Simulating PARALLELA P160X boards in PRAZOR simulator

David Greaves and Miloš Puzović

January 30, 2015

Contents

1 Introduction

This document describes design decisions that were made in order to simulate the PARALLELA board in the PRAZOR simulator [2]. The PARALLELA board is a high performance computing platform based on a dual-core ARM-A9 ZYNQ System-On-Chip and ADAPTEVA's EPIPHANY multicore coprocessor. There are three different commercially available models that are targeted to *microserver*, *desktop* and *embedded* markets. The main differentiator between these models is the host processor and FPGA logic. The host processor in microserver and desktop model is XILINX ZYNQ dual-core ARM A9 XC7Z010, while on embedded model the host processor is XILINX ZYNQ dual-core ARM A9 XC7Z020. The only difference between these two host processor is clock rate and size of caches and since these characteristics are parametirised in the PRAZOR simulator we are able to simulate both processors. The difference in FPGA logic is in number of *logic cells* and *DSP slices*. The microserver and desktop models have 28K logic cells and 80 DSP slices, while the embedded model has 80K

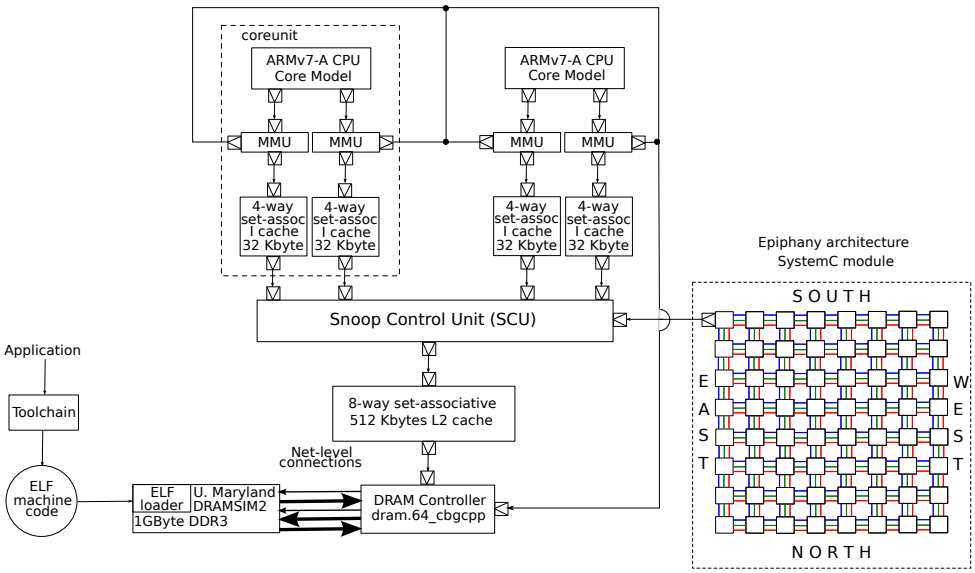

Figure 1: SystemC modules used to simulate Parallella board.

logic cells and 220 DSP slices. Since at the moment we are not simulating FPGA logic in Prazor this difference is of no concerns to us.

The rest of the document is organised as follows. In Section 2.5 we describe in details SystemC modules that are used to simulate various parts of the Parallella board. Section 3 discusses techniques that will be used to evaluate performance and power usage reported by the Prazor simulator versus numbers reported by running the same benchmarks directly on the Parallella board.

2 Implementation

Figure 1 shows SystemC modules that were used to simulate the Parallella board. From this Figure it is possible to separate implementation of simulation of Parallella board into five parts: host ARMv7-A CPU core model (Section 2.1), memory management unit (TLB and page table, L1 and L2 caches and snoop control unit, Section 2.2), DRAM controller and simulator (Section 2.3), Epiphany coprocessor (Section 2.4), and toolchain (Section 2.5) to be used to compile application to machine code runnable by the host processor.

