Lessons Learned Using ArchC in Computer Architecture Laboratory

Rodolfo Azevedo
Institute of Computing
University of Campinas - Brazil
rodolfo@ic.unicamp.br

Lucas Wanner
Institute of Computing
University of Campinas - Brazil
lucas@ic.unicamp.br

Abstract

This paper presents a set of laboratory experiments based on the ArchC Architecture Description Language designed to fulfill the practical knowledge on Computer Architecture. These activities were designed over the last years and have been used in our discipline of Laboratory of Computer Architecture, where seventh semester students apply the knowledge they acquired in the theory classes. We present the experiments, covering 10 distinct topics in Computer Architecture, along with specific sections of the textbook to which they refer. We also show some of the experiences we acquired during the last years both on learning outcomes and student feedback.

1. Introduction

The Computer Architecture community engaged a quantitative/practical approach for most of the research and teaching in the past decades. At the same time, abstraction level risen from the basic transistors, to gate level, functional units, and finally processor cores. Deciding the correct abstraction level to apply to one undergrad laboratory class is a very interesting challenge, ranging from lower level HDL processor implementation, to SoC designs, and even higher level abstract simulators for performance analysis. No matter the selected approach, it is clear that the experimental approach provides the best environment for both strengthening students previous knowledge and engaging students into new challenges.

The most recent ACM Curricula Recommendations lists desirable learning outcomes for each major area in the body of knowledge for Computer Science [12]. In Architecture and Organization, core knowledge includes the use of CAD tools for the simulation and evaluation of building blocks, instruction-level parallelism and hazards, I/O, interrupts, effects of memory latency and cache memories, and other topics relating to performance analysis.

ArchC [1,5,17] is an Architecture Description Language designed to facilitate the design and evaluation of Instruction Set Architectures – ISA, by allowing users to create and extend processors using a simple descriptive language based on SystemC/C++. The ArchC toolset comes with a simulator generator that is able to connect to external peripherals and also simulate cache behaviors. We will explore some of these features in the following sections of the paper. ArchC has been used as a basis for research in embedded systems development [4], memory hierarchy optimization [20], rapid prototyping [13], power modeling [10], multicore scalability [7], platform simulation [19], fast simulators [9, 21], among others.

Many computer architecture lab courses focus on building a processor or platform from the ground up, culminating in, for example, a software-based MIPS interpreter, or a simple VHDL processor. We find that this strategy shifts the focus from computer architecture to specific language or ISA details. Because our students take a practical digital logic course as well as an assembly language course prior to the Architecture lab, we are able to focus on core and advanced issues such as performance accounting, cache modeling, and superscalar processors.

At Unicamp, we have the following sequence of hardware related disciplines: Digital Logic (1 theory and 1 laboratory), Computer Organization and Assembly Language (integrated theory and laboratory), and Computer Architecture (1 theory and 1 laboratory). This paper talks about the last laboratory of this sequence, that students enroll in their seventh semester. For more than 10 years [16, 18], we have been using sets of laboratory experiments designed over ArchC. In this paper, we describe a series of teaching experiments using ArchC, along with collected lessons from our experience in teaching the course.

2. Teaching experiments with ArchC

In this section we describe a series of teaching experiments using ArchC. These experiments are appropriate for a two credits laboratory course following Patterson and Hennessy's Computer Organization and Design textbook [15], and in some cases may require further web and library research by students. Experiments range from basic performance accounting to caches, multicore processors, and peripherals. While we chose to follow the MIPS architecture described in the textbook, all of the experiments could easily be adapted to use other architectures supported by ArchC, including SPARC, PowerPC, ARM, and RISC-V.

Setup Prior to starting class assignments, we guide students through the setup process for ArchC with a web-based tutorial. The setup process for ArchC requires the installation of SystemC [3], a cross-compiler, and the ArchC simulation framework core, as well as the download and compilation of architectural simulation specifications. All of the experiments described in this paper only require changes to the architectural specifications, and not to the ArchC or SystemC cores. We therefore find it beneficial to provide students with pre-installed tools in shared network locations, which reduces the setup process to export environment variables pointing to appropriate installation directories. Because of library and compiler version dependencies, ArchC requires a uniform software environment across hosts sharing the same installation. Students wishing to setup the tools in their personal machines may therefore have to complete the setup process as described in the ArchC documentation [1].

