PAMS: Pattern Aware Memory System for Embedded Systems

Tassadaq Hussain[1,2], Nehir Sonmez[1] Oscar Palomar[1,2],
Osman Unsal[1], Adrian Cristal[1,2,3], Eduard Ayguadé[1,2], Mateo Valero[1,2]
[1] Computer Sciences, Barcelona Supercomputing Center, Barcelona, Spain
[2] Departament d'Arquitectura de Computadors, Universitat Politècnica de Catalunya, Barcelona, Spain
[3] Artificial Intelligence Research Institute (IIIA), Centro Superior de Investigaciones Científicas (CSIC), Barcelona, Spain
Email: {first}.{last}@bsc.es

Abstract— **In this paper, we propose a hardware mechanism for embedded multi-core memory system called Pattern Aware Memory System (*PAMS*). The *PAMS* supports static and dynamic data structures using descriptors and specialized memory and reduces area, cost, energy consumption and hit latency. When compared with a *Baseline Memory System*, the *PAMS* consumes between 3 and 9 times and 1.13 and 2.66 times less program memory for static and dynamic data structures respectively. The benchmarking applications (having static and dynamic data structures) results show that *PAMS* consumes 20% less hardware resources, 32% less on chip power and achieves a maximum speedup of 52x and 2.9x for static and dynamic data structures respectively. The results show that the *PAMS* multi-core system transfers data structures up to 4.65x faster than the MicroBlaze baseline system.**

I. Introduction

With the unveiling of on-chip memories such as memristors [1] and embedded DRAMs [2] the size of on-chip memory is getting a dramatic increase. As the amount of on-chip gates increases, there is a dramatic increase in size and architecture of local memories such as caches and scratchpads. Cache memories [3] are very effective but only if the working set fits in the cache hierarchy and there is locality. The concept of Scratch-Pad memory [4] is an important architectural consideration in modern HPC embedded systems, where advanced technologies have made it possible to combine with DRAM. 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 [5] [6] inside application specific processors improve performance of the memory hierarchy. The register file can improves the performance for applications having data locality, but does not support applications with large and irregular 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 the system platform can increase bandwidth, but would also require a number of Input/Output pins that consume power and routing overhead. Therefore, a memory system requires an intelligent memory controller that manages and schedules the data accesses. In this work, we propose the Pattern Aware Memory System (PAMS), a memory system for 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.

The research leading to these results has received funding from the European Research Council under the European Unions FP7 (FP/2007-2013) / ERC GA n. 321253. It has been partially funded by the Spanish Government (TIN2014-34557).

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 complete data structure into a special scratchpad memory. Data structures and memory accesses are arranged in the pattern descriptors and *PAMS* manages access patterns at run-time to reduce access latency. *PAMS* manages data movement between the *Main Memory* and the *Local Scratchpad Memory*; data present in the *Local Memory* is reused and/or updated when accessed by several memory transfers. The salient contribution of the proposed *PAMS* architecture are:

- Handles complex and irregular memory accesses at run-time, without the support of a processor or the operating system.
- Manages data between consecutive iterations of the memory accesses using *register file* and reuse data to minimize memory accesses.
- Supports both static and dynamic data structures using a parameterizable memory system that manages SDRAM rows/banks based on access patterns.
- When compared with the MicroBlaze baseline system implemented on the Xilinx FPGA, *PAMS* transfers memory patterns up to 4.65x faster and achieves between 3.5x to 52x and 1.4x to 2.9x of speedup for applications having static and dynamic data structures respectively.

II. Related Work

Scratchpads are a possible alternative to caches, being a low latency memory that is tightly coupled to the CPU [7]. Therefore it is a popular choice for on-chip storage in real-time embedded systems. The allocation of code/data to scratchpad memory is performed at compile time, leading to predictable memory access latencies. Panda et al. [8] developed a complete allocation strategy for scratchpad memory to improve the average-case program performance. They assume the access patterns are predictable and are available on top of the scratchpad memory. Therefore, the goal of the proposal is allocation strategy that minimizes the conflict among the variables in the local memory. Suhendra et al. [9] aimed at optimizing memory access tasks for worst-case performance. However, in the study, the scratch-pad allocation is for static and predictable access patterns that do not change during run-time. This raises performance issues when the amount of code/data is much larger than scratchpad size. Dynamic data structure management using scratchpad techniques are more effective in general, because they may keep the working set in scratchpad. This is done by copying objects at predetermined points in the program in response to execution [10]. Dynamic data structure management requires a dynamic scratchpad allocation algorithm to decide where the copy operations should be carried out. A time-predictable dynamic scratchpad allocation algorithm has been described by Deverge and Puaut [10]. The program is divided into regions, each with a different set of objects loaded into the scratchpad. Each region supports only static data structures, i.e. global and local variables. This restriction ensures that every program instruction can be trivially linked to the variables it might use. Udayakumaran et al. [11] 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 the objects are spilled into external memory when insufficient scratchpad space is available. The PAMS address manager arranges unpredictable memory access at run-time in the form of pattern descriptors. The PAMS is also able to perform data management and to handle complex memory accesses at run-time using 1D/2D/3D *Scratchpad Memory*.

