PMSS: A programmable memory system and scheduler for complex memory patterns

Tassadaq Hussain,*, Amna Haider, Eduard Ayguadé

*Barcelona Supercomputing Center, Barcelona, Spain
\^Unal Center of Education Research & Development, Barcelona, Spain

HIGHLIGHTS

- In this article, we propose PMSS a Programmable Memory System and Scheduler.
- The PMSS can operate without intervention of master core or Operating system.
- It schedules multi-accelerators and manages their memory access patterns.
- The system is evaluated with memory intensive accelerators tested on a Xilinx ML505 evaluation FPGA board.
- Results show that the PMSS system achieves 19x of speed-up compared to generic multi-accelerator system.

ABSTRACT

HPC industry demands more computing units on FPGAs, to enhance the performance by using task/data parallelism. FPGAs can provide its ultimate performance on certain kernels by customizing the hardware for the applications. However, applications are getting more complex, with multiple kernels and complex data arrangements, generating overhead while scheduling/managing system resources. Due to this reason all classes of multi-threaded machines – minicomputer to supercomputer – require to have efficient hardware scheduler and memory manager that improves the effective bandwidth and latency of the DRAM main memory. This architecture could be a very competitive choice for supercomputing systems that meet the demand of parallelism for HPC benchmarks. In this article, we proposed a Programmable Memory System and Scheduler (PMSS), which provides high speed complex data access pattern to the multi threaded architecture. This proposed PMSS system is implemented and tested on a Xilinx ML505 evaluation FPGA board. The performance of the system is compared with a microprocessor based system that has been integrated with the Xilkernel operating system. Results show that the modified PMSS based multi-accelerator system consumes 50% less hardware resources, 32% less on-chip power and achieves approximately a 19x speedup compared to the MicroBlaze based system.

© 2014 Elsevier Inc. All rights reserved.

1. Introduction

Deep research has been conducted to improve the performance of HPC systems. One way to improve the performance is to build a multi-accelerator/core system [12,33], manage/schedule [3,5] its hardware and memory [23,36] resources efficiently and write parallel code [13,22] to execute on the system. A task-based programming model [7,28,32] is significant for such architectures, as it identifies the tasks in software that can be executed concurrently.

It also provides a bridge between hardware and software, which means that high-level languages can be compiled and efficiently executed in hardware.

The effects of the memory wall can be observed in the multi-processors systems having transaction-based workloads [26], where processors remain stall during 80% of the time [14,15] (seen in Fig. 1). The load and store stall time include delay while reading and writing data to external memories. The stall time is mainly dependent on the resource scheduling, input/output synchronization, and memory accesses. Fig. 1 shows that a major portion of the system time is consumed by memory load/store accesses. Precise management of the system memory resources can improve performance of the overall system. Integrating multi-core platforms on FPGA while using low clock frequencies can balance the processor
memory clock speed. However, the cumulative memory bandwidth requirement of all processing elements is still increasing, adding a new dimension to the problem. By integrating more memory controllers [4,31] on system platform can increase the bandwidth, but it has drawbacks which are as follows.

- SDRAM controller is an expensive component both in terms of area and power consumption.
- More number of input/output pins is required.

Modern platforms often contain multiple heterogeneous processing elements, to provide a good balance between performance, cost, power consumption and flexibility. Hence multiple memory controllers are often not an option, emphasizing to use the existing SDRAM bandwidth as efficiently as possible. To improve performance of the system we implemented a memory controller in hardware called Programmable Memory System and Scheduler (PMSS) which handle data movement and computation tasks. This work extends the multi-accelerator system by arranging data/address in hardware and schedules computations without the intervention of processor (Master) core and operating system. We further parameterize on-chip memory to allow a designer to more powerfully trade area for performance scaling for data parallel applications. By combining all the functions into one chip, the FPGA based system becomes faster and less power consuming. With the support of standard C/C++ language for high level data access patterns and decreases the complexity while writing applications. These function calls support specialized hardware accelerator IPs.