After setup, we ask students to download and compile the MIPS architectural specification for ArchC. The ArchC simulator generator (`acsim`) is then used to read the specification files a set of source files implementing the simulator, which are in turn built with the help of a makefile. In order to test the setup process, we ask students to create a simple "Hello World" application, and to cross-compile it for MIPS using the ArchC specifications. The final objective of the setup task is simply to run the simple app on the simulator, but we find it helpful to include at this stage a brief discussion on cross-compilation tools, optimization levels, disassembly using `objdump`, and step-by-step execution with `gdb`.

Basic performance measurement [15, § 1.6] This introductory experiment is designed to familiarize students with usage and customization of the simulation tool, as well as to revise concepts relating to performance measurement, cycle accounting for different types of instructions, and performance comparison across a set of software programs.

The first part of the experiment has the students running different programs from the MiBench benchmark suite [2] on the simulator. Students build the simulator using the previously installed simulator generator (`acsim`) with the MIPS architecture simulator specifications, run pre-compiled binaries on the simulator, and observe program outputs. At the end of each run, ArchC outputs basic simulation statistics, such as the total number of instructions executed and total simulation time. It is important at this stage to emphasize the difference between simulation time (i.e., elapsed wall time from simulation start to finish) and benchmark execution time. Students are asked to derive the latter, initially based on the total number of instructions executed and a simple estimation of cycles per instruction (e.g, CPI=1.2).

Next, students are asked to modify the simulator to count the number of specific instructions executed by each benchmark. Instruction implementation in ArchC is captured through a hierarchical structure: first, a general behavior method, which captures functionality common to all instructions (e.g., advancing the program counter), is executed. Then, a type behavior method, and finally a specific instruction behavior method are executed. An example of the latter is shown in Figure 1 for the add immediate instruction. The core behavior of the instruction is adding the source register (`rs`) and the immediate value (`imm`) and storing the result in the destination register (`rd`). Arbitrary SystemC/C++ code may be used to provide additional functionality as needed. In the example, debug statements are used to generate detailed execution traces. We ask students write code to count the number of executions of a specific instruction for different programs (one interesting example for C programs is the MIPS R-Type `add` instruction).

```
void ac_behavior( addiu )
{
  dbg_printf("addiu r%d, r%d, %d\n", rt, rs, imm
      & 0xFFFF);
  RB[rt] = RB[rs] + imm;
  dbg_printf("Result = %#x\n", RB[rt]);
};
```

Figure 1. Behavior method for instruction `addiu`. The value in the source register is added to the immediate value and stored in the destination register.

Finally, we ask students to quantify benchmark performance using a CPI table dividing instructions into different classes (e.g., memory access, branches, and logic-arithmetic). The ArchC simulator building tool supports automatic generation of instruction execution counters, so that the technical work for the assignment consists in parsing simulator output, categorizing instructions, and calculating the number of cycles for each benchmark based on the CPI table. This experiment gives the first sense of processor performance and its implications to the students.

Pipelines and hazard detection [15, §§ 4.7–4.8] This experiment is designed to revise concepts related to pipelining, data and control hazards, and accurate performance accounting in simulation. In an introductory task, students are asked to recall why certain instructions in a pipelined processor may take more than one cycle to complete, and to think about the accuracy of the table-based cycle accounting method used in the previous experiment.

In the practical activity, we ask students to consult the textbook and to create a table of hazard types, both for control and data. Next, we ask them to list which hazards may be resolved through forwarding, and which instructions may be affected by each type of hazard. The core activity is then to create strategies for hazard detection in the simulator. This typically involves maintaining some history of recently executed instructions with their output registers, and for each new instruction, verifying readiness of source registers, and accounting for bubbles or stalls (extra cycles) when appropriate. Finally, we ask students to compare the results of the new accurate cycle accounting with the table-based method used in the previous experiment.

Branch Prediction [15, § 4.8] An advanced variation of the previous experiment asks the students to implement branch predictors to reduce stalls and improve performance. Students may for example compare performance under three scenarios: i) with no branch prediction, ii) with a static branch predictor (branch always taken or always not taken), and iii) with a simple dynamic branch predictor (e.g., a 2-bit prediction scheme). We ask the students to put their predictor implementation in the global behavior to be executed for all instructions and, at the same time, to restrict the amount of instruction information they have access to. For the prediction part, they do not know which instruction they are dealing with. Another key challenge for the students to understand is that they can execute the branch validation code in the next instruction, already knowing if the control flow changed. At the end of the experiment, they are asked to update their performance table from the previous experiments with the extra cost of the branch prediction.