Weisz et al. [12] presented a C to hardware compiler framework that generates CoRAM FPGA implementations of 2D loop nests from software source code. Pouchet et al. [13] proposed a framework that reuse data through loop transformation-based program restructuring. These frameworks generate accelerators which manage and process accessed data. The accelerator requires a master core which performs off-chip data access. The generated accelerators are used for the Convey machine, which manages off-chip data. Results indicate that loop transformation-based program restructuring increases the overall area growth on the FPGA and does not support complex and large loops (e.g. three-dimension and irregular code). On the other hand the *PAMS* can accomplish off-chip and on-chip data management and supports complex and irregular memory accesses. The current evaluation supports complex (i.e. three-dimension, pointer etc) memory accesses and can provide 3D data to processing core with a single cycle latency. The access patterns can be rearranged by re-programming the *PAMS* descriptor memory and does not require the re-synthesis.

Hussain et al. [14] also discussed the architecture of a pattern based memory controller for application specific single accelerator. He provided a memory controller [15] and [16] [17] for single core vector processor and graphics system respectively. The design is appropriate only for single core, whereas in *PAMS* we present a mechanism that supports *static* and *dynamic data structures* of real-time applications. Moreover, features of *PAMS* like the *Run-Time Address Management* enable higher performance of multi-core embedded systems.

III. Pattern Aware Memory System (PAMS)

In this work, we present a memory system (shown in Figure 1) that allocates *static* and *dynamic data structures* and improves the system performance by managing the data transfers in patterns. 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*.

A. Memory Organization

To provide isolation and improve data locality the *PAMS* memory is subdivided into three parts that are: the *Register File* the *Scratchpad Memory* and the *Main Memory*.

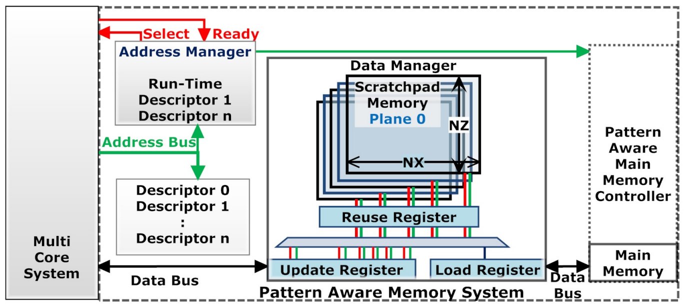


Fig. 1. Architecture of Pattern Aware Memory System

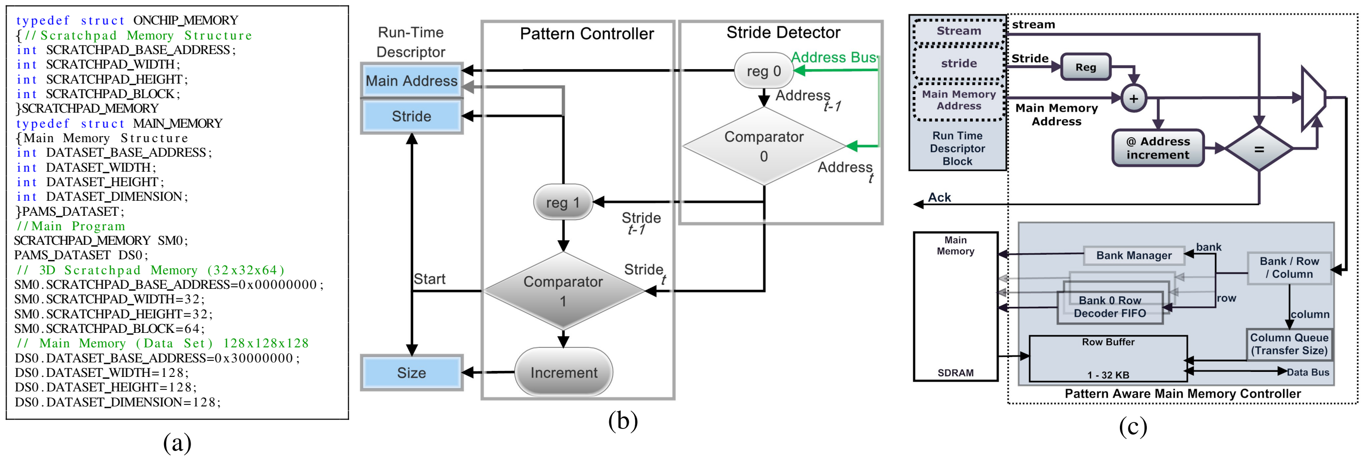
1) Register File: The Register File is fast and efficient as it provides parallel/patternized data access in a single cycle to the processing cores. It uses FPGA resources/slices that have multiple small independent memory register arrays. The *PAMS Memory Manager* (see Section III-C) splits, interleaves, or reorganizes registers that reuse the portion of data. The *PAMS* registers can be further categorized into the load register, reuse register and update register. The load register gets the input data pattern from the *Main Memory* using the *Memory Manager* and transfers those elements which are not already accessed or not present in the registers. The reuse register exploits the data reuse and forwards data requests which are not available to the load register. The update register manages the access patterns by taking the data elements from the load register and reuse register, transferring data to the processing cores in the form of patterns.

