Software for Explicitly Parallel Memory-Centric Processor Architecture

Goce Dokoski, Danijela Efnusheva, Aristotel Tentov, Marija Kalendar
SS. Cyril and Methodius University - Faculty of Electrical Engineering and Information Technologies
Karpos II bb, PO Box 574, 1000 Skopje, Macedonia
E-mail: {gocedoko, danijela, toto, marijaka}@feit.ukim.edu.mk

***Abstract*— Advances in computer memory technology justify research towards new and different views on computer organization. This paper proposes a novel memory-centric computing architecture with the goal to merge memory and processing elements in order to provide better conditions for parallelization and performance. The paper introduces the architectural concepts and afterwards shows the design and implementation of a corresponding assembler and simulator.**

***Keywords*: Explicit parallelism, memory-centric, high-performance computing, assembler, simulator.**

I. INTRODUCTION

The most notable characteristic of a classical Von-Neumann- and Harvard – based computer architecture is the clear separation between memory and processing: a computer system consists of memory that keeps program code and data; and processing units that read program code, load data from memory and process it. In this context, it is important to note that the difference between memory and processing speed has always imposed a bottleneck in computing throughput [1], which has led to development of various caching solutions. However, memory speed is significantly improved now. For example [2] shows a DDR3 memory that achieves a working frequency of 1.8 GHz.

This justifies research towards new concepts in computer architectures that would make closer ties among computing and memory resources. It should be mentioned that current research is mainly focused toward parallel processing as processor technology slowly reaches its upper bounds on chip complexity and speed.

Our research focuses on achieving stronger merge between memory and processing units, i.e. incorporating processing hardware directly in memory. As part of the memory is always used to store program code and the rest is used for data, it makes sense to add execution hardware into the code parts. Also if the need for virtual memory is taken into account, and the memory hardware is observed as a set of blocks, then it would be convenient to have an execution unit in each memory block that contains program code. This allows completely concurrent execution of all programs that fit in memory at a given time. The only issues that need to be considered in this organization are data inter-dependence and synchronization among running threads, and they can be resolved by using an appropriate interconnection system.

A single self-executing unit is solely responsible to fetch and decode instructions from its code block, and then issue data movement and arithmetic commands over the bus so that they are performed on the data blocks. Since commands are executed directly over the memory data, there is no need for processor registers and they are not used. The removal of processor registers makes a significant simplification in the way programs are written, compiled and executed. This paper presents the influence these changes will make on processor performance.

The explicitly parallel instruction computing (EPIC) architecture as a concept implies that compiler is responsible for detection of instruction-level parallelism (ILP) in the high-level program code, embedding this information into machine-code executables and possibly even preparing a complete execution plan [3]. This relieves the burden on hardware for various processor features such as: dependence checking, branch prediction, pipeline hazards, out-of-order execution etc. Although, this implies great simplification for the EPIC hardware, at the same time achieving great performance improvements, still its most notable implementation – Intel IA-64, hasn't achieved a significant success. The presented architecture will incorporate some of the EPIC concepts, probably resulting in more intensive data exchange between the memories and computing elements, thus forcing better usage of computational hardware elements. Finally, that will hopefully result in better acceptance of the EPIC philosophy.

This paper presents and examines an instruction set architecture appropriate for the suggested processor architecture. As a proof of concept we show an instruction-level simulator design for the architecture and evaluate its performance. The following chapter presents the state-of-the-art in the field of parallel computing architectures. The next two chapters describe the memory-centric computing architecture and its instruction set architecture. The fourth chapter presents the corresponding simulator, followed by simulation results and their analyses. The paper ends with conclusion and future work.

II. STATE OF THE ART

The research in the field of computer architectures is usually focused on increasing their throughput. The most common ways to deal with this issue include efficient data

organizations and optimal interconnection between memory and processing units.

Memory-centric computer architectures with special attention on connectivity issues are proposed in [4]-[5]. Here the computer system is viewed as a reconfigurable array of processing units on one side, and memory blocks on the other side, all interconnected by direct communication links. The processing units and the memory blocks have 1-1 mapping, and their interconnection is realized by interconnection circuit (crossbar switch). This organization shows an improvement of around 75% in regard to classical computer organizations.

In [6] centralized memory pools are proposed that are designed for transaction-based applications. In order to allow optimal data exchange, they introduce specialized three-dimensional interconnection network.

Large numbers of papers also investigate adjustable models for memory-centric architectures [7]-[12][13] discusses the advantages and the reasons to use memory-centric architectures, and proposes various implementation options. [14]-[16] on the other hand give proposals for hardware realizations of different memory-centric architectures.