Superscalar issue [15, § 4.10] In the earlier pipelining experiment, students should quickly realize that forwarding takes care of most data hazards. It may be interesting at this stage to contrast the performance of single-issue pipelines with that of superscalar ones. When comparing single-issue and superscalar pipelines, students may find that many benchmarks suffer from a large number of data dependencies, and that performance does not necessarily scale with issue width. This realization may in turn lead into a discussion on out-of-order execution. Because the textbook presents only a brief introduction to superscalar pipelines, it may be beneficial to supplement the reading for this exercise with other sources such as [11, §3], or with examples of superscalar pipelines in commercial processors. For the superscalar processors, they should understand when multiple instructions could be executed simultaneously and compare this approach to the hazard detection. At the end of the experiment, students are asked to derive program performance considering a specific superscalar implementation.

Memory-mapped I/O and peripherals [15, §4.9] In this exercise, students are asked to implement a simple peripheral and to write software that interacts with the peripheral through memory-mapped I/O. This is our first opportunity to show a broader view of functionalities in the simulator beyond the instruction set emulation components. ArchC allows for systematic connection of processor cores, memory, and peripherals in virtual platforms. Transaction-Level Models (TLM) [6] are used to separate computation from communication in a platform model. The basic point of connection of a processor core with other components in a platform is a TLM channel. The port provides methods for reading and writing words at specific addresses, and therefore are seen by the processor as if it were an ordinary memory element. For each memory request, ArchC creates a TLM request package, and calls the slave transport function through the channel. The slave transport implementation in turns directs requests to appropriate platform components according to request addresses. This allows for simple connection of new platform components such as memories and peripherals. Each new component must implement read and write methods that respond to TLM requests. Figure 2 shows an example of TLM transport. For addresses smaller than 100MB, the request is directed to the memory port, and for other addresses it is directed to a peripheral.

```
ac_tlm_rsp ac_tlm_router::transport( const
    ac_tlm_req &request ) {
  if((request.addr < 100*1024*1024)) {
    return MEM_port->transport(request);
  }else{
    return PERIPHERAL_port->transport(request);
  }
}
```

Figure 2. TLM transport example

We ask students to implement a peripheral providing test-and-set functionality. The peripheral is mapped to a single memory location. Write operations to the peripheral's address store the written value into a variable. Read operations return the stored value, and write a value of 1 to the variable. This peripheral can be used to implement a mutual exclusion abstraction. Entering the critical region corresponds to a busy loop reading while the value at peripheral's address is equal to 1. Leaving the critical region corresponds to writing a 0 to the peripheral. While this peripheral is not particularly useful in the single-core scenario, it will later be used in a multicore environment. For the MIPS processor, students also need to handle the endian difference among the host processor, an x86, and the simulated MIPS processor. For all previous experiments, ArchC took care of the endian but for the external peripherals, the users must be aware of it, making conversions on the read and write operations.

As an advanced version of this exercise, we may ask students to replace the peripheral by the two instructions MIPS use to implement mutual exclusion (`ll` and `sc`). For this method to perform the correct behavior, we rely on the fact that SystemC is based on discrete event simulation. In this way, all instructions behaviors are executed individually without concurrency. This method may be further improved when students design a cache coherency protocol in a later experiment.

Caches [15, §§ 5.3–5.4] The objective of this experiment is to revise caches, and to quantify the impact of cache design choices to program performance. We divide the experiment into two parts: first, students generate memory access traces for analysis with a cache simulator, and then students implement their choice of cache architecture in the simulator.

For the first part of the experiment, students generate memory access traces for analysis with a cache simulation tool such as Dinero [8]. Memory accesses for instruction fetches may be easily traced through the first-level, global instruction behavior method. Data memory accesses require changes to load and store instructions. For Dinero specifically, traces may be created in a text file, or fed directly into the simulator, which can be built into ArchC as a library. Given a memory access trace, Dinero provides performance analysis for caches with configurable levels, sizes, associativity, fetch, and write policies. Students are asked to select a number of cache configurations, and evaluate their performance for a set of benchmarks.

