in logarithmic scale (smaller is better). *X*-axis shows applications. The *ARM-GMS* uses an ARM processor, DMA and Memory Controller to transfer data between the *Main Memory* and *Local Memory*. ARM processes applications with dynamic data-structures. The processor uses load/store instructions that read/write data from the *Main Memory* to cache it. The cache manages local data and reuses it where required. The applications with static data transfers are executed on *ASHA*s. The *ASHA* takes data from scratchpad memory and uses smart buffers that analyze the array accesses and tries to reuse the data between loop iterations. These reuse patterns can be exploited and reduce the number of off-chip memory accesses. The *ARM-PAMS* uses *Specialized Scratchpad Memory* to feed data to the ARM processor and *ASHA* cores. The *ARM-PAMS Memory Manager* manages *Specialized Scratchpad Memory* data and update/reuse data using Register File. For each data miss, the *Memory Manager* transfers the descriptor request to *PAMMC* that moves the missing pattern of data.

By using the *ARM-PAMS*, the results show the *Rad_Con* and *Thresh* applications achieve 1.5× and 1.3× of speedup respectively over the *ARM-PAMS*. These application kernels have several compile-time predictable memory access requests with no data locality. The *FIR* application has a streaming data access pattern and achieves 10× of speedup. The *FFT* application kernel reads a 1D block of data, processes it and writes it back to main memory. This application achieves 13.30× of speedup. The *ARM-PAMS* uses single descriptor to transfer data for *FIR* and *FFT* applications. The *Mat_Mul* kernel accesses row and column vectors. The application attains 6.71× of speedup. The *ARM-PAMS* uses two descriptors to transfer row and column vectors. The *ARM-PAMS* manages addresses of row and column vectors in hardware. The *Smith_W* and *Lapl* applications take 2D blocks of data and achieve 4.34× and 5× of speedup respectively. The *3D-Stencil* data decomposition achieves 12.83× of speedup. The *ARM-PAMS* takes 2D and 3D block descriptors and manages them in hardware. The compile-time predictable access patterns are placed on the *Descriptor Memory* at program time and are programmed in such a way that few operations are required for generating addresses at run-time. The *ARM-GMS* uses multiple load/store, or DMA calls to access complex patterns. The CRG, Huffman, In Rem and N-Body applications process dynamic data-structures; therefore, the ARM core is used to process these applications. The CRG and Huffman applications have unpredictable memory access patterns with long strides and no data locality. While executing these applications

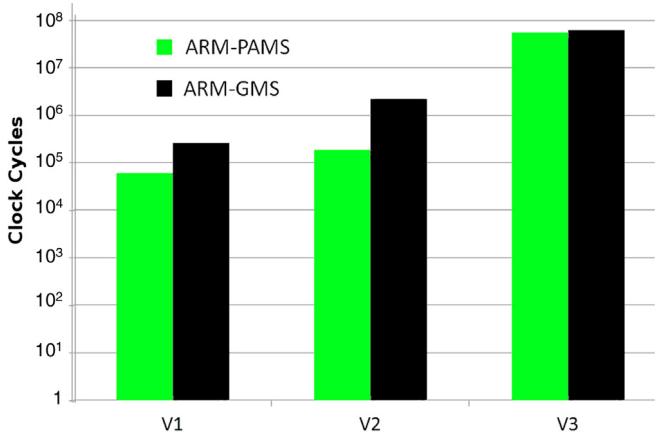

Fig. 17. ARM-PAMS and ARM-GMS: Vector system performance.

on the *ARM-PAMS*, the system achieves 1.23×, 1.20× of speedup respectively over the *ARM-GMS*. The In_Rem application has run-time predictable memory access patterns with no data locality, hence achieves 1.02× of speedup. The N-Body application includes predictable data patterns with data locality. While running on the *ARM-PAMS*, it achieves 1.19× of speedup over the *ARM-GMS*.

The *ARM-GMS* uses an ARM processor core that transfers data between *Cache/Scratchpad* memories and the *Main Memory* using load/store or DMA instructions. The AMR processor controls the applications data requests and performs on-chip data management. The *ARM-PAMS* manages *Local Memory* data and the *Main Memory* data transfers in patterns without the support of separate microprocessor core. The *ARM-PAMS* accesses data in patterns which improves the memory bandwidth by transferring descriptors to the memory controllers, rather than individual references and by accessing data from the DRAM device. The *ARM-PAMS Specialized Scratchpad Memory* accesses the whole data pattern block as a cache line and temporarily holds data to speed up later accesses.