Some salient features of the proposed PMSS architecture are included below:

- the PMSS based system can operate as stand-alone system, without support of the Master core and the operating system;
- the controller gathers multiple memory read/write requests and maximizes the utilization of SDRAM open banks. This removes the overhead from opening and closing rows as well as idle cycles on the data bus;
- the data bus is bi-directional and requires a number of clock cycles to change direction from read to write or write to read, again adding latency and wasting bandwidth. The PMSS separates the multiple individual memory accesses and read/write data to/from SDRAM in one transition of stream. This feature removes the overhead delay generated by the current and previous read/write requests;
- PMSS supports multiple hardware accelerators using event driven handshaking methodology;
- PMSS improves performance of the system by efficiently prefetching complex patterns;
- due to the light weight (logic element) of PMSS the system consumes less power;
- standard C/C++ language calls are supported to identify tasks in software.

2. DDR memory access delays

The behavior of an SDRAM memory is dependent upon the SDRAM commands used by memory access unit to communicate with the memory device. A generic FPGA DDR controller [1,40] consumes almost 30 cycles for a single load/store memory access. The controller takes 1 to 3 handshaking signals to communicate with requesting source (processor). The bank is available for subsequent row activation. After a specified time $t_{RP}$ the PRECHARGE command is issued. The PRECHARGE command is used to read/write row from a bank. The command deactivates the open row in a particular bank. Before a read or write command is issued to a bank within the DDR2 SDRAM memory, a row in the bank must be activated using an Active command. After a row is opened, read or write commands can be issued to the row subject to the $t_{RCD}$ specification. These commands tell the memory to activate (open) a particular row in the memory array, to read from or write to an open row, or to precharge (close) an open row and store its contents back into the memory array. There is also a refresh command that charges the capacitors of the memory elements to ensure that the contents of the memory array are retained. Scheduling SDRAM commands is not a simple task, since there are a considerable number of timing constraints that must be satisfied before a command can be issued. These timing constraints are minimum delays between issuing particular SDRAM commands, such as two activates, or an activate and a read or a write. Table 1 shows clock cycles taken by DDR memory after read/write commands are given with valid data.

3. Programmable memory system and scheduler (PMSS)

Programmable Memory System and Scheduler (PMSS) is based on high level data patterns that simplify programming of HPC applications while ensuring high performance and efficiency. To present the functioning of PMSS, we first depict its inner architecture in Fig. 2, which also briefly shows the interconnection with the external processing units. PMSS design supports multiple hardware accelerators using special event driven handshaking methodology. Each hardware accelerator has local scratch pad memory to operate. Therefore, other events cannot affect the execution process.

During compile-time an API is used which separates, pipelines, overlaps and schedules memory read/write operations and generates executable PMSS binary file. At initialize-time (a), the PMSS
uses Program Line to write PMSS binary file to the Descriptor Memory. At run-time (b), the Memory Manager accesses a complex memory pattern from Descriptor Memory. Depending upon the access pattern the Memory Manager takes single or multiple descriptors from Descriptor Memory and accesses data to/from Buffer Memory and Stream Prefetcher. During step (c), the Memory Manager takes scheduling information from the Scheduler and prioritizes memory accesses appropriately. The Scheduler keeps gathering memory requests from external sources and place them in the local buffer. During step (d and e), Stream Prefetcher takes a single descriptor from Memory Manager and read/write data to/from SDRAM memory. During step (f), the Scheduler provides a link to the external compute unit to access Data Memory. The entire PMSS system for multi-accelerator can be fully described as six separate units: the front end interface, the memory system, the scheduler, the memory manager, the stream prefitcher, and the memory controller.

3.1. Front end interface

The front-end interface provides a link between PMSS and computing units. It includes two distinct interfaces, the Scratchpad Controller Interface and the High-Speed Source Synchronous Interface, to link with the microprocessor unit (master) and the hardware accelerator unit (slave), respectively.

3.1.1. Scratchpad controller interface

The Bus Interface as shown in Fig. 2 provides a link between the master unit (microprocessor) and the Scratchpad Memory. It also maintains the information including the program code (memory access patterns) and makes it available to the PMSS unit. The Bus Interface is used to initialize the Descriptor Memory with the support of PMSS device drivers calls such as send(), receive(), send_tile(), receive_tile() and mem_copy(). The send() and receive() occupy a single Descriptor Memory block, whereas send_tile(), receive_tile() and mem_copy() occupies descriptor blocks according to the volume of the dataset.

3.1.2. High-speed source synchronous interface

The High-Speed Source Synchronous Interface (Fig. 2) is used to supply high-speed data to hardware accelerators. To request and grant data for multiple accelerators, a handshaking protocol is applied to the bus arbiter. The transfer of data is accomplished according to the physical memory clock.