III. Architecture Details

As previously mentioned, the main idea of the proposed architecture is to merge the processing and memory units. This is done by implementing the processor's fetch and decode phases directly in the program part of the memory i.e. exactly where the instructions are located. Furthermore, the complete fetch/decode circuitry may be added to every program memory block, and in case this is too expensive, it can also be added to a group of blocks. So, one block or a group of blocks with execution capabilities will be called a *self-executing memory block* in the rest of the text.

Therefore, a memory-centric EPIC architecture as proposed in this paper consists of self-executing memory blocks, data memory blocks and their interconnection network.

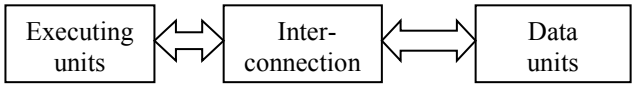

Fig. 1. Memory-centric computer architecture.

Every active self-executing block fetches and decodes its instructions and issues them to the memory blocks and to the ALUs. Whenever a new process is created, the system scheduler will assign a free block if available, or will free one in case all are busy.

Given that instructions are executed in-memory, the need for classical register becomes obsolete, and the instructions will work directly with memory locations. In order to decrease instruction length, it is a good idea to use a concept similar to segmentation. In this case, every executing block can work with one associated data block at a time, and the memory operands will represent address offsets in the data page instead of complete addresses. When data from another page is needed, the re-association will be performed by a special instruction. It should be noted that in this context, the terms block and page are used interchangeably.

Having only one associated data page at a time will impose problems when a thread accesses data parts that are interleaved over several data pages, because this will result in frequent data page re-associations, so it is a compiler's responsibility to provide an efficient data-to-page mapping and data manipulation algorithm. This is not a trivial task, and there is a lot of research that needs to be done in order to resolve this issue.

As mentioned previously, the goal of this research is to observe memory as a set of memory pages with execution capabilities that are interconnected by an appropriate bus system. It should be noted that this organization is especially convenient for FPGA implementation, because FPGA technology is already designed as a reconfigurable network of small memory and processing blocks. The only issue is the limit on achievable working frequency of the FPGA technology as well as its relatively high cost per implemented logic. Therefore, FPGA should be used for prototyping and afterwards the final product should be produced as integrated circuit.

IV. Instruction Set Architecture

The initial instruction set has simple MIPS-like instructions for arithmetical-logical operations, program flow control as well as auxiliary operations. Instruction operands always represent memory addresses that can address data either directly or indirectly. The instruction formats and their functions are shown in Table 1.

TABLE I
INSTRUCTION SET ARCHITECTURE

	Operation	**Instruction format**
arithmetical-logical	Addition	add dest, op1, op2
	Subtraction	sub dest, op1, op2
	Multiplication	mul dest, op1, op2
	Division	div dest, op1, op2
	Logical Bit-wise AND	and dest, op1, op2
	Logical Bit-wise OR	or dest, op1, op2
	Logical Bit-wise XOR	xor dest, op1, op2
	Logical Bit-wise NOT	not dest, op1
program flow control	Branch to dest if op1 is equal to op2	beq dest, op1, op2
	Branch to dest if op1 is not equal to op2	bne dest, op1, op2
	Branch to dest if op1 is greater than op2	bgt dest, op1, op2
	Branch to dest if op1 is less than op2	blt dest, op1, op2
	Branch to dest if op1 is greater than or equal to op2	bge dest, op1, op2
	Branch to dest if op1 is less than or equal to op2	ble dest, op1, op2
	Branch to dest	b dest
auxiliary	Terminate running thread and deactivate page	hlt
	Load byte op to dest	lb dest, op
	Load 2 bytes op to dest	lh dest, op
	Load 4 byte word op to dest	lw dest, op

All operations by default use direct addressing. Additionally, any operand may be set to use indirect addressing, by surrounding it in square brackets. For example, the instruction

add [000F0000h], 00FF0004h, [000F0008h]

will perform addition of the word on address 00FF0004h, and the word on address that is specified by the word on address 000F0008h. The result than will be written to the address specified by the word that resides on address 000F0000h.

Additionally, every instruction may accept an immediate value as a second operand. This is specified by adding *i* to the opcode (*addi*, *subi*, *muli* etc.) For example,

muli 000F0000h, 00FF0004h, 000F0008h

will calculate the product of the word on address 00FF0004h and the immediate value 000F0008h, and will store the result on address 000F0000h.

The assembler supports the data memory directives .space, .byte, .half and .word that allocate and initialize the data segments.