There is one important assumption that we make in the SystemC model presented in Figure 1 about the *connection* between the host processor and Epiphany coprocessor. When one of the nodes in the mesh network of Epiphany coprocessor requests access to the memory that is off-chip (i.e. DRAM) then requests will be sent along Epiphany's east eLink to the FPGA. It is important to note that by design of

PARALLELA board anything sent to the west ELINK will simply disappear, while south and north ELINKS are used for the expansion boards. Once the off-chip memory request has been received by the FPGA the address in the memory request will be translated from the EPIPHANY's address space (from `0x8e000000` to `0x8fffffff`) to the physical address space (from `0x1e000000` to `0x1fffffff`) and forwarded to the host processor. The host processor is aware that that region of the physical address space is reserved for the communication with the EPIPHANY coprocessor and can be only accessed for that purpose. In our simulation address translation that is performed on PARALLELA board is modelled inside the mesh node that is in the top-left corner of the EPIPHANY's mesh network. We believe this model to be accurate enough and it does not require modelling of the FPGA logic.

The rest of this Section describes implementation details of each of five parts of simulator introduced previously.

2.1 Host processor

The XILINX ZYNQ programmable System-on-Chip contains dual-core ARM CORTEX-A9 based application processor unit with ARMV7-A architecture. This CPU can do up to 2.5DMIPS/MHz and has frequency up to 1 GHz. On PARALLELA board this CPU is clocked at 800MHz. Furthermore. The Cortex-A9 architecture was designed to be highly-efficient, dynamic length, multi-issue superscalar, out-of-order microarchitecture with 8-stage pipeline [3].

The pipeline supports up to *four instruction cache line prefetch-pending* and between *two and four instructions per cycle forwarded continuously into instruction decode*. The instruction decode stage is capable of *decoding two full instructions per cycle*. It supports *speculative execution* of instructions, *increased pipeline utilisation* by removing data dependencies between adjacent instructions, *four data cache line fill requests* and *out of order write back of instructions*.

Based on the current status of PRAZOR simulator the following new features need to be implemented in order to be able to simulate the host processor found on PARALLELA board and described in the previous paragraphs:

- Full support of ARMV7 microarchitecture instruction set - **Must Have** ☐
- Full support of ARM THUMB-2 instruction set - **Must Have** ☐
- Full support of ARM NEON media-processing instruction set - **Nice To Have** ☑

2.2 Memory management unit (MMU)

2.2.1 TLB and page table

In order to be able to compare PRAZOR to the other simulators it would be ideal if we could run LINUX operating system on it. The main missing part in the current implementation is that *translation lookaside buffer* (TLB) and *page table* walking algorithm were not implemented. As a part of this work we are going two implement

	Valid		Invalid
	Unique	**Shared**	
Dirty	Unique Dirty	Shared Dirty	Invalid
Clean	Unique Clean	Shared Clean	

Table 1: ACE cache line states

TLB and page table walking. It is important to notice that TLB and page table walking can be implemented as pluggable SYSTEMC module (as illustrated by block MMU in Figure 1) or it can be part of ARM core (*composite reuse principle*). We will provide both implementation where SYSTEMC module will be a wrapper around concrete implementation of TLB and page table walking algorithm.

Thus, the following work needs to be done:

– Implement coprocessor 15 form ARMv7 microarchitecture because registers present in that coprocessor are needed in order to make translation lookaside buffer and page table walking algorithm work - **Must Have** ☐

– Implement translation lookaside buffer - **Must Have** ☐

– Implement page table walking algorithm - **Must Have** ☐

– Implement SYSTEMC wrapper around the implementation of translation lookaside buffer and page table walking algorithm so that this unit can be made pluggable - **Nice To Have** ☑

2.2.2 Caches

The current implementation of the memory management unit in the PRAZOR simulator consists of standard snooping-based MESI coherent protocol and advanced *hammer* MOESI cache protocol that is found on the most modern AMD OPTERON processors. One of the main problems behind the current implementation of MMU is the lack of generality between these two different implementation as there is a lot of conditional code that is executed depending on the coherence protocol used. We are planning to remove this limitation by implementing ACE (AXI Coherence Extensions) protocol specification. This specification can implement various different policies such as directory based, snoop filter or no snoop filter and various different coherence protocols such as MSI, MESI and MOESI. Table 1 shows five states in which cache line can be, while Table 2 shows the mapping between ACE and MOESI cache line states.