5.3. Vector processing

We have executed vector multiplication applications on NEON coprocessor of the *ARM Cortex-A9 Evaluation Systems*. NEON coprocessor uses SIMD instruction sets to process vector multiplication. *ARM Cortex-A9 Evaluation System* architecture uses dual NEON coprocessor having 512 bytes of register file, 64-bit register size and performs single precision floating point instructions in all lanes.

Three types of vector data structures (V1, V2, and V3) are selected to test the performance of system architecture. The V1 vector has fixed strides between two elements, and its memory accesses are predictable at compile time. The V2 memory accesses are not predictable at compile but known at run-time, the distance between two consecutive elements of V2 is greater than the cache line size. The V3 memory accesses are not known at run-time, we used a random address generator that provides address before computation.

Fig. 17 shows the number of clock cycles taken *ARM-PAMS*, and *ARM-GMS* to access and processes V1, V2 and V3 vectors. *Y*-axis presents the number of clocks in logarithmic scale. The *x*-axis shows the applications with V1, V2 and V3 vector data access patterns. While perforating vector multiplication on V1 type vector, the *ARM-PAMS* achieves 4.12× of speedups against the *ARM-GMS*. The V1 addresses are aligned; therefore, the ARM system uses multiple direct memory access calls which require the start address, the end address and the size of the transfer. The *ARM-PAMS* organizes the known address in descriptors which reduces the run-time address generation and address management time. The ARM snoop control unit (SCU) reuses and pre-fetches the data in parallel with the computation, but there is still on-chip bus management and *Main Memory* data transfer issues. For vector V2 addition, the *ARM-PAMS* achieves 12.68× of speedup. The V2 has run-time predictable stride size, having a size greater than the cache line. The baseline system uses multiple transfer calls to access the vector, which generates address generation, on-chip bus arbitration and the *Main Memory* bank section time. The *ARM-GMS* uses multiple load/store and DMA transfer call to access V2 vector; this adds on-chip bus delay. The *ARM-PAMS* efficiently handles strides at run-time and manages address/data of access patterns, translates/recorders in hardware, in parallel with ARM processor cores. As the vector addresses are in the form of descriptors the PAMS on-chip bus and the *PAMMC* manage data transfers in a single or few bursts. The *SSM* places complete data structure in continuous format and feeds it to processing cores that reduce the local memory management time. The *SSM* banks read/write operations are performed in parallel. The vector addition of type V3 vectors gives 1.21× of speedup. The V3 vector requires pointer/irregular data transfer calls, which generates address management, bus arbitration, scheduling and the *Main Memory* delays. As the address is not known, the *ARM-PAMS Memory Manager* and *PAMMC* are not able to work in parallel. The Baseline System Architecture uses generic *Local Memory*, *Bus Management* and the *Main Memory* units. This results in a settlement on the possible performance because of the generic units that require extra handshaking and synchronization time. In order to achieve performance *ARM-PAMS* bypasses the generic system units and introduces specialized *Local Memory*, *Memory Management* and *Main Memory*. The *PAMS* internal units have the ability to operate independently, in parallel with each other and *ARM-PAMS* achieves maximum performance when all of its units work in parallel.

5.4. Memory performance

In this section, we measure the Memory performance of *Tasksim-PAMS* and *Tasksim-GMS* using Tasksim Simulator by reading and writing two types of data patterns. The memory performance includes local memory, main memory load store throughput, bus system throughput and the DRAM read/write throughput. The system architecture uses different number of *Worker* cores. The *X*-axis (shown in Fig. 18) presents two types of data transfers and the number of cores. *Y*-axis presents achieved memory bandwidth in MBytes per seconds (higher is better). Each data transfer reads and writes a data set of 8 MB from/to the main memory. The type *Short Window* contains data transfers that have a maximum transfer size of 128 B and the type *Long Window* has a transfer size of 4 kB. The data transfers have an unpredictable jump between two consecutive transfers. These access patterns are selected to check the performance of local memory, bus arbitrations and main memory read/write throughput. While using 1, 2, 4, 8 and 12 requesting *Workers* for *Short Transfer* type, results show that the PAMS transfers data 3.56×, 4.29×, 4.34×, 4.69× and 5.12 times faster respectively than the *Tasksim-GMS*. The *Short Window* data transfer has multiple transfers calls which generate bus switching and the main memory bank read write delays. While transferring data with the *Long Window* type, the PAMS improves throughput 3.53×, 3.41×, 3.24×, 3.13 and 3.30× times. Results show that the PAMS improves the aggregated throughput for *Short Windows* when increasing the number of cores. The *Tasksim-GMS* uses the DMA controller that forces to follow the bus protocol and requires a processor that provides data transfer instructions. The DMA responds as a slave when its registers are being read or written and acts as a bus master once it initiates data transactions. For multiple cores, the *Tasksim-GMS* uses multiple instructions to