3.2. Scratchpad memory

The scratchpad memory subsystem consists of the Buffer Memory along with the Descriptor Memory. The Buffer Memory is used to hold data temporarily while it is being moved to/from physical memory. Depending on the system architecture and applications access pattern the Buffer Memory can be partitioned.

To reduce stall time the PMSS sets memory access descriptors in Descriptor memory in priority order. The Memory Manager (see Section 3.4) accesses these descriptor blocks without any delay, thus reducing the additional synchronization time required to access non-contiguous memories. Each descriptor block contains information of source unit and destination unit that eliminates request/grant time. The PMSS descriptor memory holds the information of multi-accelerator memory access patterns and scheduling methodology. The descriptor memory is a fixed length data word with a number of attribute fields that describe the access pattern. A single descriptor describes the access pattern of a strided stream, and multiple descriptors are used to define complex memory patterns. The least set of parameters for a memory descriptor block, includes command, source address, destination address, stride, stream and priority. Command specifies whether to read/write a single element or a stream of data. The address parameters specify the starting addresses of the source and destination locations. Stride indicates the distance between two consecutive physical memory addresses of a stream. The priority describes the selection and execution criteria of a process. The program structure of PMSS is stack based where the descriptor holds information of complete memory access pattern. The machine code of PMSS consists of just an opcode that activate/deactivate accelerator kernel. Memory management and optimization for repeated memory access is done at compile time and are executed in hardware at runtime. The proposed PMSS system provides C and C++ language support that initializes descriptor blocks. The program example of a single memory access is presented in Fig. 3. The local address and external address parameters hold the start address of (buffer) memory and main memory (SDRAM) data set respectively. The Priority defines the order in which memory access is entitled to be processed. The parameters Stream and Stride define the type of memory access. The value 0x08 and 0x00 of Stream and Stride parameters respectively initialize PMSS to access a stream of 8-words with a unit stride. At run-time, the PMSS accesses 32-bit word from the 0x000000100 address location of main memory and write it to the 0x000000000 address location of Buffer Memory. The following memory access till the end of stream, the PMSS generates main memory address by taking the address of main memory from the previous access and adds stride (0x08) value. The Buffer Memory address is sequential (contiguous) and requires unit increment.

---

**Fig. 2.** PMSS architecture.

**Fig. 3.** PMSS function call: single descriptor program.
The priorities are assigned by the programmer and can be arranged at compile time depending upon the size of stream and stride. There is no value limit of stream and stride except it should not exceed main memory and local memory address range. After transferring of 1K words of stream PMSS checks weather, any other application with higher priority is waiting. If an application with higher priority is waiting, PMSS will hold current transfer and process waiting application.

3.3. The scheduler

The Scheduler manages read, write and execute operations of multiple accelerators. At the end of each accelerator’s operation, the PMSS invokes the scheduler to select the next accelerator. This selection depends on the accelerator’s requests and priority level. The accelerators are categorized into three states, busy (accelerator is processing on local buffer), requesting (accelerator is idle), and request and busy. In the request and busy state the accelerator is assumed to have double or multi buffers. In this state, the accelerator is processing a buffer while making requests to fill other buffers. To provide a feature of multi buffer support in the current devolved platform, a state controller (see Fig. 4(b)) is instantiated with each accelerator that handles the states of accelerator using a double buffering technique. The state controller manages the accelerator’s Request and Grant signals and communicate with the scheduler. Each Request includes a read and write buffer operation. Once the request is accepted the state controller provides a path to the PMSS read/write buffer.

PMSS supports two scheduling policies, symmetric and asymmetric, that execute accelerators efficiently. In Symmetric multi-accelerator strategy, the PMSS scheduler manipulates the available accelerator’s request in FIFO (First in First out). The task buffer (shown in Fig. 4(a)) manages the accelerator’s request in FIFO order. The Asymmetric strategy emphasizes on priority and incoming requests of the accelerators. Like Xilinx Xilkernel scheduling model, the PMSS scheduling policies are configured at program-time and are executed by hardware at run-time. The number of priority levels can be configured for asymmetric scheduling. Assigned priorities of the accelerators are placed in the programmed priority buffer (see Fig. 4(a)). The comparator picks an accelerator to execute, only if it is ready to run and there are no higher priority accelerators that are ready. If same priorities are assigned for more than one accelerator, PMSS scheduler executes them as the first-in first-out (FIFO) method.