2) Scratchpad Memory: A programmable and parameterizable *Scratchpad Memory* [3] architecture acts as a cache in the system. It accesses the whole data pattern as a cache line and temporarily holds data to speedup later accesses. The structure of *Scratchpad Memory* is programmed according to the application data structures and access patterns. Unlike a cache, the accessed block can have data of non-contiguous memory locations and is deliberately placed in the *Scratchpad Memory* at a known location, rather than automatically cached according to a fixed hardware policy. A program structure that is used to initialize a 3D *Scratchpad Memory* is shown in Figure 2(a). The values for *SCRATCHPAD_WIDTH*, *SCRATCHPAD_HEIGHT* and *SCRATCHPAD_BLOCKS* describe the size of row *NX*, column *NZ* and the 3rd dimension (in this example 32x32x64) in size. The 32x32 size of the block is selected to fit in one BRAM of the target device, therefore, in the current evaluation, 64 BRAM blocks are used to design this 3D data memory structure. Furthermore, a *master port* (M) and *load register* are used to transfer data between the *Main* and *Scratchpad* Memories.

3) Main Memory: The slowest memory in the *PAMS* is the shared *Main Memory* which is accessible by the whole system. A program structure to define a 3D data set in *Main Memory* is also shown in Figure 2(a) for a 3D data set of 128x128x128 of size. The *PAMS* divides the *Main Memory* data set into tiles according to the *Scratchpad Memory* size. At run-time, the *PAMS* performs auto-tiling [18] and transfers a tile of data, having a size equivalent to *Scratchpad Memory*.

B. Data Structures and Access Description

Just like the cache hierarchy, the *PAMS* supports two types of data structure classes for memory allocation that are: the *static data structure* and the *dynamic data structure*, 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 unpredictable and unaligned data access requests.



```
typedef struct ONCHIP_MEMORY
{//Scratchpad Memory Structure
int SCRATCHPAD_BASE_ADDRESS;
int SCRATCHPAD_WIDTH;
int SCRATCHPAD_HEIGHT;
int SCRATCHPAD_BLOCK;
}SCRATCHPAD_MEMORY
typedef struct MAIN_MEMORY
{Main Memory Structure
int DATASET_BASE_ADDRESS;
int DATASET_WIDTH;
int DATASET_HEIGHT;
int DATASET_DIMENSION;
}PAMS_DATASET;
//Main Program
SCRATCHPAD_MEMORY SM0;
PAMS_DATASET DS0;
// 3D Scratchpad Memory (32x32x64)
SM0.SCRATCHPAD_BASE_ADDRESS=0x00000000;
SM0.SCRATCHPAD_WIDTH=32;
SM0.SCRATCHPAD_HEIGHT=32;
SM0.SCRATCHPAD_BLOCK=64;
// Main Memory (Data Set) 128x128x128
DS0.DATASET_BASE_ADDRESS=0x30000000;
DS0.DATASET_WIDTH=128;
DS0.DATASET_HEIGHT=128;
DS0.DATASET_DIMENSION=128;
```

Fig. 2. *PAMS* : (a) Scratchpad and *Main Memory* Structure & Initialization (b) Run-Time Address Manager (c) Pattern Aware Main Memory Controller

The *PAMS* organizes *static* and *dynamic* data structures requests in two types of descriptors: the *regular* and the *irregular descriptors*. The *regular descriptor memory* [18] holds information of *static data structures* (1D/2D/3D arrays) and their access patterns. The *static data structures* are aligned in terms of memory addressing, managing data structures that have compile-time predictable and aligned data accesses. The *irregular descriptor memory* [3] holds information about unaligned *dynamic data structures* (tree-based) and their access patterns, where the descriptor memory is allocated during run-time. The size of the allocated memory can vary between executions of the program. The *PAMS descriptor memory* has a number of attribute fields. The set of parameters for a *descriptor memory* block includes, the *Local Address*, the *Main Address*, *Priority*, *Size*, *Stride* and *Offset*. The *Address* parameters hold the base addresses of *Scratchpad Memory* and *Main Memory*. The *Priority* defines the order by which a memory access pattern is entitled to be processed. The parameters *Size* and *Stride* define the types of memory access. The *Offset* register field is used to point to the next linked memory access pattern.

The *PAMS descriptor memory* aims to identify the *static* and *dynamic data structures*. It presents data transfers in the form of a descriptor that tailors variables of random locations. The *PAMS* categorises access patterns into three sections:

a) Compile-time predictable: These accesses are managed in descriptor memory before execution. At run-time, *PAMS* reads the descriptor information, tailors it in pattern descriptors before execution and transfers the access pattern to the *Scratchpad Memory*.

b) 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 III-C).

c) Run-time unpredictable: For access patterns with run-time unpredictable accesses, *PAMS* uses separate control (*select & ready*) signals to communicate with the processing core, that generates address for next access pattern.