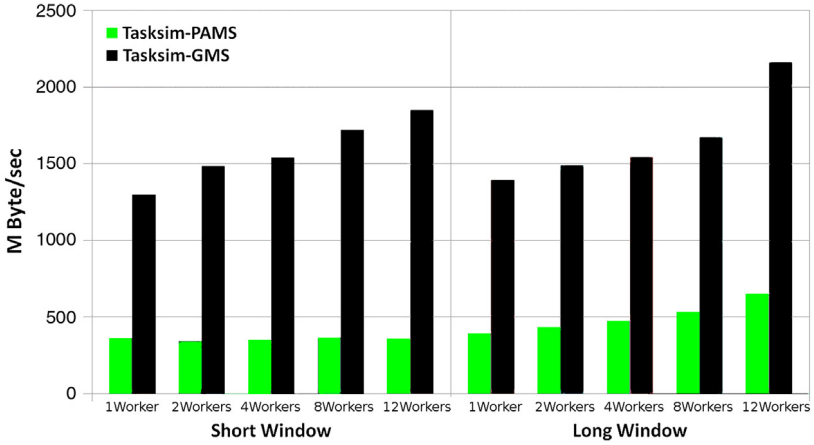

Fig. 18. Tasksim-GMS and Tasksim-PAMS: Memory performance.

Table 1
ARM based Generic Memory Systems: Dynamic Power and Energy consumption for FIR application.

	Dynamic Power (W)	Energy (J)
ARM-PAMS	2.52	14.63
ARM-GMS	2.64	23.92

initialize DMA. DMA 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 DRAM bank/row management. The *Short Window* type uses few descriptors that reduce run-time address generation and address request/grant delay and improve the throughput by managing addresses at compile-time and by accessing data from multi-banks in parallel.

5.5. Dynamic Power and Energy consumption

The *Dynamic Power* and *Energy* of the *ARM-PAMS* and the *ARM-GMS* were measured (shown in Table 1), while executing the applications (discussed in Section 4.1) having 4 MB of input data set. To measure voltage and current the *ARM Cortex-A9 Evaluation System* uses a resistor and ADC to sense current/voltage. The *ARM-GMS* is using an ASIC based memory controller designed to consume less power. For real prototyping we implemented the *ARM-PAMS* on FPGA (on ASIC the power consumption will be lesser).

While comparing, the *Dynamic Power* results show that the *ARM-PAMS* draws 4.6% less dynamic power than the *ARM-GMS*. Whereas while comparing the *Energy*, the *ARM-PAMS* consumes 1.63 times less *Energy* than the *ARM-GMS*. The reason of consuming less *Dynamic Power* and improvement in the *Energy* of *ARM-PAMS* are: The PAMS allows processing system to execute application efficiently by reordering multiple data transfers in pattern and schedules them efficiently; this reduces the bus switching and arbitration which improves the application performance and reduces the power consumption.

The DRAMs of the Main Memory system consume a significant amount of power while refreshing, charging and enabling banks, row, and column. The *ARM-PAMS* PAMMC manages and controls DRAMs banks and rows, opens single or multiple banks and rows if an access pattern demands. At run-time the *ARM-PAMS* PAMMC organizes data transfers in patterns, by using these patterns the PAMS reuses and manages Local Memory data, reduces on/off-chip bus flow control, arbitration, translation, interconnection delay and manage DRAM banks, columns, and rows. Once the data transfer is organized in pattern descriptor, the PAMS reuses data on the local memory, avoids extra bus switching and manages the main memory banks that reduce the dynamic power.

6. 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.

6.1. Scratchpad

Scratchpad is a low latency memory that is tightly coupled to the CPU [57]. Therefore, it is a popular choice for on-chip storage in real-time HPC systems. The allocation of code/data to scratchpad memory is performed at compile time leading to predictable memory access latencies. Panda et al. [46] 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. [59] aim at optimizing memory access tasks worst-case performance. However, in that study, scratchpad 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 is 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 [13]. 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 [13]. 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. [60] 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. PAMS address manager arranges unknown memory access at run-time in the form of pattern descriptors. PAMS performs data management and handles complex memory accesses at run-time using 1D/2D/3D *Scratchpad Memory*.