The scheduler unit accumulates requests from accelerators and maintains them in the task buffer as per predefined scheduling policy. In Asymmetric scheduling mode, requests are organized in priority and requesting order. For example, different requests (A1 to A5) are generated concurrently shown in Fig. 5(a). The scheduler processes first request (A1) as it is (Fig. 5(c)), remaining requests are executed based on the priority level defined in programmed priority buffer (see Fig. 5(b)). When two concurrent requests (A3 and A4) are generated the PMSS scheduler selects one which has the highest priority placed in the programmed priority buffer. When the currently running accelerator process finishes, the PMSS does a selection and select the next accelerator. Depending upon the scheduling policies, the new accelerator is selected that has the highest priority and in request or request and busy state.

The Memory Manager (Section 3.4) of PMSS has particular descriptor memory (register set) for each accelerator unit. These descriptors are masked with interrupt and request signal. Once a request is generated, the Memory Manager starts memory operation for the requested accelerator using its descriptors. After the completion of memory read/write operation, the PMSS scheduler receives an interrupt (ack) signal from the memory manager unit. This signal informs the scheduler about the selection of the next accelerator to process. The scheduler captures the ack signal from the memory manager and assigns the grant signal to the appropriate accelerator unit.

3.4. Memory manager

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

Fig. 4. PMSS: (a) Scheduler (b) State Controller.
3.4.1. Memory organization

PMSS core holds information of physical memory and local memory address space that is partitioned into segments (Fig. 6(a)). The segments are organized in the same way as the program is written. PMSS provides protection at the segment level e.g. segment can be read/written by the accelerator for which it is allocated. Multiple noncontiguous accesses of memory leads to a high degree of read/write delay due to the control selection of SDRAM memory. PMSS overcomes this problem by organizing multiple noncontiguous memory access together.

3.4.2. Locality and isolation

The PMSS provides isolation by allocating separate register set (Descriptor Memory) and scratchpad (local) buffers for each accelerator. This guarantees that no other accelerator can access the memory belonging to an accelerator. In the PMSS system, the local scratchpad memory is tightly connected to the accelerator. This avoids the cost of transferring memory content in terms of latency and core stalling. PMSS keeps the knowledge of memory whether a certain memory area is in the accelerator's local scratch pad. This knowledge allows the PMSS to manage the placement of memory as well as reusing and sharing already accessed memory. As the dataset access pattern and size description are known at program time, PMSS efficiently utilizes these access patterns at run time.

3.4.3. Programmability

The PMSS provides instructions to allocate and map application kernel local memory buffer and physical dataset. A PMSS_MEMCPY instruction is created which reads/writes a block of data from physical memory to the accelerator's local memory buffer. PMSS improves the command efficiency, which is data movement dependent, and can be managed at run time. By managing read/write command PMSS reduces delay occurred by other commands that are already issued in the access cycle. PMSS gathers consecutive read/write commands that increases the burst length, as smaller bursts result in more activate and precharge commands which degrade the command efficiency.

3.5. Stream prefetcher

The stream prefetcher takes a dataset (main memory) description from the Memory Manager as shown in Fig. 6(b). This unit is responsible for transferring data to/from the memory controller and accelerator buffer memory. It takes the strided streams description from the Memory Manager and read/write data from/to physical memory. The Stream prefetcher enables the data stream to be written to the appropriate physical memory by generating the write-enable along with the write-data and mask-data control signals.

3.6. Memory controller

The memory controller translates physical addresses given by the processor into memory addresses in the form of channel id, rank id, bank id, row id, and column id. A modular DDR2 SDRAM controller is used in the current PMSS architecture, to access data from physical memory. The DDR2 SDRAM controller provides a high-speed source-synchronous interface and transfers data on both edges of the clock cycle.

4. Use case example

To explain the working principle of the PMSS system, in this section we briefly describe its supported access patterns and programming.

4.1. Data access pattern

In order to elaborate the functionality of PMSS two types of Data access patterns are discussed in this section. Vector Access Pattern (AoS or SoA) and Tiling Access Pattern.