C. Memory Manager

The *PAMS Memory Manager* uses descriptor history table that keeps the knowledge of memory and whether a certain memory area is in the *Scratchpad Memory*. The history table allows the *PAMS* to reuse the already accessed memory and to share it between applications. As the data access pattern size and stride are arranged in *regular* and *irregular descriptors*, the *PAMS* efficiently utilizes these access patterns at runtime. The *PAMS Memory Manager* supports programmed and automatic scheduling policies. These 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. It also schedules memory requests depending on the access pattern, transfer size and program priorities. The *Memory Manager* is further divided into two sections which are: the *Address* and the *Data* Manager (shown in Figure 1).

1) Address Manager: The *Address Manager* uses one or multiple descriptors at run-time to describe the data access. Unlike the cache, which transfers an aligned block of data for each data miss, the *PAMS Address Manager* accesses only the missed data by gathering address requests at run-time and transfers irregular blocks of data. The *Address Manager* also manages run-time unpredictable memory accesses and places them in *descriptor memory*. The *Address Manager* takes memory address requests from a *Processor Core*, buffers them and compares the consecutive requesting addresses with the previous one. If the addresses of consecutive memory requests have constant strides, the *Address Manager* allocates a *descriptor* block by defining *Stride* and *Size* parameters. If the request has variable strides, then the *Address Manager* uses the *Offset* parameter of the *descriptor* that points at the random location of the *Main Memory*. The structure of run-time *Address Manager* is shown in Figure 2(b). The *Address Manager Stride Detector* takes an address from the address bus and gives it to *reg 0* and *comparator 0*. The *comparator 0* compares current address value ($Address_t$) with the previous one ($Address_{t-1}$) and generates the stride which is given to *Pattern Controller*. The *Pattern Controller* compares two strides ($Stride_t$ and $Stride_{t-1}$) and checks if they are same. If they are same then it increment in *Size* register of *descriptor memory* 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) Data Manager: The *Data Manager* [15] takes *Scratchpad Memory* information from the *PAMS* descriptor memory and manages the *Register File* data. The *Data Manager* uses the *Register File* to arrange and align the data patterns and saves read data information for further reuse. It loads/updates patterns from the *Main Memory* which are not present in *Scratchpad Memory*. The *Data Manager* improves the *Computational Intensity* (the number of operations per byte accessed from the memory) by organizing and managing the memory accesses. For a core processing a single *computed point*, the maximum achievable (ideal) *Computational Intensity* is 1. The *Data Manager* accesses the data elements in the form of patterns which are required for a single output ($Computed_{element}$). After accessing the first access pattern, the *Data Manager* reuses and updates data where required. The *Data Manager* reuses elements that are available in the reuse register and updates the accesses (*elements*) required for the processing core.

D. Pattern Aware Main Memory Controller (PAMMC)

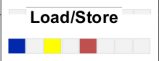
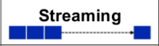
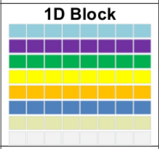
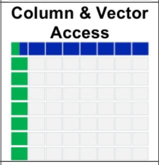
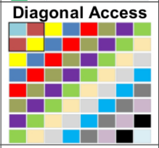
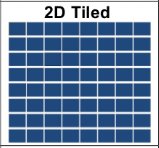
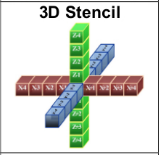
Unlike a conventional *Main Memory* controller, the *PAMMC* (Figure 2(c)) uses descriptors to access data. At run-time *PAMMC* takes descriptors for an access pattern from the *Memory Manager*. The *PAMMC* deals with the *Address*, *Stream* and *Stride* registers of pattern descriptors and translates patterns into main memory addresses. Each address is categorized into *Bank*, *Row* and *Column* address of SDRAM. The *PAMMC* supports two possible modes of operations for bank management: the single-bank mode and the multi-bank mode to parallelize data accesses. In the single bank mode, the controller keeps one bank and row combination open at any given time. In the multi-bank mode, the controller keeps multiple banks open at any given time. This mode is used when the data access patterns of an application require data from different banks at the same time. The *PAMMC Bank Manager* is integrated in the design to reduce the memory access time and power by managing either the single- or the multi-bank mode according to the memory access pattern descriptions. The multi-bank mode is used for complex data patterns having long strides that access data from multiple banks in parallel. 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 bursts and keeps the same bank open and, if required, keeps contiguous rows precharged. These types of access patterns are very common in the *PAMS* system because access patterns are organised in descriptors and occurs for around 70% of all memory patterns for the studied benchmarks. Depending on the SDRAM banks, the *PAMMC* processes multiple patterns in parallel, each accessing a single bank.