6.2. Cache

G. Stitt et al. [58] 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. [10] extended the idea of traversal data caches [58] 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 scratchpad memory and data access in hardware allows PAMS to manage pointer-based data structures without a microprocessor core. This PAMS reduced memory address space by transforming large and irregular address spaces into a few descriptor blocks and mapping them to the cache framework. Mellor-Crummey et al. [45] 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. [14] 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). PAMS on-chip specialized scratchpad memory unit maps and allocates different types of applications with irregular memory patterns. PAMS 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 [67], 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 [9] vector architecture with a scratchpad to read and write data and a crossbar network to shuffle vector operations. VENICE [54] 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 [66] 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. PAMS eliminates the crossbar network and the limitation of using a scalar processor for data transfer. PAMS manages addresses in hardware with the pattern descriptors and accesses data from the *Main Memory* without support of a scalar processor core. The PAMS data manager rearranges on-chip data using the buffer memory without a complex crossbar network, which allows the vector processor to operate at higher clock rates.

6.3. Memory Manager

The snoop control unit (SCU) manages 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 cores through a *Local Memory System* coherency bus and manages the coherency between the L1 and the L2 cache. 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. [53] 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 PAMS system overcomes this problem by scheduling task operations while taking into account the programmed priorities, and on-chip specialized scratchpad 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-Advanced High-performance Bus (ML-AHB) bus-matrix proposed by ARM [56] has been used in many SoC designs due to its simplicity, simple architecture and low power. The bus-matrix interconnection scheme provides parallel access paths between multiple masters and slaves in a system. The Processor Local Bus (PLB) crossbar switch (CBS) from IBM [32] allows communication between masters on one PLB and slaves on the other. The CBS supports concurrent data transfers on multiple buses along with a prioritization method to hold multiple requests to a generic slave port. Like other on-chip Bus Units, ML-AHB and PLB (CBS) use a master core that manages on-chip bus transactions.

Marchand et al. [43] have developed software and hardware implementations of the Priority Ceiling Protocol that control the multiple-unit resources in a uniprocessor environment. Yan et al. [65] have 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 [35,8] 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 begins 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 not only tends to reduce both latency, but also optimizes memory

bandwidth usage: if the prediction is correct, valuable memory traffic is prioritized, and unimportant one can be avoided. Thread level speculation [23] 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 PAMS holds information of memory patterns in the form of *Descriptor Memory*. Currently, accessed patterns are placed in the address manager of PAMS. The PAMS monitors the access patterns without using a separate core and reuses these patterns for multiple cores if required.

Several DRAM access scheduling techniques [51,55] 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. [37] 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. [4] 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 the *Main Memory* system. APEX [24] 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 PAMS *Pattern Descriptor Unit* not only determines access patterns at compile-time but also manages at run-time complex patterns which are difficult to predict.

6.4. Prefetching

Porterfield et al. [5,48] present 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 [5] 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. [38] extended 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. [19,20] 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. [15] 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. PAMS manages software prefetching by using known *Descriptor Memory*. The PAMS *Descriptor Memory* manages memory accesses at compile-time and prefetches them at run-time before computation. The PAMS 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 has 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 has overhead to initialize hardware block at run-time. Porterfield [48] 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. [40] 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 [2] 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. [33] 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. [21] 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. [52] 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 [17,36] 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 PAMS *Memory Manager* dynamically initializes the *Descriptor Memory* for known and unknown memory access patterns. PAMS also schedules the sparse vector, linked list, tree, etc. based access patterns at compile or runtime and prefetches them dynamically.

6.5. Scatter Gather controllers

The XPS Channelized DMA Controller [16] 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 [39] and ALTERA Scatter/Gather DMA Controller core [1] 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. PAMS 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.

Wen et al. [63] explain FT64 and Multi-FT64 based system for High Performance Computing with streams. FT64 is a programmable 64 bit 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. [64] present an FT64 based on chip memory sub system that combines software/hardware managed memory structure. Chai et al. [7] 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 aligns 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. C. Gou et al. [22] 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. PAMS 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 PAMS uses *Specialized Scratchpad* memory which handles complex data transfers and feeds them to processing cores.

McKee et al. [44] 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 [6,68] that supports application-specific optimizations through configurable physical address remapping. The Impulse memory controller [68] 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. PAMS remaps and produces physical addresses in the hardware unit without the overhead of operating system intervention. Based on its C/C++ language support, PAMS can be used with any operating system that supports the C/C++ stack.

Corbal et al. [11] 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. Wang et al. [62] present a novel mathematical computing architecture called MaPU for data-intensive computing with power efficiency and sustained computation throughput. The MaPU architecture applies multi-granularity parallel (MGP) memory system with fundamental shuffle and reorder ability, pipelined with wide SIMD data paths. The memory system supports row- and column-major accesses for matrices with common data types and uses a low-level state-machine-based programming model, therefore; extra efforts are needed in programming. PAMS handles complex, dense and irregular memory patterns and gives support to SISD, SIMD and MIMD architectures. PAMS improves on-chip communication bandwidth by managing both compile- and run-time generated access pattern descriptors. The PAMS descriptor block uses offset parameter that aligns complex transfers, each transfer with variable stride. The *Pattern Aware Main Memory Controller* manages SDRAM by using a bank management policy selected by run-time access patterns. Based on the pattern descriptors the PAMMC performs memory shuffling and reordering without affecting the fairness and quality of service.