Following their initial analysis, students are asked to select one cache configuration for implementation in ArchC. The memory implementation in ArchC essentially stores and retrieves data into an array of size equal to the declared memory size. For this experiment, students must simply introduce a cache into the processor–memory path. The challenge of the exercise lies in correct cache implementation. In particular, programs must produce identical outputs across the cached and non-cached platforms, and cache performance should match the one reported by the simulator in the previous part of the exercise.

A memory performance model could be created afterwards, improving even further the system performance evaluation. At this point, we ask the students to consider only two fixed delays, one for the L1 cache and another for memory access. We ask the students to calculate the AMAT (Average Memory Access Time) for each program they run.

Multicore processors [15, §6.5] The objective of this exercise is to create a multicore platform, and to write parallel software that takes advantage of the multiple cores. Compared to similar exercises students may have seen before, for example, in an introductory parallel programming course, our focus is on the hardware/software mechanisms that enable parallel applications. In order to complete the exercise, students must learn to manage data stacks for each processor, and create basic synchronization and mutual exclusion abstractions.

In the first experiment, we ask the students to design a dual-core processor based on the MIPS platform and to correctly execute the initial setup of this processor. ArchC makes it easy to declare a dual core processor. It is just a matter of declaring the second processor as in `mips core1("core"), core2("core2");` and connecting both cores to the external bus or memory. The key challenge of this experiment relies on the booting process. Like all dual core processors, only one of the cores should start, executing the startup code, setting up the environment and,

Address	Symbol or segment
0x0000000	Data+ BSS segments
	...
	Stack space for processor 1
	...
0x7FF0000	Stack Pointer for processor 1
	Argument vector for processor 1
	...
	Stack space for processor 0
	...
0x7FFF000	Stack Pointer for processor 0
	Argument vector for processor 0
0x8000000	RAM_END

Figure 3. Memory map example for two processors

finally, signaling to the second processor to start. This task can be accomplished by creating an external controller and an enable signal for each processor. Initially, only one core has its enable signal set. Later, this primary core uses the external controller to enable the second one. This process could go on for several cores as necessary.

At this moment, the first core could start its execution but it is still necessary to setup the execution environment. As for ArchC MIPS model, this tasks involves setting up the stack address and split the execution flow between the cores. The stack setup is easily executed inside the ArchC model. The correct behavior is to create one stack to each core and we recommend the simple implementation inside the processor startup code. As for the execution flow split, students may use the previous created mutual exclusion peripheral or an specific booting sequence that independently enables each core with the respective id.

The last experiment in this exercise asks the students to emulate the PThread [14] library so that traditional thread code be executed inside the simulated environment. For this task, students are required to create/mimic the `pthread_create` code that enqueue one function to be executed by another core in the system. We do not require a scheduler, so each function should be executed to completion. Each extra core besides the first should wait for a data structure that is updated by `pthread_create` calls. Whenever this list is not empty and there is an idle core, the first function in the list will be executed. This environment could be incremented by adding extra PThread functions.

Figure 3 shows an example of a memory map for a platform with two processors sharing 128MB of RAM. In addition to setting up the stack pointer (register R29 in MIPS) to an appropriate location for each core, the arguments for main must be copied and setup in accordance with the MIPS ABI. The first two arguments for a function should be passed in registers R4 and R5 respectively. For the main function, the first parameter is the argument count, and the second parameter is the pointer to the argument vector. As

```
#define N 1024
#define NCORES 2
int coreNumber = 0, sum = 0, done = 0, v[N];
int main() {
  acquireLock();
  int myCoreNumber = coreNumber++;
  releaseLock();
  int start = myCoreNumber * N/NCORES;
  int finish = (myCoreNumber+1) * N/NCORES - 1;
  int mysum = partialSum(v,start,finish);
  acquireLock();
  sum += mysum;
  done++;
  releaseLock();
  if (myCoreNumber == 0) {
    while (done != N);
    printf("Result: %d\n", sum);
  }
}
```

Figure 4. Example of primitive work division and synchronization for multicore

a simplification, we can reserve a fixed length of memory for the argument vector (e.g. 4 KB).