IV. Experimental Framework

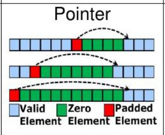
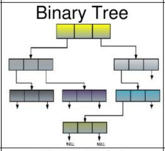
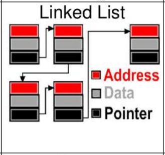
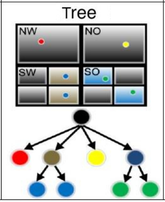
In this section, we describe and evaluate the *PAMS* and compare with a *Baseline Memory System* (*BMS*). The *BMS* uses cache, scratchpad and buffer memories. The Xilinx Platform Studio 14.3 and Integrated Software Environment 14.3 are used to design the memory systems on the Virtex-7 FPGA VC707 Evaluation Kit. The power analysis is done by Xilinx Power Estimator. The section is divided into three subsections: the *Processing Cores*, the *Baseline Memory System* and the *Pattern Aware Memory System*.

A. Processing Cores

Table 3 shows the application kernels that are used in the design. Column `Access Pattern` presents type of access patterns of the application kernels. Each color represents a separate memory transfer pattern. Figure 3(a) has compile-time predictable access patterns with *static data structure*. Column `Descriptor` shows the number of descriptors requires to access a data structure for the computation. Figure 3(b) shows applications having access patterns for *dynamic data structure*. Column `Com_Pre %` presents percentage of compile-time predictable access patterns. Column `Run-time %` shows percentage of run-time predicate and un-predictable access patterns. Application Specific Hardware Accelerator (*ASHA*) and Microblaze cores are used to process the application kernels. MicroBlaze is a 32-bit soft-core processor [19] and features a Harvard RISC architecture, 32-bit instructions, a 3-stage pipeline, a 32 register wide register file, a shift unit and two

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

(a)

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

(b)

Fig. 3. Application Kernels: (a) Static data structures (b) Dynamic Data Strictures

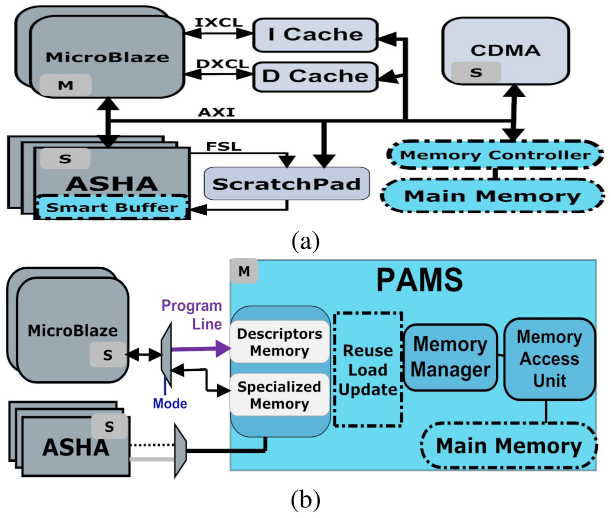


Fig. 4. System Architecture: (a) BMS (b) PAMS

levels of interrupt. The *ASHA* is generated by ROCCC [20], a tool that creates streaming hardware accelerators from a subset of C. The ROCCC generates the smart buffers for computationally intensive window operations. The hardware generated by ROCCC speeds up applications by replacing critical regions in software with a dedicated hardware component. These dedicated accelerators have low footprint and low power consumption and provide high performance. The applications having dynamic data structures with irregular accesses are executed on the MicroBlaze processor, and the application kernels with *static data structure*s having regular accesses are executed on *ASHA*. In our current designs, applications are executed at 100 MHz and the highest bandwidth required 400 MB for computational intensity=1 (i.e. the number of operations per word accessed from the memory). The *Main Memory* is shared between all application kernel.

B. Baseline Memory System (BMS)

A Xilinx FPGA based state of the art memory system is used as a *BMS* shown in Figure 4(a). Like a conventional memory system, the *BMS* uses a microprocessor that manages and handles static and dynamic data structures. The memory system takes instructions from the MicroBlaze processor and performs memory management and data transfer operations. The MicroBlaze core of *BMS* uses Instruction Cache Link (IXCL) and Data Cache Link (DXCL) to access instruction Cache and Data Cache memory respectively. Fast Simplex Link (FSL) is used to feed data to *ASHA*. Each FSL has one input and one output port and provides a low latency dedicated interface to the *ASHA*. A High Performance Multi-Port Memory Controller (MPMS) is used to provide an efficient interfacing between the Microblaze processor and SDRAM using the Advanced eXtensible Interface (AXI) bus. A modular DDR3 SDRAM controller (with an AXI4 Wrapper) is used with the MPMS system to access the *Main Memory*. The *BMS* uses Central Direct Memory Access (CDMA) controller to improve the performance of the SDRAM controller by managing complex patterns in hardware. The CDMA allows full-duplex, high-bandwidth bus interfaces into memory.

To manage on-chip data for computationally intensive window (loop) operations, the *ASHA* uses smart buffers in *BMS*. The smart buffer helps to minimize the accesses to the *Main Memory* 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 having regular access patterns through loop transformation-based program restructuring. This store the input and processed data for future iterations and remove the old data if it is not required in future.