7. Conclusion

Multi-core HPC systems suffer from poor performance due to the memory system. In this work, we had studied some existing and future memory technologies and proposed an efficient and intelligent memory system in hardware called Pattern Aware Memory System (*PAMS*). The proposed memory architecture supports both *Static* and *Dynamic Data-structures* for the heterogeneous multi-core system using memory pattern descriptors that reduce the impact of memory latency. *Static* and *Dynamic Data-structures* of multi-cores are described using a separate *Descriptor Memory*, which reduces the on-chip communication time and run-time address generation overhead. The PAMS 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 noncontiguous memory accesses simultaneously which reduce read/write delay due to the control selection of DRAM 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 concerning the access pattern. The PAMS is easy to integrate in real-prototype and trace based simulation environments. The results on real-prototype environment show that PAMS achieves a maximum speedup of 12.83× for applications. The PAMS consumes 4.6% and 1.6 times less dynamic power and energy respectively. The simulator environment results show that the PAMS transfers data-structures up to 5.12× faster than the baseline system.

Acknowledgment

This work has been supported by the Unal Color of Education Research and Development (Private) Limited Islamabad (Grant Number: (UC2016-0111)).

References

[1] Altera Corporation. Scatter-Gather DMA Controller Core, Quartus II 9.1, November 2009.
[2] Jean-Loup Baer, Tien-Fu Chen, An effective on-chip preloading scheme to reduce data access penalty, in: Proceedings of the ACM/IEEE Conference, 1991, pp. 176–186.
[3] Rajeshwari Banakar, Stefan Steinke, Bo-Sik Lee, Mahesh Balakrishnan, Peter Marwedel, Scratchpad memory: design alternative for cache on-chip memory in embedded systems, in: Proceedings of the Tenth International Symposium on Hardware/software Codesign, ACM, 2002, pp. 73–78.
[4] Mahdi Nazm Bojnordi, Engin Ipek, Pardis: A programmable memory controller for the ddrx interfacing standards, in: 39th Annual International Symposium on Computer Architecture (ISCA), 2012, IEEE, 2012, pp. 13–24.
[5] David Callahan, Ken Kennedy, Allan Porterfield, Software prefetching, in: ACM SIGARCH Computer Architecture News, Vol. 19, ACM, 1991, pp. 40–52.
[6] John Carter, Wilson Hsieh, Leigh Stoller, Mark Swanson, Lixin Zhang, Erik Brunvand, Al Davis, Chen-Chi Kuo, Ravindra Kuramkote, Michael Parker, Lambert Schaelicke, Terry Tateyama, Impulse: Building a Smarter Memory Controller, Prentice-Hall, Inc., 1999.
[7] Sek M. Chai, N. Bellas, M. Dwyer, D. Linzmeier, Stream memory subsystem in reconfigurable platforms, in: 2nd Workshop on Architecture Research using FPGA Platforms, 2006.
[8] Shailender Chaudhry, Robert Cypher, Magnus Ekman, Martin Karlsson, Anders Landin, Sherman Yip, Håkan Zeffer, Marc Tremblay, Simultaneous speculative threading: a novel pipeline architecture implemented in sun's rock processor, in: ACM SIGARCH Computer Architecture News, Vol. 37, ACM, 2009, pp. 484–495.
[9] Christopher H. Chou, Aaron Severance, Alex D. Brant, Zhiduo Liu, Saurabh Sant, Guy G.F. Lemieux, Vegas: soft vector processor with scratchpad memory, in: Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays, ACM, 2011, pp. 15–24.
[10] James Coole, John Wernsing, Greg Stitt, A traversal cache framework for fpga acceleration of pointer data structures: A case study on barnes-hut n-body simulation, in: 2009 International Conference on Reconfigurable Computing and FPGAs, IEEE, 2009, pp. 143–148.
[11] Jesus Corbal, Roger Espasa, Mateo Valero, Command vector memory systems: High performance at low cost, in: International Conference on Parallel Architectures and Compilation Techniques, 1998. Proceedings, IEEE, 1998, pp. 68–77.