4.1.1. Vector access pattern (AoS or SoA)

It has been observed [11] that most HPC applications are in favor of operating on SoA format. The PMSS system is one way to access Structure of Array (SoA) in the Array of Structure (AoS) format without generating any delay. AoS or SoA data access is performed by the send() and receive() calls. Each call is associated with
a single descriptor block which needs to be initialized for each transfer. SoA data access requires unit-stride, whereas the AoS requires strided access. The stride is determined by the size of the working set. Fig. 7(a) presents three different vector accesses (x[n] [0], y[0] [n] and z[n] [n]) for a n × n matrix where vector x corresponds to the contiguous row having unit stride, vector y belongs to column access with stride equal to row length and vector z is a diagonal SoA pattern. Stride of z[n] [n] access is the addition of row length and unit stride.

4.1.2. Tiling access

The Tiling technique is employed to improve system performance. It is widely used for exploiting data locality and improving parallelism.

**Single Tile Access:** to access a single tile, the PMSS uses multiple descriptors. By combining these descriptors, the PMSS exchanges tiled data between the physical memory and scratch pad memory buffer. To read/write a single tile, the send_tile() and receive_tile() calls are used to initialize the descriptor memory blocks. Each call requires two input parameters Buffer_# and Physical_Address. Parameter Buffer_# indicates the starting address of the buffer where the tile is read/written. Physical_Address holds start address of the working dataset. **Multi-Tiling Access:** the PMSS multi-tiling method attempts to derive an effective tiling scheme. It requires information about the Physical Memory Tiled Dataset (MxN) and Buffer’s Tile size (MxN) of computing engine. Where (M, m) represents width and (N, n) represents the dimension of each tile. Depending upon the structure of the buffer memory, the PMSS partitions Physical memory into multiple tiles. The number of tiles for multidimensional memory access is given by Eq. (1).

\[
\text{Number of Tiles} = \left[ \frac{\text{Dataset Width}}{\text{LocalBuffer Width}} \times \frac{\text{Dimension}}{\text{Dimension}} \right].
\]  

An example of PMSS (4 × 4) 2D-tile access is shown in Fig. 7(b). The PMSS uses 4 descriptors to access a single tile. Channel 0 takes descriptor 0 from descriptor memory and accesses data starting at location Physical_Address with unit-stride and n-stream (row width) size. Depending on the size of the main memory dataset (MxN), different jumps (address) are used between channels. The starting address of the next channels is dependent on Eq. (2). Channel 1 starts right after the completion of Channel 0. Channel 3 is the last channel. After the completion of channel 3, PMSS generates an interrupt signal indicating to the external source that the tile data transfer has finished. The PMSS will move ahead to the next tile and restart processing.

\[
\text{Dataset BaseAddress}_\text{tile} + \text{Buff Width}_\text{tile} \times \text{ch num}.
\]

4.2. PMSS programming

The program structure of PMSS is stack based where read and write accesses are separated with descriptor of complete memory access pattern. The machine code of PMSS consists of just an opcode that activate/deactivate accelerator kernel. As the PMSS compiler is simple, the code generation is independent of prior or subsequent program. Memory management and optimization for repeated memory access is managed in the compiler which is executed in hardware at runtime.

The proposed PMSS system provides C and C++ language support. The system can be managed by C based device drivers. An example used to program the PMSS based multi-accelerator system is shown in Fig. 8. The program initializes two hardware accelerators and their 2D and 3D tiled data pattern. The first part of the program structure specifies the scheduling policies that include the accelerator id and its priority. The PMSS scheduler supports the scheduling policies similar to the Xilkernel. If the same priority is assigned to multiple accelerators, the PMSS scheduler processes the accelerators in symmetric mode. The second part of the code belongs to the physical memory dataset. The PMSS automatically adjusts the dataset into appropriate tile sizes, to be processed by the hardware accelerator. The third part defines the size of the accelerator’s buffer memory. The same programming style is used for other hardware accelerators.

To program the PMSS a MicroBlaze API is used that translates the multi-accelerator program for PMSS system. The program flow is shown in Fig. 8. In the first step, the API takes C code (shown in Fig. 8) and applies parallel extraction. During compile-time, the API separates read, write, and compute operations. In the second step, it pipelines and overlaps the PMSS operations and then translates it to PMSS binary file. MicroBlaze is used that takes the binary file of PMSS and program its Descriptor Memory. At run-time, the PMSS executes the operations depending on its state.