C. PAMS based System

The *PAMS* based system is shown in Figure 4(b). The major difference between *PAMS* and the *BMS* is that the baseline system uses a MicroBlaze microprocessor, which manages the local and scratchpad memory data and transfers data between the SDRAM controller and the *Processing Cores* using the cache hierarchy. The MicroBlaze processor works as slave in the *PAMS* system. The *PAMS* uses *Scratchpad Memory* to manage *static* and *dynamic* data structures, a *Data Manager* that organises and reuses, and an *Address Manager* that manages the data transfers at compile-time and run-time for the *Processing Cores*.

V. Results and Discussion

This section analyzes the results of different experiments conducted on *PAMS* and *BMS*. The experiments are characterized into five subsections: *Resource Utilization*, *Application Performance*, *Program Memory*, *Throughput* and *Power Consumption*.

A. Resource Utilization

In this section we measure the resource utilization of the memory hierarchy of the *BMS* and *PAMS*. The memory hierarchies of *BMS* and *PAMS* are compiled for cache/scratchpad memories and scratchpad memory respectively. The *BMS* smart buffers and the *PAMS* register file size is dependent on the number and the size of *ASHA* nested loops.

Table I shows the resource utilization of the memory hierarchy of the *BMS* and the *PAMS*. The memory hierarchy of the baseline system is compiled for a 64KB of data cache and 64KB of scratchpad. The *PAMS* is compiled for 128KB of *Scratchpad Memory*. The *Local Memory* column presents area of usage the data-path and control unit and BRAM usage in bits of *cache/Scratchpad* and *Scratchpad Memory* of *BMS* and *PAMS* respectively. The column (*Reg, LUT*) of *Local Memory* shows the resources used by the cache hierarchy and scratchpad controller of the *BMS* and the *Memory Manager* of the *PAMS*. The *BRAM* presents the number of Block RAMs (36k bitss). Table I shows that the *Memory Manager* of the *PAMS* memory system occupies 2 times less resources (*reg and LUT*) than the *BMS* scratchpad and cache management. The *Main Memory* column presents the resource utilization for the SDRAM DDR3 controller.

Table II shows *ASHA buffer memory* and *register file* that are used to fetch and store data for *BMS* and *PAMS* ASHA respectively. The *ASHA* of *BMS* uses ROCCC based *Smart Buffers* that uses the data between the loop iterations, which consist of registers that cache the portion of memory used. The *# loops* presents the number of nested loops that are used to perform computations on large blocks of data placed in local scratchpad memory. The *window size* column shows the amount of data that has to be dispatched to the *ASHA* datapath per clock cycle. Results show that the impact of loop transformations on the size of generated *Smart Buffers* hardware is high. i.e. generated hardware to support loop transformation

TABLE I. Resource Utilization of the Memory Hierarchy

	Local Memory				Main Memory	
		Reg	LUT	BRAMs (36kb)	Reg	LUT
BMS	Cache	502	1188	19	5678	7589
	Scratchpad	402	533	16		
PAMS	Scratchpad	190	1030	32	4742	6249

TABLE II. RESOURCE UTILIZATION OF SMART BUFFER AND REGISTER FILE

ASHA	# of Loops	Window	Reg	LUT
BMS	1D (FIR) 1024	128	12203	4045
	2D Filter (128x128)	8x8	1833	555
		16x16	3306	1036
		32x32	6122	2028
	3D Stencil (128x128x128)	8x8x8	26364	43296
		16x16x16	Failed	
		32x32x32	Failed	
PAMS	1D (FIR) 1024	1024	4100	3912
	2D Filter (128x128)	16x16	534	1018
		32x32	2084	1958
	3D Stencil (128x128x128)	8x8x8	10861	12323
		16x16x16	42861	30323
		32x32x32	112861	82323

consumes more FPGA logic. For large and complex window sizes Xilinx ISE failed to synthesize hardware for *Smart Buffers* due to complex loop transformations. The *BMS ASHA* hardware register size increases when the loop is pipelined, which holds a large window of input data, however the control logic cost remains the same. The *PAMS* uses the *Register File* which is handled by the *Memory Manager*. The *PAMS* itself performs off-chip and on-chip data management and supports complex and irregular memory accesses. Table II shows that *PAMS* supports complex (i.e. three-dimension, pointer, etc.) memory accesses and uses less hardware resources then the *BMS*. The *PAMS* access patterns are rearranged by re-programming the *PAMS* descriptor memory, which does not require the re-synthesis of hardware accelerators. The *BMS ASHA* uses *Smart Buffers* to manage on-chip data, but in order to read data from the *Main Memory* it requires a MicroBlaze processor core.

B. Program Memory

The program memory allocates memory space for the retrieval and the manipulation of memory assignments. The Xilinx library generator (libgen) is used to generate libraries and header files necessary to build an executable file for application kernel. Libgen parses the system hardware drivers, interrupt handling, etc. and creates libraries for the system. The libraries are then used by the MicroBlaze GCC compiler to link the program code for the processing cores. The object files from the application and the software platform are linked together to generate the final executable object file. In this section we compare the executable object files of the *PAMS* and *BMS* for each kernel.