[12] Louise H. Crockett, Ross A. Elliot, Martin A. Enderwitz, Robert W. Stewart, The zynq book: Embedded processing with the arm cortex-a9 on the xilinx zynq-7000 all programmable soc. Strathclyde Academic Media, 2014.
[13] J.-F. Deverge, Isabelle Puaut, Wcet-directed dynamic scratchpad memory allocation of data, in: 19th Euromicro Conference on Real-Time Systems, 2007. ECRTS'07.
[14] Pedro C. Diniz, Joonseok Park, Data search and reorganization using fpgas: application to spatial pointer-based data structures, in: IEEE, 11th Annual IEEE Symposium on Field-Programmable Custom Computing Machines, 2003, pp. 207–217.
[15] Gautam Doshi, Rakesh Krishnaiyer, Kalyan Muthukumar, Optimizing software data prefetches with rotating registers, in: International Conference on Parallel Architectures and Compilation Techniques, IEEE, 2001, pp. 257–267.
[16] Embedded System Tools Reference Manual EDK 13.1.
[17] Keith I. Farkas, Norman P. Jouppi, Paul Chow, How useful are non-blocking loads, stream buffers and speculative execution in multiple issue processors? in: Proceedings., First IEEE Symposium on High-Performance Computer Architecture, IEEE, 1995, pp. 78–89.
[18] Wu-chun Feng, Kirk W. Cameron, The green500 list: Encouraging Sustainable Supercomputing. Vol. 40, IEEE, 2007, pp. 50–55.
[19] E.H. Gornish, Compile time analysis for data prefetching (Master's thesis), University of Illinois at Urbana-Champaign, 1989.
[20] Edward H. Gornish, Elana D. Granston, Alexander V. Veidenbaum, Compiler-directed data prefetching in multiprocessors with memory hierarchies, in: ACM International Conference on Supercomputing, 25th Anniversary Volume, ACM, 2014, pp. 128–142.
[21] Edward H. Gornish, Alexander Veidenbaum, An integrated hardware/software data prefetching scheme for shared-memory multiprocessors, in: International Conference on Parallel Processing, ICPP Vol. 2, IEEE, 1994, pp. 281–284.
[22] Chunyang Gou, Kuzmanov Georgi, Georgi N. Gaydadjiev, SAMS multi-layout memory: providing multiple views of data to boost SIMD performance, in: ICS'10, New York, NY, USA, 2010. Proceedings of the 24th ACM International Conference on Supercomputing.
[23] Steffan J. Greggory, Christopher B. Colohan, Zhai Antonia, Todd C. Mowry, A scalable approach to thread-level speculation, in: ACM SIGARCH Computer Architecture News, Vol. 28, 2000.
[24] Peter Grun, Nikil Dutt, Alex Nicolau, Apex: access pattern based memory architecture exploration, in: Proceedings of the 14th International Symposium on Systems Synthesis, ACM, 2001, pp. 25–32.
[25] Matthias Hartmann, Praveen Raghavan, Liesbet Van Der Perre, Pulin Agrawal, Wim Dehaene, Memristor-based (reram) data memory architecture in asip design, in: Conference on Euromicro, Digital System Design, DSD, 2013.
[26] Tassadaq Hussain, A novel access pattern-based multi-core memory architecture (Ph.d. thesis), Universitat Politècnica de Catalunya, 2014.
[27] Tassadaq Hussain, Amna Haider, Eduard Ayguade, PMSS: A programmable memory system and scheduler for complex memory patterns, Sci. Direct J. Parallel Distrib. Comput. (2014).
[28] Tassadaq Hussain, Oscar Palomar, Adrian Cristal, Eduard Ayguade, Amna Haider, Access pattern based multi-layer bus virtualization controller, in: The 13th International Bhurban Conference on Applied Sciences & Technology, IBCAST, 2016.
[29] Tassadaq Hussain, Oscar Palomar, Adrian Cristal, Osman Unsal, Eduard Ayguady, Memory controller for vector processor, J. Signal Process. Syst. (2016).
[30] Tassadaq Hussain, Oscar Palomar, Adrian Cristal, Osman Unsal, Eduard Ayguady, Mateo Valero, Advanced pattern based memory controller for FPGA based HPC applications, in: International Conference on High Performance Computing & Simulation, ACM, IEEE, 2014, p. 8.
[31] Tassadaq Hussain, Miquel Pericas, Nacho Navarro, Eduard Ayguade, PPMC: Hardware scheduling and memory management support for multi hardware accelerators, in: 22nd International Conference on Field Programmable Logic and Applications, (FPL), IEEE, 2012, pp. 571–574.