V. Instruction-Level Simulator

In order to make initial analysis, an instruction-level simulator is developed in the Python 2.7 programming language. It consists of several modules: *parser*, which contains an assembler grammar definition, parses the code, and returns python objects representing the instructions; *program memory units* that contain the code and execute it instruction by instruction; and *data memory units* that contain the data segments. It should be noted that the parser is developed by using the *pyparsing* library.

The main work of the simulator is done in the program memory units. Every unit has a program counter, page number, program number, time tag and a flag that indicates whether the unit is in use. The time tag is used to identify the least recently used unit, in the case when all units are active and there is a new thread that needs to be executed.

The simulator maintains an array of all code pages and performs cycle by cycle execution of all active units. Aside from showing the transition of memory states, it keeps information about the number of executed cycles, page faults and data page re-allocations. As such it is a valuable tool for performance estimation of the proposed architecture.

Currently the simulator doesn't simulate the interconnection among memory units and assumes constant time for data exchange. This will not be the case in real scenarios, so for more accurate results a simulation on FPGA prototype is needed.

VI. Simulation Results

In order to estimate and compare the performance benefits of a memory-centric architecture, we perform simulations and analysis for a typical computing problem: Fibonacci array generation. The program is written in assembler for both a classical MIPS assembler language as described in [17]; and the proposed MIPS-like assembler presented in the previous chapters of this paper. The former is simulated in the MARS simulator [17].

The tested program calculates a Fibonacci array of 4-byte elements and stores it in memory. The program code is first written for the memory-centric MIPS-like architecture, and is simulated for variable array lengths. Then it is rewritten for the original MIPS assembler and simulated for the same array lengths. Figure 1 shows a comparison of the total number of executed instructions. It can be easily noticed that the number of instructions is almost 50% lower with the memory-centric architecture. This means that even with half of the working frequency, the memory-centric architecture can potentially achieve the same performance.

This would also mean a 50% decreased power consumption.

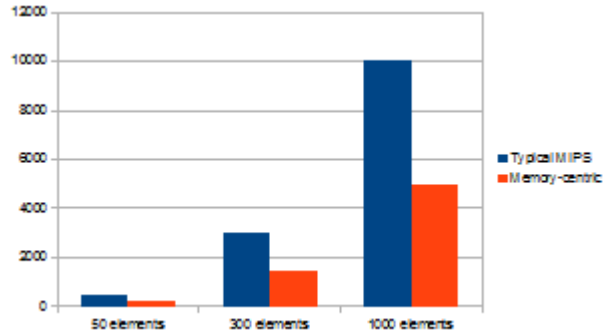

Fig. 1. Comparison of total number of instructions.

In order to make a fair comparison of execution time, we equalize the number of cycles for the two instruction set architectures. For a typical multi-cycle MIPS architecture the loads execute in 5 cycles; stores and R-type instructions in 4 cycles; whereas branching occurs in 3 cycles [16].

Since the memory-centric architecture accesses memory in every instruction we can safely assume that it doesn't take more than 6 cycles for each instruction. Figure 2 shows a comparison of the equivalent number of execution cycles. When using this model the number of executed instructions is still smaller although not with the same degree.

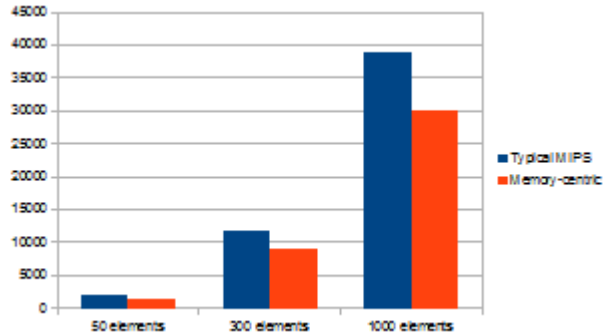

Fig. 2. Comparison of equivalent number of cycles.

VII. Conclusion and Future Work

This paper shows that a memory-centric architecture has the potential for achieving much better performance and lower power consumption than that of a traditional computer architecture. One reason for this improvement is the fact that programs contain significantly smaller number of instructions when written for the proposed architecture and require less clock cycles to perform the same operations.

If we also consider the better parallelization opportunities, it should become clear that this architecture is very

promising, and may even revolutionize the parallel computing paradigm.

It should be noted however, that presented analyses do not take into account cache block misses. This should not influence the results significantly, as the size of the self-executing blocks is similar to the cache memory blocks, and the time penalty for a miss should be of the same order of magnitude. Comparison that takes this parameter into account is left for future work.