The cache line changes state once it receives one of the transactions shown in Figure 2. For the brevity we only illustrate how memory request sent from the EPIPHANY coprocessor is satisfied by the host processor that has the requested memory address in L1 cache. For the full description of each transaction please refer to the official documentation [4, 9] about the ACE protocol.

Figure 3 shows what happens in the ACE protocol if EPIPHANY coprocessor wants to write to a cache line that is in *UniqueDirty* state in L1 cache of the second core in

ACE	MOESI	ACE meaning
UniqueDirty	Modified (M)	Not shared, dirty, must be written back
SharedDirty	Owned (O)	Shared, dirty, must be written back
UniqueClean	Exclusive (E)	Not shared, clean
SharedClean	Shared (S)	Shared, clean or dirty, no need to write back
Invalid	Invalid (I)	Invalid

Table 2: Mapping between ACE and MOESI cache line states

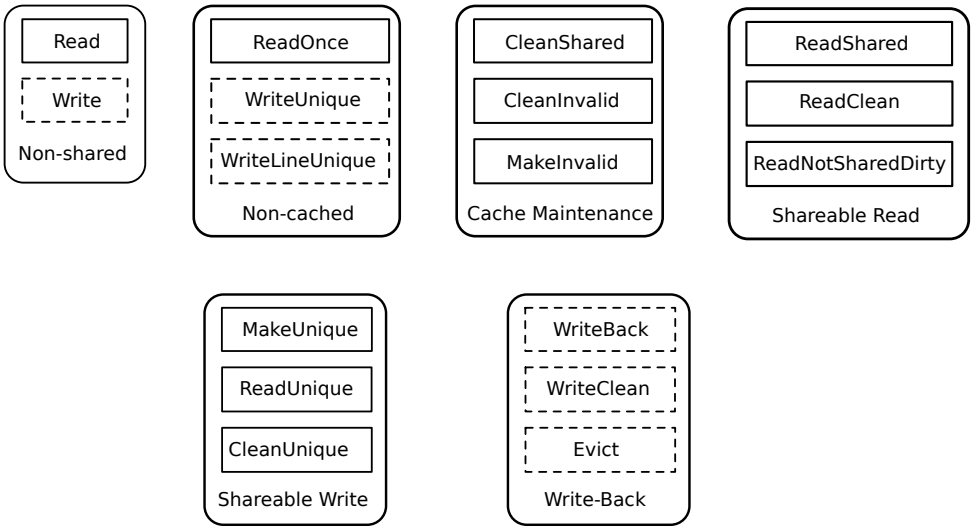

Figure 2: Transactions supported by the ACE protocol.

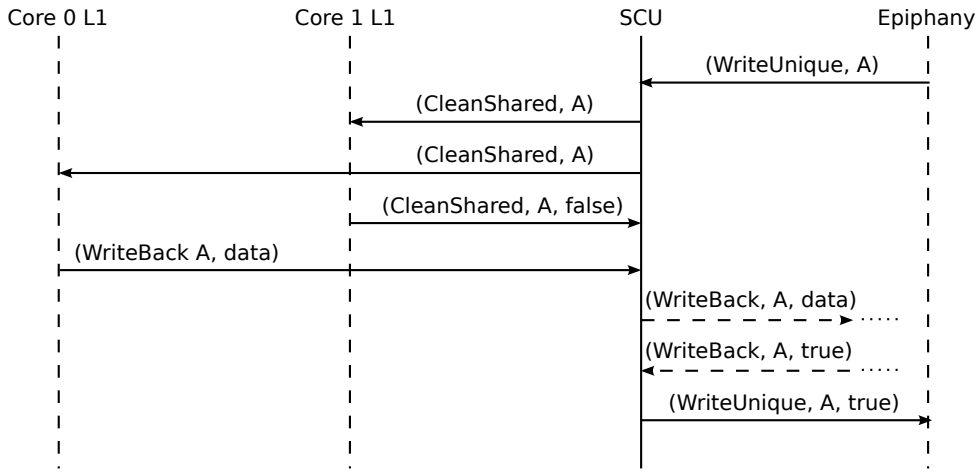

Figure 3: Execution scenario of a *WriteUnique* transaction.