PMSS removes the programmer effort of manually arranging memory accesses. A wrapper library that contains (function calls i.e. 3D_Sten()) information of application complex access patterns. The programmer only needs to identify the appropriate call to the library function call and the PMSS automatically transfers data pattern to local memory of compute unit. A PMSS API is used to parse the annotated code and then program the PMSS descriptor memory. The PMSS API is integrated with the GCC compiler included in Xilinx Platform Studio and uses MicroBlaze processor. Programmer has to annotate PMSS code using PMSS_MEMCPY and PMSS_DESCRIPTOR function calls for applications. The minimum requirement of each access pattern is that the programmer has to describe main memory data sets and local buffer size of application kernels. The translation of regular memory patterns from annotated code to PMSS binary file is automatic which exploits the parallelism. The PMSS provides PMSS_MEMCPY function call to read/write dense data structure and access pattern. The PMSS takes PMSS_MEMCPY parameters from the code and automatically rearranges and parallelizes access patterns in software and program the PMSS descriptor memory. For the complex memory patterns having load/store access, the programmer parallelizes the code by...
5. Evaluations of PMSS

In this section, we describe and evaluate the PMSS-based multi-accelerator system having Thresholding, Radian Converter, FIR, FFT, Matrix Multiplication, Smith Waterman, Laplacian solver, and 3D-Stencils application kernels. In order to evaluate the performance of the PMSS System, the results are compared with a similar system having MicroBlaze master core. The Xilinx Integrated Software Environment [21] and Xilinx Platform Studio [43] are used to design systems. Xilinx Power Estimator [42] is used to analyze the system power. A Xilinx ML505 evaluation FPGA board [30] is used to test the multi-accelerator system.

5.1. MicroBlaze based multi-accelerator system

A MicroBlaze based Multi-Accelerator system is proposed (Fig. 9(a)). The MicroBlaze soft-core processor is used that controls the resources of the system. The Real-Time Operating Systems (RTOS) Xilkernel [41] is executed on the MicroBlaze soft processor. The Xilkernel has POSIX support and can declare threads at program time that start with the kernel. From the main, application is spawned as multiple parallel threads using the pthread library. Each thread controls a single hardware accelerator and its memory access. The Xilinx Multi-Port Memory Controller (MPMC) is employed as it provides an efficient means of interfacing the processor to SDRAM. MicroBlaze accesses memory either through the On-chip Peripheral Bus (OPB) port or through the On-chip Peripheral Bus (OPB). The LMB provides fast access to on-chip block RAM (BRAM) memories. The OPB provides a general purpose bus interface to on-chip or off-chip memories as well as other non-memory peripherals. The target architecture has 16 KB of each instruction and data cache. The design (excluding hardware accelerators) uses 7225 flip-flops, 6142 LUTs and 14 BRAMs.

5.2. PMSS based multi-accelerator system

The PMSS based multi-accelerator system is shown in Fig. 9(b). PMSS schedules accelerators similar to the Xilkernel scheduling model. Scheduling is done at the accelerator event level. The PMSS contains hardwired scheduler whereas Xilkernel performs scheduling in software while using few hardware resources. In the current design, each hardware accelerator is equipped with two read/write buffers to balance the workload. The read/write
buffers are connected with PMSS via source synchronous interface and a state controller (Fig. 4). In the current implementation of PMSS, on a Xilinx ML505 evaluation FPGA board, a modular DDR2 SDRAM [29] controller is used with PMSS to access data from physical memory and to perform the address mapping from physical address to memory address. A 256 MByte (32M × 16) of DDR2 memory having SODIMMI/O module is connected with memory controller. The system (excluding hardware accelerators) consumes 3786 flip-flops, 2830 LUTs, 24 BRAMs.

5.3. Test applications

The application kernels that are used in the design are shown in the Table 2. These kernels are selected to test the performance of the system for different data access patterns. The results are validated by comparing the execution time of these kernels on the PMSS system and with the MicroBlaze based system. Separate ROCCC [37] generated hardware IP cores are used to execute the kernels. In order to give a standard control and interface to IPs with the PMSS system, the state controller is used.

6. Results and discussion

This section analyses the results of different experiments conducted on the PMSS and MicroBlaze based system. The experiments are characterized into four subsections: Memory Access; Application Performance; System Performance; and Power.