Figure 5 presents the memory used by the *BMS* and the *PAMS*. The y axis is in logarithmic scale. In the *PAMS*, the *regular descriptor memory* holds the local variables and the descriptor information for static access patterns. In the *BMS* 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 6 times less space in the *regular descriptor memory* compared to the *BMS*. The FIR application has a streaming data pattern, which

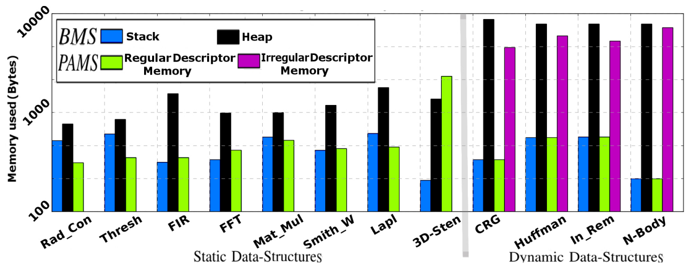


Fig. 5. Program Memory: Static and Dynamic Data Structures

uses 6 times less memory. The FFT and Mat_Mul/Lapl/Smith-W kernels access 1D and 2D patterns respectively. These applications use 4, 3, 5, and 9 times less space in the *regular descriptor memory* respectively. The 3D-Sten kernel accesses 3D data patterns having 32x32x32 dimensions. Due to the small dimensions of the 3D-stencil data set, the *PAMS* uses two times more regular memory than the *BMS*. However across results show as a rule of thumb, the *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 *BMS* for applications having dynamic data structures and compared it with the *PAMS irregular descriptor memory*. For CRG, Huffman, In_Rem and N-Body, the *PAMS* system requires 1.13x to 2.66x less *irregular descriptor memory*, compared to the baseline system. The *stack* and *regular descriptor memory* size is the same in both systems.

C. Application Performance

In this section we compare the performance of *BMS* and *PAMS*. Figure 6, shows the number of clock cycles taken by an application kernel to manage and process the data set. Each application data set is placed in *Main Memory*. In Figure 6 the *BMS Data Access* presents main memory data access and cahce/scratchpad memory management time. The *BMS* uses a MicroBlaze processor, DMA and Memory Controller to transfer data between the *Main Memory* and Cache/Scratchpad memory. The *BMS ASHA* bar shows the number of clock cycles to process the data available on scratchpad memory. The *PAMS* uses *Scratchpad Memory* to feed data to the *MicroBlaze* and *ASHA*. The *PAMS Data Access* column presents the time to access data patterns from main memory and time to manage it on *Scratchpad Memory*. The *PAMS ASHA* column shows the number of clock cycles required to process the data available on the *Scratchpad Memory*.

By using the *PAMS* system, the results show the *Rad_Con* and *Thresh* applications achieve 3.3x of speedup over the *BMS*. 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 25x of speedup. The *FFT* application kernel reads a 1D block of data, processes it and writes it back to main memory. This application achieves 10.2x of speedup. The *Mat_Mul* kernel accesses row and column vectors. The application attains 14x of speedup. The *PAMS* system manages addresses of row and column vectors in hardware. The *Smith_W* and *Lapl* applications take 2D blocks of data and achieve 38x and 36x of speedup respectively. The *3D-Stencil* data decomposition achieves 52x of speedup. The *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 baseline system 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 MicroBlaze 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 on the PAMS, the system achieves 1.5x, 1.7x of speedup respectively over the BMS. The In_Rem application has run-time predictable memory access patterns with no data locality, hence achieves 2.0x of speedup. The N-Body application includes predictable data patterns with data locality. While running on the PAMS system, it achieves 2.9x of speedup over the BMS system.

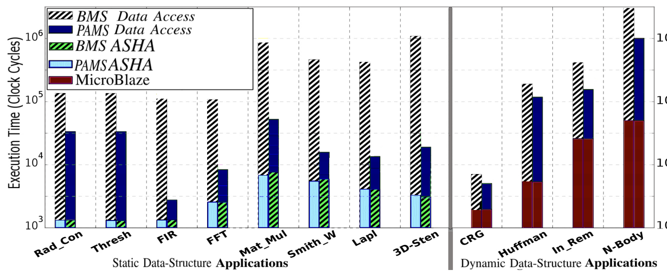


Fig. 6. Application Performance: Static and Dynamic Data Structures

D. Throughput

In this section, we measure the data transfer throughput of *PAMS* and *BMS* for a different number of *ASHA* cores by reading and writing data set with two types of transfers. The X-axis (shown in Figure 7) presents two types of data transfers and number of cores. Each data transfer reads and writes a data set of 2MB from/to the SDRAM. The type *Short Window* contains data transfers that have a maximum transfer size of 128B and the type *Long Window* has a transfer size of 4KB. Therefore a single *ASHA* has 32768 and 1024 read-after-write requests of *Short Transfer*s and *Long Transfer*s respectively. The requests increase with the number of requesting *ASHA*s. While using 1, 2, 4 and 8 requesting *ASHA*s for *Short Transfer* type, results show that the *PAMS* transfers data 3.80x, 4.338x, 4.48x and 4.65x times faster respectively than the *BMS*. While transferring data with the *Long Window* type, the *PAMS* improves throughput 2.79x, 2.90x, 3.08x and 3.10x times. Results show that the *PAMS* improves the aggregated throughput for *Short Window*s when increasing the number of cores. The *BMS* uses the CDMA controller that forces to follow the bus protocol and requires a processor that provides data transfer instructions. The CDMA 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 *BMS* uses multiple instructions to initialize CDMA. CDMA can begin a new transfer before the previous data transfer completes with a delay called pipeline latency. The pipeline latency increases with the number of data transfers. Each Data Transfer requires bus arbitration, address generation and SDRAM bank/row management. The *PAMS 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.