[32] IBM CoreConnect. PLB Crossbar Arbiter Core. 2001.
[33] Ganusov Ilya, 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.
[34] Villarreal Jason, Park Adrian, Najjar Walid, 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.
[35] Lu Jiwei, Das Abhinav, Hsu Wei-Chung, Nguyen Khoa, Santosh G. Abraham, Dynamic helper threaded prefetching on the Sun UltraSPARC® CMP processor, in: Proceedings 38th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO-38, IEEE, 2005, p. 12.
[36] Norman P. Jouppi, Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers, in: 17th Annual International Symposium on Computer Architecture, 1990, IEEE, 1990, pp. 364–373.
[37] Yoongu Kim, Dongsu Han, Onur Mutlu, Mor Harchol-Balter, Atlas: A scalable and high-performance scheduling algorithm for multiple memory controllers, in: HPCA-16 2010 The Sixteenth International Symposium on High-Performance Computer Architecture, IEEE, 2010, pp. 1–12.
[38] Alexander C. Klaiber, Henry M. Levy, An architecture for software-controlled data prefetching, in: ACM SIGARCH Computer Architecture News, Vol. 19, ACM, 1991, pp. 43–53.
[39] Lattice Semiconductor Corporation. Scatter-Gather Direct Memory Access Controller IP Core Users Guide, October 2010.
[40] R.L. Lee, The Effectiveness of Caches and Data Prefetch Buffers in Large-Scale Shared Memory Multiprocessors (Ph.D. thesis), Department of Computer Science, University of Illinois at Urbana-Champaign, 1987.
[41] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, Kim Hazelwood, Pin: building customized program analysis tools with dynamic instrumentation, in: ACM Sigplan Notices, Vol. 40, ACM, 2005, pp. 190–200.
[42] Ken Mai, Tim Paaske, Nuwan Jayasena, Ron Ho, William J. Dally, Mark Horowitz, Smart memories: A modular reconfigurable architecture, in: ACM SIGARCH Computer Architecture News, Vol. 28, ACM, 2000, pp. 161–171.
[43] P. Marchand, P. Sinha, A Hardware Accelerator for Controlling Access to Multiple-unit Resources in Safety/time-critical Systems, Inderscience Publishers, 2007.
[44] Sally A. McKee, William A. Wulf, James H. Aylor, Robert H. Klenke, Maximo H. Salinas, Sung I. Hong, Dee A.B. Weikle, Dynamic Access Ordering for Streamed Computations. Vol. 49, IEEE, 2000, pp. 1255–1271.
[45] John Mellor-Crummey, David Whalley, Ken Kennedy, Improving Memory Hierarchy Performance for Irregular Applications Using Data and Computation Reorderings. Vol. 29, Springer, 2001, pp. 217–247.
[46] Preeti Ranjan Panda, Nikil D. Dutt, Alexandru Nicolau, Memory Issues in Embedded Systems-on-chip: Optimizations and Exploration, Springer Science & Business Media, 1999.
[47] Stylianos Perissakis, Yangsung Joo, Jinhong Ahn, A. Dellon, J. Wawraynek, Embedded dram for a reconfigurable array, in: Symposium on VLSI Circuits 1999. Digest of Technical Papers.
[48] Allan Kennedy Porterfield, Software methods for improvement of cache performance on supercomputer applications. Rice University, 1989.
[49] Nikola Rajovic, Alejandro Rico, Nikola Puzovic, Chris Adeniyi-Jones, Alex Ramirez, Tibidabo: Making the case for an arm-based hpc system, Elsevier: J. Future Gener. Comput. Syst. 36 (2014) 322–334.
[50] Alejandro Rico, Felipe Cabarcas, Antonio Quesada, Milan Pavlovic, Augusto Javier Vega, Carlos Villavieja, Yoav Etsion, Alex Ramirez, Scalable simulation of decoupled accelerator architectures. Tech. Rep. UPC-DACRR-2010-14, Universitat Politecnica de Catalunya, 2010.
[51] Scott Rixner, William J. Dally, Ujval J. Kapasi, Peter Mattson, John D. Owens, Memory access scheduling, in: ACM SIGARCH Computer Architecture News, Vol. 28, ACM, 2000, pp. 128–138.
[52] Amir Roth, Andreas Moshovos, Gurindar S. Sohi, Dependence based prefetching for linked data structures, ACM SIGOPS Oper. Syst. Rev. 32 (1998) 115–126.