Another important consideration for this architecture is the interconnection network, because its characteristics will have a crucial impact on the system performance. The influence will depend on many factors regarding the interconnection system, such as its structure and implementation, and this is a crucial part of our future research.

The proposed architecture is suitable for FPGA implementation due to its similarity to the FPGA technology – both are structured as a set of data and execution units. This solution however is limited when it comes to execution speeds due to the nature of the FPGA. Higher speeds will be achieved by using ASIC implementation and this opens up a whole new field of research.

REFERENCES

[1] C. Carvalho, „The Gap between Processor and Memory Speeds“,3rd Internal Conference on Computer Architecture (ICCA'02), Braga, 2002

[2] Memory Performance of Xeon E5-2600 v2 (Ivy Bridge-EP) based Systems, White Paper, Fujitsu, 2013

[3] Schlansker, M.S.; Rau, B.R., "EPIC: Explicitly Parallel Instruction Computing," Computer , vol.33, no.2, pp.37,45, Feb 2000

[4] P. Sirisuk et. al. (Eds.): ARC 2010, LNCS 5992, pp. 400-405, Springer-Verlag, Berlin Heidelberg, 2010

[5] Kiyoung Choi, „Reconfigurable Computing: Architectures, Tools and Applications“, 6th International Symposium, ARC 2010, Bangkok, Thailand, March 17-19, 2010

[6] Wassal, A.G.; Sarhan, H.H.; ElSherief, A., „Novel 3D memory-centric NoC architecture for transaction-based SoC applications", Electronics, Communications and Photonics Conference (SIECPC), 2011 Saudi International, pp.1,5, 24-26, April 2011

[7] Yamin Li, Sanli Li, and Wanming Chu, „Memory Centric Interconnection Mechanism for Message Passing in Parallel Systems“, Third International Conference on Massively Parallel Computing Systems (MPCS98), Colorado Springs, Colorado, Apr. 6–9, 1998

[8] ChunYi Su, Dong Li, Dimitrios S. Nikolopoulos, Kirk W. Cameron, Bronis R. de Supinski, Edgar A. Le´on, Model-Based, „Memory-Centric Performance and Power Optimization on NUMA Multiprocessors, Workload Characterization (IISWC)“, 2012 IEEE International Symposium on , vol., no., pp.164,173, 4-6 Nov. 2012

[9] D. Li, B. de Supinski, M. Schulz, K. Cameron, and D. Nikolopoulos, „Hybrid MPI/OpenMP Power-Aware Computing“, in IEEE International Symposium on Parallel Distributed Processing, 2010

[10] K. Singh, M. Curtis-Maury, S. A. McKee, F. Blagojevi´c, D. S. Nikolopoulos, B. R. de Supinski, and M. Schulz, „Comparing Scalability Prediction Strategies on an SMP of CMPs“, Proc. of the 16th international Euro-Par conference on Parallel processing, 2010.

[11] Berić, A.; Van Meerbergen, J.; de Haan, G.; Sethuraman, R., „Memory-Centric Video Processing“, Circuits and Systems for Video Technology, IEEE Transactions on , vol.18, no.4, pp.439,452, April 2008

[12] Ivan Lunteren, J., „Towards memory centric computing: a flexible address mapping scheme", Electrical and Computer Engineering, 1999 IEEE Canadian Conference on , vol.1, no., pp.385,390 vol.1, 9-12 May 1999

[13] Burger, D.; Goodman, J.R., „Memory-centric architectures: why and perhaps what", Innovative Architecture for Future Generation High-Performance Processors and Systems, 1997 , vol., no., pp.92,, 22-24 Oct 1997

[14] Bonatto, A.C.; Susin, A.A., „Memory subsystem architecture design for multimedia applications“, VLSI (ISVLSI), 2013 IEEE Computer Society Annual Symposium on , vol., no., pp.213,214, 5-7 Aug. 2013

[15] Gwangsun Kim; Kim, J.; Jung Ho Ahn; Jaeha Kim, „Memory-centric system interconnect design with Hybrid Memory Cubes“, Parallel Architectures and Compilation Techniques (PACT), 2013 22nd International Conference on , vol., no., pp.145,155, 7-11 Sept. 2013

[16] David A. Patterson, John L. Hennessy, “Computer Organization and Design: The Hardware/Software Interface (The Morgan Kaufmann Series in Computer Architecture and Design)”, Paperback, Morgan Kaufmann Publishers, Oct 2013

[17] Kenneth Vollmar, Pete Sanderson, “MARS: An Education-Oriented MIPS Assembly Language Simulator”, SIGCSE'06 Houston, USA, March 2006