Memory-based Computing for Performance and Energy Improvement in Multicore Architectures

Kamran Rahmani
University of Florida
kamran@cise.ufl.com

Prabhat Mishra
University of Florida
prabhat@cise.ufl.edu

Swarup Bhunia
Case Western Reserve Univ.
sxb21@case.edu

ABSTRACT

Memory-based computing (MBC) is promising for improving performance and energy efficiency in both data- and compute-intensive applications. In this paper, we propose a novel reconfigurable MBC framework for multicore architectures where each core uses caches for computation using Look Up Tables (LUTs). Experimental results demonstrate that on-demand memory-based computing in each core can significantly improve performance (up to 4.7X, 3.3X on average) as well as reduce energy consumption (up to 4.7X, 2X on average) in multicore architectures.

Categories and Subject Descriptors

C.1 [**PROCESSOR ARCHITECTURES**]: Adaptable architectures

General Terms

Design, Performance

Keywords

Multicore systems, memory-based computing, acceleration, energy optimization

1. INTRODUCTION

There are two common reconfigurable computing categories, FPGA [1], and CGRA [2]. FPGA suffers from poor technological scalability of performance and CGRA suffers from lack of flexibility to map diverse applications. In addition, both technologies fail to improve performance and energy efficiency in case of data-intensive applications. MBC is a promising alternative to address the above challenges. For ease of comparison, different features of all approaches are summarized in Table 1.

A promising solution to the challenges above is to use a reconfigurable memory-based computing (RMBC) framework. A RMBC framework uses dense 2-D memory arrays for computation by configuring them as LUTs to hold the configuration for the mapped application. RMBC addresses above challenges by making a trade-off between the flexibility and granularity. It is beneficial due to following reasons: i) it is amenable to efficient functional decomposition of complex operations (e.g. square root, trigonometric functions etc.) with fewer inputs (one operand often being constant coefficient) which are easier to implement as lookup table (LUT) than logic block; ii) they typically have larger than 1-bit pathwidth [3] which is amenable for sparse interconnect structure; and iii) most applications have high temporal locality in data that can be exploited to reduce computation overhead. In addition, RMBC leverages on traditional on-die cache memory which leads to minimum changes in hardware architecture. Necessary circuit and architecture level modifications however need to be incorporated into the conventional memory array before it can be used as a reconfigurable framework. This logic-memory duality benefits memory-intensive applications in addition to compute-intensive ones. In this paper we propose an architecture that uses RMBC in multicore architectures where each core has a tightly-coupled RMBC unit.

Table 1: Comparison of reconfigurable frameworks

Feature	FPGA	CGRA	RMBC
Flexibility (compute-intensive)	High	Moderate	High
Flexibility (data-intensive)	Poor	Poor	High
Energy Efficiency	Poor	Moderate	High
Resource Utilization	Moderate	Moderate	High
Reconfiguration Latency	Moderate	Low	Moderate
Technology Scalability	Poor	Moderate	High

Paul and Bhunia proposed a novel MBC platform in [4]. This framework is however, primarily optimized for fine-grained combinatorial logic. In this paper, we optimize the same for coarse-grained regular data path, common to algorithmic tasks. The only work on MBC in a multicore architecture is done by Hajimiri et al. [5]. They proposed an architecture for improving reliability in multicore architectures using MBC. However, this improvement in reliability degrades performance. In this paper, our focus is on using MBC as an accelerator in a multicore architecture.

GLSVLSI'12, May 3–4, 2012, Salt Lake City, Utah, USA.