the host processor. First, the EPIPHANY coprocessor will send request to the cache controller (in this case snoop control unit) that it wants to write to address `A`. Type of this request is going to be *WriteUnique* because EPIPHANY coprocessor is not cached. Once snoop control unit (SCU) receives this request it will send to all coherent caches *CleanShared* transaction. This transaction will find that cache line in L1 cache of the second core is in *UniqueDirty* state and that it should be written back to the secondary storage. As a result of this the cache will send *WriteBack* transaction with data to the SCU that will be then forwarded to the secondary storage. Once this transaction has completed the SCU you will inform Epiphany coprocessor that it can now write to address `A`.

Another problem with the current implementation of the MMU is that we cannot guarantee that all methods are *reentrant* and *thread-safe*. The main stumbling block is the implementation of the coherence checks where the cache that is checking for the line simply invokes method to check for the same line in the caches that are in the same coherence groups. This can lead to inconsistent states for the cache lines if caches that are being checked are operating on parts of that cache line (for example, when we have *false sharing*). To overcome this problem instead of using *cache coherence groups* we will replace them with the *Snoop Control Unit* and keep caches consistent using the ACE protocol. This will also enable us to remove locks that we have at the moment that make access to the caches sequential.

Therefore, the following works needs to be done for the memory management unit:

– Record metrics for certain set of benchmarks that collect how many times each lock has been taken and proportion of cache lines that has been accessed simultaneously by caches in the same coherent group. Use this data to discuss how contention can be modelled at the end of simulation - **Must have**, □

– Implement ACE protocol and replace the current cache lines states transition □

diagram with it - **Must have**,

– Replace cache consistent groups with Snoop Control Unit - **Must have**, ☐

– Run the new implementation in SYSTEMC multi-core simulator - **Must have**. ☐

The following list is a list of features that are **nice to have** and are closely related to the characteristics of ARM CORTEX A9 microarchitecture. Since most of these features are switched off by default we do not consider them as the main necessity for simulating PARALLELA board in PRAZOR simulator:

– **Speculative read** - at the same time when request is send to L1 cache it is also sent to L2 cache in order to reduce potential latency, ☑

– **Data and instruction prefetching** - there is a possibility to automatically prefetch both data and instructions from the caches from the cache line that is immediately next to the one that was requested, ☑

– **Clock gating** - L2 cache controller can automatically stop clock when no requests has been received after a few cycles. The clock is reactivated as soon as new request arrives, ☑

– **L2 power modes** - L2 cache has four different modes: *run*, *standby*, *dormant* and *shutdown*. ☑

– **Small loop optimisations** - there is a *small loop memory* coupled to the instruction cache that is used to store small loops that are less then 64 bytes of instructions. ☑

2.3 DRAM controller and simulator

The PARALLELA board uses 1GB 32-bit wide DDR3L SDRAM for off-chip memory. In order to simulate this memory we will be using SYSTEMC module that wraps a cycle accurate memory system simulator DRAMSIM2 [8]. No changes are needed to the current implementation of the SYSTEMC wrapper in order to simulate PARALLELA board in the PRAZOR simulator.

2.4 EPIPHANY coprocessor

The EPIPHANY coprocessor is a multicore, scalable, shared memory, parallel computing fabric [1]. It can contain 16 or 64 *computing nodes* that are arranged in two-dimensional array. These computing nodes are connected by a low-latency mesh network-on-chip. Figure 4 shows the implementation of the EPIPHANY architecture in SYSTEMC with a computing node zoomed in.