[53] Zhuravlev Sergey, Blagodurov Sergey, Fedorova Alexandra, Addressing shared resource contention in multicore processors via scheduling, in: ACM SIGARCH Computer Architecture News, 2010.
[54] Aaron Severance, Guy Lemieux, Venice: A compact vector processor for fpga applications, in: International Conference on Field-Programmable Technology, (FPT), IEEE, 2012, pp. 261–268.
[55] Jun Shao, Brian T. Davis, A burst scheduling access reordering mechanism, in: 2007 IEEE 13th International Symposium on High Performance Computer Architecture, IEEE, 2007, pp. 285–294.
[56] Anurag Shrivastav, G.S. Tomar, Ashutosh Kumar Singh, Performance comparison of amba bus-based system-on-chip communication protocol, in: 2011 International Conference on Communication Systems and Network Technologies, CSNT, IEEE, 2011, pp. 449–454.
[57] Stefan Steinke, Nils Grunwald, Lars Wehmeyer, Rajeshwari Banakar, M. Balakrishnan, Peter Marwedel, Reducing energy consumption by dynamic copying of instructions onto onchip memory, in: 15th International Symposium on System Synthesis, 2002.
[58] Greg Stitt, Gaurav Chaudhari, James Coole, Traversal caches: A first step towards fpga acceleration of pointer-based data structures, in: Proceedings of the 6th IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis, ACM, 2008, pp. 61–66.
[59] Vivy Suhendra, Tulika Mitra, Abhik Roychoudhury, Ting Chen, Wcet centric data allocation to scratchpad memory, in: 26th IEEE International Real-Time Systems Symposium, 2005. RTSS 2005.
[60] Udayakumaran Sumesh, Dominguez Angel, Barua Rajeev, Dynamic allocation for scratch-pad memory using compile-time decisions, ACM Trans. Embedded Comput. Syst. (TECS) (2006).
[61] Xiang Tian, Khaled Benkrid, Massively parallelized quasi-monte carlo financial simulation on a fpga supercomputer, in: Second International Workshop on High-performance Reconfigurable Computing Technology and Applications, 2008, HPRCTA 2008, IEEE, 2008, pp. 1–8.
[62] Donglin Wang, Xueliang Du, Leizu Yin, Chen Lin, Hong Ma, Weili Ren, Huijuan Wang, Xingang Wang, Shaolin Xie, Lei Wang, Z. Liu, Mapu: A novel mathematical computing architecture, in: 2016 IEEE International Symposium on High Performance Computer Architecture, (HPCA), IEEE, 2016, pp. 457–468.
[63] Mei Wen, Nan Wu, Chunyuan Zhang, Wei Wu, Qianming Yang, Changqing Xun, Ft64: scientific computing with streams, in: International Conference on High-Performance Computing, Springer, 2007, pp. 209–220.
[64] Mei Wen, Nan Wu, Chunyuan Zhang, Qianming Yang, Jun Ren, Yi He, Wei Wu, Jun Chai, Maolin Guan, Changqing Xun, On-chip Memory System Optimization Design for the ft64 Scientific Stream Accelerator. Vol. 28, IEEE, 2008, pp. 51–70.
[65] L. Yan, W. Hu, T. Chen, Z. Huang, Hardware assistant scheduling for synergistic core tasks on embedded heterogeneous multi-core system, J. Inf. Comput. Sci. (2008).
[66] Peter Yiannacouras, J. Gregory Steffan, Jonathan Rose, Vespa: portable, scalable, and flexible fpga-based vector processors, in: Proceedings of the 2008 International Conference on Compilers, Architectures and Synthesis for Embedded Systems, ACM, 2008, pp. 61–70.
[67] Jason Yu, Christopher Eagleston, Christopher Han-Yu Chou, Maxime Perreault, Guy Lemieux, Vector processing as a soft processor accelerator, in: ACM Transactions on Reconfigurable Technology and Systems (TRETS), Vol. 2, ACM, 2009, p. 12.
[68] Lixin Zhang, Zhen Fang, Mike Parker, Binu K. Mathew, Lambert Schaelicke, John B. Carter, Wilson C. Hsieh, Sally A. McKee, The impulse memory controller, IEEE Trans. Comput. 50 (2001) 1117–1132.
[69] Chuanjun Zhang, Frank Vahid, Using a victim buffer in an application-specific memory hierarchy, in: Proceedings of Design, Automation and Test in Europe Conference and Exhibition, 2004., Vol. 1, IEEE, 2004, pp. 220–225.

Tassadaq Hussain received **Ph.D. degree** in computer architectures at the Universitat Politécnica de Catalunya (UPC) in collaboration with Barcelona Supercomputing Center and Microsoft Research Center (BSCMSRC).

He obtained **M.Sc. (Electronics)** degree in 2009 from the **Institut Supérieur d'Electronique de Paris France**. He worked for Infineon Technology digital design department south France. During the stay in Infineon, he worked over Ultra-low Cost Mobile Base Band Chips, from September-2009 to December-2014.

Tassadaq is working as **Assistant Professor** in **Riphah International University Islamabad** and serving **Unal Color of Education Research and Development** as **Research Director**. His main research interests include heterogeneous multi-core architectures with the focus on efficient scheduling, data and access patterns management strategies for HPC applications.