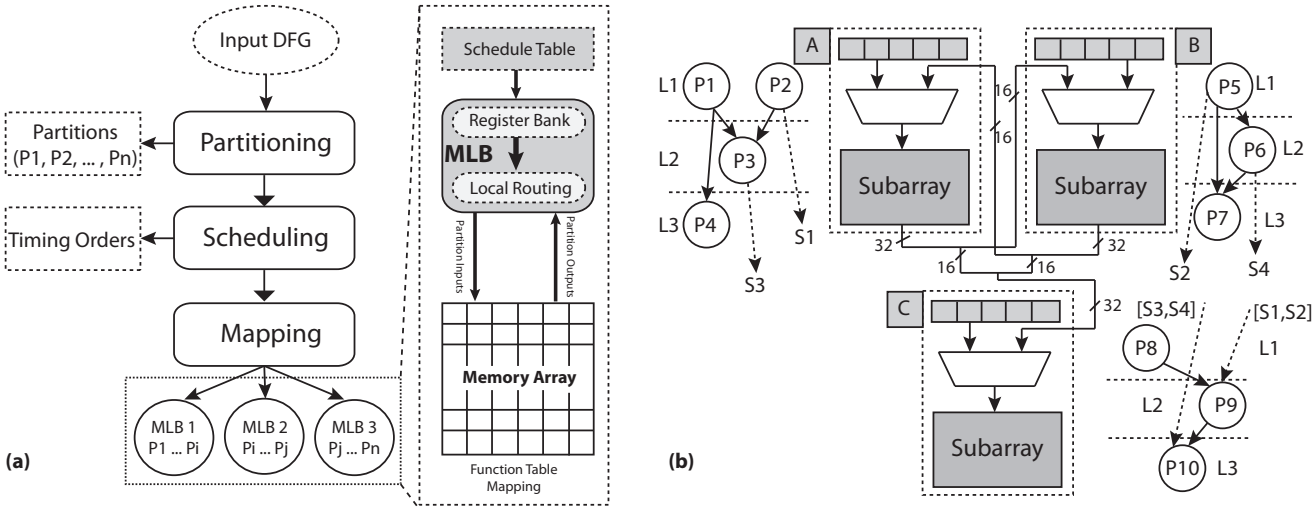

Figure 1: a) Functional block diagram for memory based computing b) Synchronization across multiple MLBs

2. AN OVERVIEW OF MBC

Figure 1(a) shows the functional block diagram of MBC model proposed in [4, 6]. The target application is partitioned into multi-input multi-output partitions, which are then mapped to multiple computing elements, each computing unit being referred to as a Memory Logic Block (MLB). Inside each MLB, the partitions are evaluated over multiple clock cycles in a topological manner. Intermediate partition outputs are stored in the register bank. LUTs are mapped to a dense 2-D memory array which is otherwise referred to as the function table. Schedule Table stores the microcode which determines the sequence of operations inside each computing element. A fast local routing network is used to select the intermediate partition outputs from the register bank. Figure 1(b) shows the communication and synchronization among multiple MLBs through a time-multiplexed usage of the local and global interconnect in the MBC model. As shown in Figure 1(b), S1 and S2 are outputs of MLB A and B at the end of cycle 1, while S3 and S4 are outputs at the end of cycle 2. Signals at the end of each cycle are transmitted over the same local/global channel to MLB C. Furthermore, intra-MLB cycle time is determined by the time taken to read the operands from the intermediate registers and the time to read the LUTs mapped to the subarray inside each MLB. Inter-MLB cycle time is determined by the LUT read time inside one MLB and its output to reach the inputs of LUTs in other MLBs.

3. MBC IN MULTICORE SYSTEMS

This section presents our RMBC framework for multicore architectures. The framework which normally acts as a computing resource can on-demand be transformed into data memory. Using memory for computation gives the dual benefit that the same can be used as normal storage for memory-intensive applications. Consequently, by customizing itself to different application requirements, it can provide speed-up for a wide range of compute and data-intensive applications. RMBC framework operates as a Reconfigurable Functional Unit (RFU) in a conventional RISC pipeline. Consequently, it is important to consider the modifications of both