E. Power Consumption

On-chip power dissipation in a Xilinx Virtex-7 XC7VX485T device is 2.95 Watts while running the *BMS* using 64KB cache and 64KB scratchpad memories. The *PAMS* system with 128KB *Scratchpad Memory* draws 2.36 Watts of on-chip power. The *PAMS* consumes 20% less

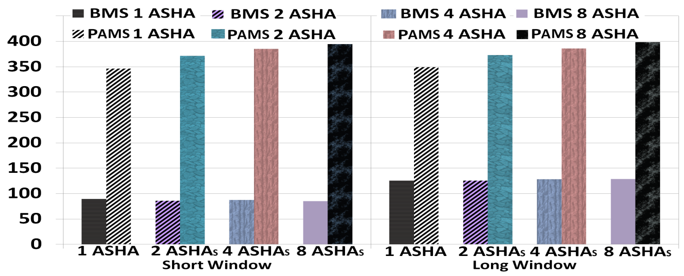


Fig. 7. BMS and PAMS: Memory Throughput

on-chip power than the *BMS*. The *PAMS* provides low-power and simple control characteristics by rearranging data accesses and utilizing hardware units efficiently.

VI. Conclusion

High performance applications suffer from poor performance on FPGA architectures due to the memory system. In this work, we proposed an efficient and intelligent memory system in hardware called Pattern Aware Memory System (PAMS). The proposed memory system manages static and dynamic data structures in hardware and controls their memory accesses in the form of patterns that improve system data transfer throughput. In order to prove that our memory system is efficient in a variety of scenarios, we used several benchmarks with different data structures and memory access patterns. The benchmarking results show that the *PAMS* transfers memory patterns up to 4.65x faster, achieves between 3.3x to 52x and 1.5x to 2.9x of speedup for applications having static and dynamic data structures respectively and consumes up-to 9 times less program memory.

References

[1] Matthias Hartmann et al. Memristor-based (reram) data memory architecture in asip design. In *Digital System Design (DSD), 2013*.

[2] Stylianos Perissakis et al. Embedded dram for a reconfigurable array. In *VLSI Circuits, 1999. Digest of Technical Papers*.

[3] T. Hussain et al. Advanced pattern based memory controller for fpga based applications. In *International Conference on HPCS*, 2014.

[4] Rajeshwari Banakar et al. Scratchpad memory: Design alternative for cache on-chip memory in embedded systems. In *Proceedings of the tenth international symposium on Hardware/software codesign*.

[5] Zhang, Chuanjun et al. Using a victim buffer in an application-specific memory hierarchy. In *DATE2004*.

[6] Ken Mai and Paaske et al. Smart memories: A modular reconfigurable architecture. 2000.

[7] Stefan Steinke et al. Reducing energy consumption by dynamic copying of instructions onto onchip memory. In *15th International Symposium on System Synthesis, 2002.*

[8] Preeti Ranjan Panda et al. *Memory issues in embedded systems-on-chip: optimizations and exploration*. Springer, 1999.

[9] Vivy Suhendra et al. Wcet centric data allocation to scratchpad memory. In *26th IEEE International Symposium on Real-Time Systems, 2005.*

[10] J-F Deverge and Isabelle Puaut. Wcet-directed dynamic scratchpad memory allocation of data. In *Real-Time Systems, 2007. ECRTS'07. 19th Euromicro Conference on*.

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

[12] Gabriel Weisz et al. C-to-coram: compiling perfect loop nests to the portable coram abstraction. In *FPGAs*, 2013.

[13] Louis-Noël Pouchet et al. Polyhedral-based data reuse optimization for configurable computing. In *International symposium on FPGAs*, 2013.

[14] T. Hussain et al. Recongurable Memory Controller with Programmable Pattern Support. *HiPEAC WRC*, Jan, 2011.

[15] T. Hussain et al. Memory controller for vector processor. In *The 25th IEEE ASAP 2014 Conference*.

[16] T. Hussain et al. Stand-alone memory controller for graphics system. In *The 10th International Symposium on ARC*. ACM, 2014.

[17] Tassadaq Hussain and Amna Haider. Pgc: a pattern-based graphics controller. *Int. J. Circuits and Architecture Design*, 2014.

[18] T. Hussain et al. PPMC: A Programmable Pattern based Memory Controller. In *ARC 2012*.

[19] Embedded Development Kit EDK 10.1i. *MicroBlaze Processor Reference Guide*.

[20] Riverside Optimizing Compiler for Configurable Computing (ROCCC 2.0), 3,April 2011.