Each computing node contains a *superscalar*, *floating-point* RISC CPU. This CPU can execute *two* floating-point operations and a 64-bit memory load instruction on every clock cycle. The computing node has a local memory that supports simultaneous instruction fetching, data fetching and multicore communication. To achieve this, the

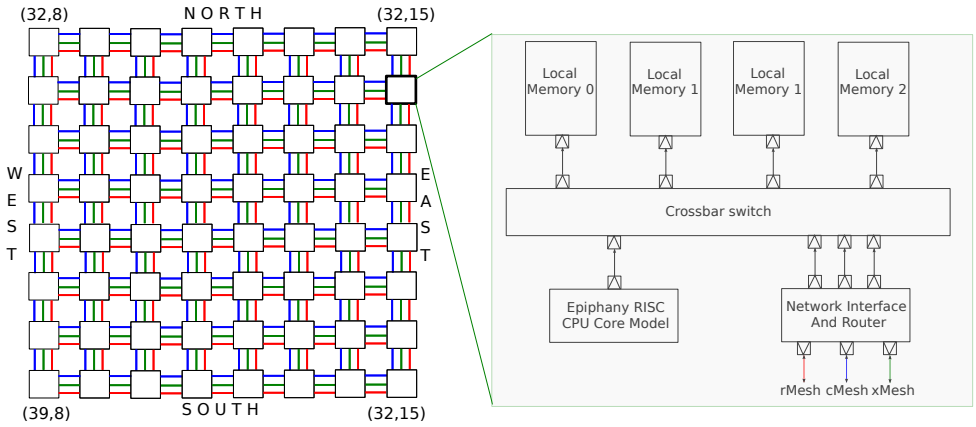

Figure 4: Implementation of the EPIPHANY architecture in SYSTEMC

local memory is divided int for 8-byte-wide banks each 8KB in size, giving in total 32KB of local memory.

The CPU core and memories are connected to a *mesh-node crossbar switch*. The crossbar implements fixed-priority arbitration and it is only needed when there is a potential for a shared-resource conflict. The access priorities needed to resolve conflicts are listed in the official documentation [1].

The connections between different computing nodes are managed by *network interface* and *router*. There are three different types of transaction traffic. The first type is *rMesh* and this is used for sending read requests with throughput of 1 read transaction every 8 clock cycles in each routing direction. The second type is *cMesh* and this is used for write transaction destined for on-chip mesh node with throughput of 8 bytes/cycle. Latency of write transactions is 1.5 clock cycles per routing hop. The final type is *xMesh* that is used for write transactions to off-chip memory. In the implementation proposed in this document the top-right computing node with coordinates $(32, 15)$ in Figure 4 is connected to off-chip memory and only that computing node is responsible for sending data off-chip. This is achieved by compiler from the toolchain that compiles code such that all off-chip access is encoded in address by setting first 6 upper bits to be 32 and second 6 upper bits to be in the range from 32 to 63. The full details behind the routing protocol and arbitration schemes can be found in the documentation on EPIPHANY [1].

Therefore, to implement EPIPHANY architecture in the PRAZOR simulator the following steps must be taken:

- Implement RISC CPU core with EPIPHANY instruction set architecture described in Appendix in [1] - **Must have** □
- For local memories use the existing implementation of `smallram` but add support for transactional access - **Must have** □
- Implement crossbar switch with arbitration as described in [1] - **Must have** □

– Implement network interface and router with routing protocol and arbitration as described in [1] - **Must have** □

– Parametirised the connections between computing nodes such that mesh networks with a different number of computing nodes can be created easily - **Must have** □

2.5 Toolchain

There are two different toolchains to consider: GCC and LLVM. Recently LLVM toolchain has been receiving a lot of attention because of its modular design and ease of extension. Although there exists a pass in the LLVM framework that can compile code for ARMv7 instruction set architecture there are no passes that can target EPIPHANY instruction set architecture. Some work has been started a year ago to add back end pass to the LLVM framework but it has been dropped. The code is still available on `github` [7] and we can pick it up if there is a need to use LLVM framework.

As a result of lack of support for EPIPHANY instruction set architecture in the LLVM framework the toolchain used in this project will be used is GCC.

3 Evaluation

The main goal of this project is to compare the accuracy of the PRAZOR simulator against a real hardware platform: PARALLELA board. As far as we are aware the most recent and the only work that has similar goal to ours is presented in [6]. In that work the authors investigated the sources of error in GEM5 [5] by validating it against the ARM Versatile Express TC2 development board. After making some modifications to the simulator they were able to achieve a mean percentage runtime error of 5% and a mean absolute percentage runtime error of 13% for the SPEC CPU2006 benchmarks and mean percentage runtime error of -11% and -12% for a single and dual-core runs respectively and mean absolute percentage runtime error of 16% and 17% for a single and dual-core runs respectively for the PARSEC benchmarks. Furthermore, authors in [6] have investigated accuracy of several microarchitectural statistics and showed that they were able to achieve accuracy within 20% on average for a majority of them. They have concluded that main source of the errors was modelling of similar, but not identical components.