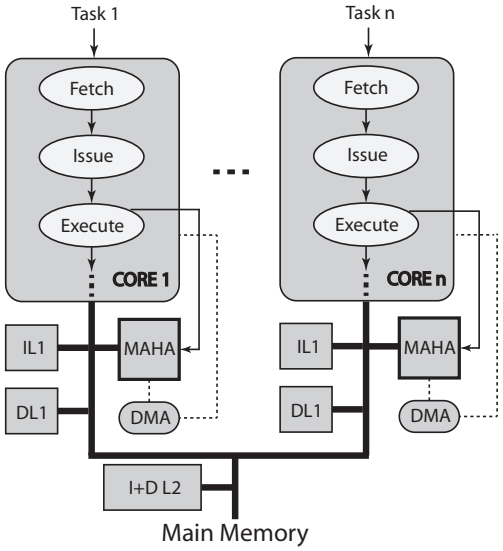

Figure 2: RMBC in a Multicore Architecture

hardware as well as software in the system. This modifications should enable the multicore architecture to on-demand execute a set of applications in the RMBC units. Figure 2 shows the proposed integration scheme of RMBC into a multicore architecture. In this architecture, each core has an independent RMBC unit that can be on-demand reconfigured in order to fulfill the application's requirements. Required modifications are discussed in the following sections.

3.0.1 Hardware Architecture

Constructing an RFU around the normal cache architecture imparts the exceptional advantage that the RMBC architecture can be interfaced with the main memory as well as with the processor register file for transfer of working dataset and configuration data. Since on-chip cache is already provisioned with these interfaces, the design overhead involved is minimal. However, for data-parallel applications for which large volume of data need to be loaded onto the RFU, large number of cycles will be wasted by the processor mediating between the cache and the main memory. The proposed

framework therefore utilizes a DMA interface between the RFU and the main memory similar to the scheme proposed in [7]. A data buffer is used to overlap computation in the RFU with loading/unloading of data from the main memory through the DMA controller. It is to be noted here that the *context memory* as described in [7] is virtually distributed as schedule table inside individual MLBs of the RMBC framework. This opens up the opportunity that sets of MLBs can operate and be reconfigured independently. Thus normal function in one set of MLBs proceed with the reconfiguration in others, effectively hiding the reconfiguration latency. Following are the sequence of actions taken by each core to initiate computation in the RMBC framework.

Write-back dirty blocks: Since the storage which serves as an RFU on-demand is otherwise part of the on-chip cache, dirty lines present in this storage need to be written back to the main memory before any RFU operation can proceed. This latency can be minimized by periodic cleaning of the dirty blocks.

Load reconfiguration data: RMBC exploits the large data bandwidth of memory interface to load the reconfiguration data into the MLBs in groups of 128 bits. In the worst case, when the entire 16KB memory inside an MLB requires reconfiguration, the total number of cycles required to configure a single MLB is only 1024. Additional 8 cycles are required to configure the schedule table. Programming the PI architecture however cannot be performed in parallel and requires 128 cycles to program the external connection for the 32-bit input to each MLB. The configuration data is brought into the RMBC framework from the main memory using the DMA controller. The core initiates this transfer by providing the starting address of reconfiguration data to the DMA controller.

Load working data: For loading the working data set, the DMA controller is supplied with the starting address by the core. Data from the main memory is first moved onto the data buffer which is then transferred over to selected MLBs. Unlike Morphosys, inputs for the application mapped to the RMBC framework also arrive at selected MLBs from the processor register file via normal load/store instructions. This helps to speed-up crypto applications and other combinatorial functions where data-parallelism is not inherent. In order to hide the latency of data arrival, the DMA controller is programmed to overlap data pre-fetch from the main memory with normal RFU operation.

3.0.2 Software Architecture

Integration of an RFU into a dynamic pipeline requires extension of the instruction set to incorporate the new RFU opcodes. As illustrated in Figure 3, such an extension is achieved for an Portable Instruction Set Architecture (PISA) through identification of the application and modification of the software toolflow to handle the RFU opcodes at different stages of compilation, and linking. The first step in the methodology is the identification of the hotspot or computational kernels where the code spends major portion of the execution time. The assembly instructions corresponding to these hotspots are determined from the disassembled binary. The subsequent step identifies the section of the C-code corresponding to the assembly instructions. A new library containing pseudo-assembler instructions is defined for the RFU functions using GCC inline assembler macro feature. This library is then used to compile the C-code.