6.1. Memory access

We compared PMSS systems memory access time with MicroBlaze processor based system by executing thresholding application having load/store memory access pattern. Fig. 10 shows a plot with Read/Write data accesses. The X-axis presents data sets that are read and written by image thresholding application from/to the main memory. The Y-axis of the plot represents the number of clock cycles consumed, while accessing the dataset. The main memory single access latency on a 125 MHz MicroBlaze processor with 125 MHz DDR SDRAM is measured almost 50 cycles. The results show that PMSS system while accessing complex memory accesses are 1.4 times faster than MicroBlaze based system. The reason of this speed up is PMSS manages address in the hardware. The memory access speed up is furthered improved up-to 33x for tiled access pattern. This is because PMSS manages complex patterns independently without the support of processor/OS and uses few descriptors that reduce run-time address generation and (on-chip) address request/grant delay.

6.2. Application’s performance

The application kernels are executed on PMSS and MicroBlaze based systems. Fig. 11 shows the execution time (clock cycles) of the application kernels. Each bar represents the application kernel's computation time on the hardware accelerator and execution time on the system. The application kernel time contains task execution, scheduling (request/grant) and data transfer time.
The X and Y axis represent application kernels and number of clock cycles, respectively. By using PMSS system, the results show that Thresholding and Radian converter applications achieve 3.5× speed-ups compared to the MicroBlaze based system. These applications have load/store memory access pattern and achieves less speed-ups compared to other application kernels. The FIR application has streaming data access pattern with 26.5× speed-up. The FFT application kernel reads a 1D block of data, processes it and writes it back to physical memory. This application achieves 11.9× speed-up. The Matrix Multiplication kernel accesses row and column vectors. The application attains 14× speed-up. The Laplacian and Smith Waterman applications take 2D block of data and achieve 36× and 38× speed-ups respectively. The 3D-Stencil data decomposition achieves 52× speed-up. This speed-up is gained because PMSS stores 3D access patterns in descriptor memory which reduces the address generation time.

6.3. System performance

In the multi-accelerator system, the total execution time includes multiple delays such as interconnect, memory architecture, cache coherence and memory consistency protocols, bus arbitration, on-chip/off-chip bus translation and flow control. Fig. 12 illustrates the execution time of the system and categorizes the execution time into two factors: arbitration (request/grant) time among the scheduling, and the memory management (bus delay and memory access) time. The computation time of application kernels in both systems overlaps under the scheduling and memory access time (shown in Fig. 11). In the PMSS system memory management time is dominant, and the PMSS overlaps scheduling and computation under memory access time. The complete PMSS multi-accelerator system achieves 18.6× speed-up.

6.4. Power

Studies [34] have shown that discrete GPUs can offer performance higher than CPUs and FPGAs. However, a compute-capable discrete GPU can draw more than 200 watts by itself. The On-Chip Power in a Xilinx V5-Lx110T device is 1.97 W while running MicroBlaze based system. PMSS system draws 1.33 W on-chip power on a V5-Lx110T device. Due to the light weight of PMSS, the architecture consumes 50% less slices and 32% less on-chip power than the MicroBlaze based system.

7. Related work

A number of scheduling and memory management approaches exist for multi-accelerators, but to the best of our knowledge a challenge is still there to find mechanisms that can schedule dynamic operations while taking both the processing and memory requirements into account. McKee et al. [27] introduced 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. The PMSS system prefetches both regular and complex streams and also supports dynamic streams whose addresses are dependent on run-time computation. McKee et al. also proposed the Impulse memory controller [6], which supports application-specific optimizations through configurable physical address remapping. By remapping the physical addresses, applications can manage the data to be accessed and cached. The Impulse controller works under the command of the operating system, relies on caches memory system and performs scheduling and physical address remapping in software, which may not always be suitable for HPC applications using hardware accelerators. PMSS schedules address patterns and remaps and produces physical addresses in the hardware unit without the overhead of operating system intervention. At run-time PMSS uses buffer memory to organize data in the form of patterns. Based on its C/C++ language support, PMSS can be used with any operating system that supports the C/C++ stack.

A multi-accelerator having 16 AMD dual-core CPU computing nodes with 4 NVIDIA GPUs and a Xilinx FPGA is presented by Showerman et al. [35]. The cluster nodes are interconnected with both infiniband and Ethernet networks. The software stack consists of standard cluster tools, the accelerator-specific software package, enhancement of the resource allocation and branch subsytem. Anderson [2] has proposed and implemented a scheduling
method for real-time systems for multicore platforms that encourage certain groups of tasks to be scheduled together while ensuring real time constraints. This method can be applied to encourage tasks that share a common working set to be executed in parallel, which makes more effective use of shared caches. Wolf et al. [39] provide a real-time capable thread scheduling interface to the two-level hardware scheduler of MERASA multi-core processor. A time-bounded synchronization mechanism for the concurrent threads execution is proposed for multi-core architecture. The architecture is capable of executing hard real-time threads.