In order to achieve our goals and based on the results obtained in [6] we are planning to do the following:

– Collect runtime and scaling accuracy for PARSEC benchmarks. This metrics will be used to evaluate the whole simulator shown in Figure 1 - **Must have**, □

– Collect observed memory access latency and bandwidth measurements for PARSEC benchmarks. This metrics will be used to evaluate memory management unit and DRAMSim2 of PRAZOR simulator - **Must have**, □

– Collect cache miss and access stats for PARSEC benchmarks. This metric will be used to evaluate memory management unit only - **Must have** ☐

– Collect branch and TLB miss states for PARSEC benchmarks. This metric will be used to evaluate behaviour of simulated ARM core. ☐

– Collect power and energy consumption for PARSEC benchmarks. This metric will be used to evaluate TLM POWER3 library - **Must have**. ☐

We are planning for the simulation of PARALLELA board in the PRAZOR simulator to achieve the following accuracy:

– Mean percentage runtime error between 7% and 10% for a single and dual-core runs of the PARSEC benchmarks, ☐

– Mean absolute percentage runtime error within 10% for a single and dual-core runs of the PARSEC benchmarks, ☐

– Accuracy of microarchitectural statistics between 10% and 15% for the PARSEC benchmarks and ☐

– Mean percentage power and energy consumption within 10% for a single and dual-core runs of the PARSEC benchmarks. ☐

– Mean absolute percentage power and energy consumption within 15% for a single and dual-core runs of the PARSEC benchmarks. ☐

Finally, since we are planning to put every physical/logical hardware unit that we are modelling in PRAZOR in a separate thread we should be able to see decrease in simulation time as number of cores dedicated for the simulator is increased. Therefore, we plan to do the following:

– Check to see if SYSTEMC kernel can run in multicore mode. If not investigate how difficult would it be to modify it such that it can run in multicore mode - **Must Have**. ☐

– Report metrics that measure impact on simulation time as number of cores used by PRAZOR increases - **Must Have**. ☐

References

[1] Adapteva, Inc. EPIPHANY *Architecture Reference*. 2013. Available: http://www.adapteva.com/docs/epiphany_arch_ref.pdf

[2] Adapteva, Inc. *Parallela-1.x Reference Manual*. Sept. 2014. Available: http://www.parallella.org/docs/parallella_manual.pdf

[3] ARM Ltd. *The ARM Cortex-A9 Processors*. ARM Whitepaper. Sept. 2007. Available: http://www.arm.com/files/pdf/armcortexa-9processors.pdf

[4] ARM Ltd. *AMBA AXI and ACE Protocol Specification*. Feb. 2013. version ARM IHI 0022E. Available: http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ihi0022e

[5] N. L. Binkert et al. *The* GEM5 *Simulator*. SIGARCH Computer Architecture News 39.2: 1-7. 2011.

[6] A. Gutierrez, J. Pusdesris, R. G. Dreslinski, T. N. Mudge, C. Sudanthi, C. D. Emmons, M. Hayenga, N. C. Paver. *Sources of error in full-system simulation*. ISPASS 2014: 13-22.

[7] Y. Hu. *LLVM back-end for Epiphany ISA*. May 2013. Available: https://github.com/Hoernchen/Epiphany

[8] P. Rosenfeld, E. Cooper-Balis, B. Jacobs. *DRAMSim2: A Cycle Accurate Memory Simulator*. Computer Architecture Letters 10(1): 16-19. 2011.

[9] A. Stevens. *Introduction to AMBA 4 ACE and big.LITTLE Processing Technology*. ARM Whitepaper. Jul. 2013. Available: http://www.arm.com/files/pdf/CacheCoherencyWhitepaper_6June2011.pdf