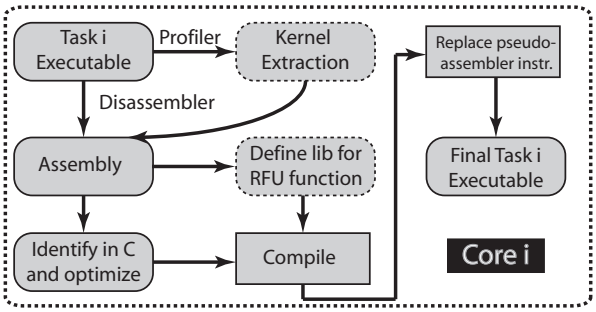

Figure 3: Software architecture for automatic translation of application kernel to RFU instructions

A post-processing on the compiled executable replaces the pseudo-assembler instruction with the correct machine opcode. This approach avoids modifying the C-compiler to take into account new opcodes corresponding to the RFU instructions. The correct machine opcode used to replace the pseudo-assembler instruction utilizes the unused opcode space of the host processor to define new RFU instructions.

4. EXPERIMENTS

4.1 Experimental Setup

To check the effectiveness of our proposed scheme, we used a widely used multicore simulator, M5 [8]. We enhanced M5 to make the required modifications in processor cores to support on-demand computation transfer to MBC. We configured the simulated system with a two-core, three-core, and four-core processor each of which runs at 500MHz. The DerivO3CPU model [8] in M5 is used which represents a detailed model of an out-of-order SMT-capable CPU which stalls during cache accesses and memory response handling. In addition, for MBC part we implemented a mapper using C that gets the critical-section of the application (kernel) DFG as input and provides the detailed MBC mapping information including required number of cycles and energy consumption per vector. The effectiveness of our proposed framework was investigated and validated for two different scenarios: i) improving the performance, and ii) improving memory sub-system energy consumption. For the second part, we applied the same energy model used in [9], which calculates both dynamic and static energy consumption, memory latency, CPU stall energy, and main memory fetch energy. We updated the dynamic energy consumption for each cache configuration using CACTI 6.0. We developed a Perl script that gets M5 simulation output as input and provides total consumed energy in memory hierarchy including all private instruction L1 and data L1 (for each core) and shared L2 caches in the multicore architecture.

In order to investigate our proposed framework, we formed different task sets to be executed together in a multicore environment. The tasks are independent and each task is assigned to a core. These tasks are collected from both compute- and memory-intensive applications. In the 2-core framework, the tasks are organized as *set1 (sha, atr), set2 (aes, ci), set3 (me, dwt), and set4 (census, dct)*. Similarly, for 3-core scenario they are organized as *set5 (aes, sha, me)*, and *set6 (ci, atr, dwt)*. Likewise, we formed *set7 (aes, sha, me, atr)* for 4-core scenario. We simulated these scenarios and investigated the effect of using RMBC on performance as well as memory sub-system energy consumption.

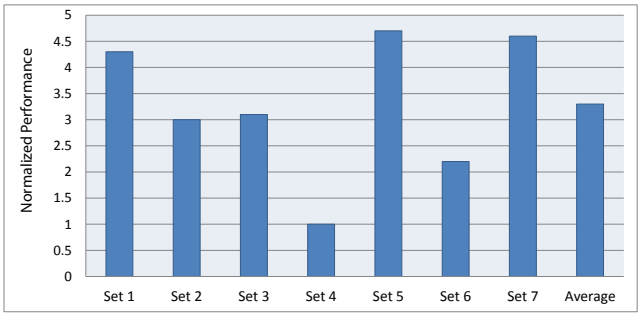

Figure 4: Performance of the multicore+RMBC normalized to the multicore only scenario

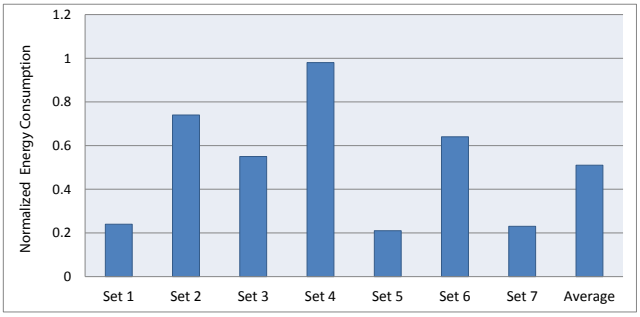

Figure 5: Energy consumption in multicore+RMBC normalized to the multicore only scenario