S. Zhuravlev et al. [45] presented an intelligent scheduler that is aware of underlying caches and schedules applications with respect to memory demands which generate the hardware caches contentions between threads. The PMSS system overcomes this problem by scheduling task operations while taking into account the programmed priorities, on-chip specialized memories and runtime memory accesses patterns.

Ferrante [9] has developed a scheduling algorithm which allows distributing IPSec – a suite of protocols that provides security to communications at IP level – packet processing over the CPU and multiple accelerators and to support soft QoS. He provides some high-level simulations to prove that the algorithm works as desired and that it can provide a performance enhancement especially when the system is overloaded. Marchand et al. [25] have developed software and hardware implementations of the Priority Ceiling Protocol that control the multiple-unit resources in a uniprocessor environment. Yan et al. [44] has designed a hardware scheduler to assist the synergetic processor cores (SPCs) task scheduling on a 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 scouting hardware thread [24] tends to reduce latency, but also optimize memory bandwidth usage by predicting memory accesses and by prioritizing valuable memory traffic by using a separate core. The information of memory accesses is stored thus helping the scouting core to fetch and update data from the cache. The PMSS holds information of memory patterns in the form of descriptor memory. Currently accessed patterns are placed in the address manager of PMSS. The PMSS monitors the access patterns without using a separate core and reuses these patterns for multiple cores if required.

Ganuso et al. [10] has proposed Efficient Emulation of Hardware Prefetchers via Event Driven Helper Threading (EDHT). EDHT gives the idea of using accessible general purpose cores in a chip multiprocessor environment. It acts as helper engine for separate threads working on the active cores. Wen et al. [38] presented a FT64 based on chip memory sub system that combines software/hardware managed memory structure. The stream accelerator is a HPC application specific coprocessor. It combines caching and software managed memory structures, capturing locality exhibited in regular/complex stream access without data transfer between stream register file and caches. Chai et al. [8] presented a configurable stream unit for providing streaming data to hardware accelerators. The stream unit is located in the system bus, and it prefetches and aligns data based on streams descriptors. PMSS improves on-chip communication bandwidth by managing both compile- and run-time generated access pattern descriptors without support of extra master processor. At run-time PMSS schedules the access patterns with respect to the available resources such as buses and memories.

Hussain et al. [16,17,19] discussed the architecture of a pattern based memory controller for application specific single accelerator. He also provides a memory controller [18] for single core vector processor. The design is appropriate only single core cores, whereas in PMSS we present a mechanism both for multi-core accelerators. Moreover, features of PMSS like the Scheduler and Memory Manager enable higher performance of multi-core applications.

8. Conclusion

In this work, we have proposed PMSS for multi-accelerator environment. The technique improves the system performance by reducing accelerator/router and memory speed gap, and schedule/manage complex memory patterns without processor intervention. The PMSS system provides strided, scatter/gather and tiled memory access support that eliminates the overhead of arranging and gathering address/data by the master core (microprocessor). The proposed environment can be programmed by a microprocessor using High Level Language (HLL) API or directly from an accelerator using a special command interface. The experimental evaluations based on the Xilinx MicroBlaze multi-accelerator system having Xilinx Kernel (RTOS) demonstrates that PMSS based multi-accelerator system best utilizes hardware resources and efficiently accesses the physical data. In the future, we are planning to embed a selective static/dynamic set of data access pattern inside PMSS for multi-accelerator (vector accelerator) architecture that would effectively eliminate the requirement of programming PMSS by the user for a range of applications.

Acknowledgments

This work has been supported by the Ministry of Science and Innovation of Spain (CICYT) under contract TIN–2007–60625 and by the European Union Framework Program 7 HiPEAC2 Network of Excellence. The authors would like to thank the Barcelona Supercomputing Center, Microsoft’s Barcelona Research Center, Ural Center of Education Research and Development (UCERD) and the Universitat Politècnica de Catalunya (UPC) for their support. The authors also wish to thank the reviewers for their insightful comments.

References

[28] Xilinx, Memory Interface Solutions, December 2, 2009.