4.2 Performance Analysis

The metric that we used for performance comparison is the average of normalized performance improvements (cycle number) in all cores. Figure 4 shows the performance improvement in an RMBC-enabled multicore system normalized to the multicore system without RMBC. In most cases we have drastic improvements in performance. However as can be seen, this improvement depends on applications and their nature. For example in set1 we have 4.3X improvement in performance while it is 1.02X for set4. This variation shows that although RMBC is beneficial in both computation and memory intensive applications, some applications are not a good-fit when they are mapped to RMBC. Nonetheless, as it can be observed in most applications RMBC is a promising framework to improve the performance in multicore systems. In addition, it can be observed that generally the performance improvement increases as the number of cores increases. This is because of sharing resources, doing computation in RMBC in the cores releases some shared recourses (L2 and shared bus). In other words, while one or more cores are doing computation in RMBC, it is not using shared resources. This can make more resources available for other cores that are using it (doing computation in CPU); more resources lead to more performance for these cores. In summary, for our application sets the performance improvement is up to 4.7X for set5 and is 3.3X on average.

4.3 Energy Consumption

In order to investigate the effect of using RMBC on energy consumption, we compared the total energy consumption of memory sub-system in our proposed multicore framework. This metric comprises of total energy consumption in all caches plus RMBC units for RMBC-enabled system normalized to energy consumption in all caches for traditional multicore scenario. Figure 5 illustrates this metric for the same application sets discussed before. Like performance improvement, in most cases we have drastic reduction in energy consumption. As expected, in most cases there is a strong relation between improvement in performance and reduction in energy consumption. For example set1 that has 4X improvement in performance has 4X reduction in energy. As it can be observed, enabling RMBC in multicore systems is a promising approach to reduce energy consumption. For our application sets the energy reduction is up to 4.7X for set5 and is 2X on average.

5. CONCLUSIONS

We have presented RMBC, an embedded memory-based computing framework, which can serve as a reconfigurable functional unit in multicore systems to provide hardware acceleration. The framework leverages on the high integration density offered by modern embedded memory and drastically reduces the requirements of programmable interconnects which impose major bottlenecks in terms of performance and technology scalability. Another important benefit of using a memory array is that it can be dynamically configured into a custom logic or memory block, unlike CGRA or FPGA. Such malleable nature of RMBC can be extremely useful for compute-intensive as well as data-intensive applications, which benefits from improved memory latency and/or bandwidth. Experimental results demonstrated that the proposed framework achieves significant improvement in performance (up to 4.7X, 3.3X on average) and energy consumption reduction (up to 4.7X, 2X on average).

6. ACKNOWLEDGMENTS

This work was partially supported by NSF grants ECCS-1002237, CCF-0903430, and CNS-0915376. We would like to thank Dr. Somnath Paul for his comments and insights on memory-based computing in embedded systems.

7. REFERENCES

[1] S. Hauck et al. The chimaera reconfigurable functional unit. *IEE Trans. on VLSI*, 2004.

[2] Y. Park et al. Cgra express: accelerating execution using dynamic operation fusion. In *CASES*, 2009.

[3] S. Majzoub et al. Reconfigurable platform evaluation through application mapping and performance analysis. In *IEEE SPIT*, 2006.

[4] S. Paul and S. Bhunia. A scalable memory-based reconfigurable computing framework for nanoscale crossbar. *IEEE Trans. on Nanotechnology*, 2010.

[5] H. Hajimiri et al. Reliability improvement in multicore architectures through computing in embedded memory. In *MWSCAS*, 2011.

[6] S. Paul, et al. Nanoscale reconfigurable computing using non-volatile 2-d sttram array. In *IEEE-NANO*, 2009.

[7] H. Singh et al. Morphosys: an integrated reconfigurable system for data-parallel and computation-intensive applications. *IEEE Trans. on Computers*, 2000.

[8] N. Binkert et al. The m5 simulator: Modeling networked systems. *IEEE Micro*, 2006.

[9] C. Zhang et al. A highly configurable cache architecture for embedded systems. In *Computer Architecture*, 2003.