Figure 4 shows a simplified example of memory sharing and synchronization across cores. The `partialSums` function, not shown in the example, simply iterates over a range of the input vector, returning the sum of all elements in the range. Global variables will be placed in the program's data segment, and therefore will be shared across all cores. Local variables are placed relative to each core's stack pointer. This way each core may compute a partial sum over the input vector independently (note that the simple work division strategy presented in the code only works for even vector lengths and number of cores). The `acquireLock` and `releaseLock` functions may be reused from the previous exercise about peripherals in order to ensure mutually exclusive access when writing to global variables. Primitive synchronization is performed through a worked counter (the `done` variable in the example). Students may build upon this simple strategy to build a PThreads-like interface for parallel programming.

One problem students frequently face in this exercise is proper stack sizing. If the stack for any core exceeds the allotted space, there will be data corruption across cores. A second problem is often found in the evaluation of the chosen parallel program. In many cases, synchronization overheads may lead to sub-optimal parallel performance. Replacing global locks with local locks protecting specific regions of memory or peripheral often ameliorates this problem. This issue is also a good opportunity to recall Amdahl's law.

Cache Coherence [15, §5.10] This exercise relates both to caches and multicore design and requires students to implement a cache coherence protocol either through directory or snooping. This implementation increases the system complexity requiring more attention to the details. Usually, the high level understanding of a cache coherence protocol is not enough to solve all implementation details and we give references to the students so that they can better understand the all required sub-states of the cache coherence protocol.

Uncore Accelerators The final experiment in the course asks the students to create an off-core accelerator and use it in order to speed-up the execution of a chosen application. Examples include floating-point co-processors, multimedia, cryptographic, or vector processing accelerators. Students must implement the peripheral for ArchC, define a timing model and software/hardware interface protocol, and create application-level abstractions or drivers to take advantage of the accelerator. Alternatively, the exercise may ask the students to implement a complex peripheral, such as a frame-buffer based display, a network interface card, or a sound card. In this case, the complexity exercise shifts from program acceleration to peripheral implementation.

3. Lessons Learned

In this section, we highlight lessons learned in teaching Computer Architecture Lab using ArchC.

Infrastructure matters: we found that oftentimes infrastructure issues take valuable time away from content and experiments. Releases of ArchC and SystemC have dependencies to specific compiler and library versions, and mixing and matching tools across versions often leads to errors that are perceived as cryptic by students. Due to inevitable differences in environments in students personal machines, it is beneficial to setup a uniform laboratory environment, complete with all the required tools and a pre-compiled version of ArchC. We typically allow students to setup ArchC in their own machines, but support only the pre-installed version. Recently, we also offered a virtual machine setup to the students. Because experiments usually require extra-class time to complete, there should be remote access available to students. Some experiments, such as generating memory traces, may have significant memory and storage requirements. Finally, while a small class may be manageable by a single instructor, in larger classes a teaching assistant may be needed to provide support for tools.

Programming skills: by the time our students take the Architecture Lab course, they have completed several semesters of programming-intensive courses. Nevertheless, we found that oftentimes, ArchC is the first *large* codebase students have worked with. Most of the experiments in our course require only small changes to the ArchC models, but students often struggle with finding appropriate places in code to make the changes, as well as with debugging issues such as segmentation faults or incorrect simulator behavior. In our first exercise, we typically include a brief crash-course on basic tools such as binutils and gdb. Students also have difficulty moving from the high-level specification of the problem we provide to a viable implementation strategy.

Attention to details: students typically take our lab course right after completing the prerequisite Computer Architecture course, and we found that most of the knowledge from the previous course is retained. Students can typically describe concepts such as benchmarks, CPI, pipeline, cache, and superscalar. Nevertheless, in translating these concepts to code, attention to detail is oftentimes lacking, perhaps due to the nature of the simulator, which by design uses arbitrary C code to simulate architectural functionalities. For example, students have designed caches with a number of blocks that is not a power of two. Sometimes, in the theory course, we only focus on higher level structures without implementation detail, like the cache coherence example, where sub-states are required for correct implementation but not discussed in the theory course.

Structuring of work: the experiments described in this paper may be structured both as individual exercises focusing on each functionality, as well as group projects encompassing two or three experiments. We found that it is beneficial to structure the more fundamental experiments as individual exercises, and the more complex ones as projects for groups of 3 or 4 students. While groups may suffer from an imbalanced distribution of work, they allow students to aim for more ambitious results. Interesting student projects encompassing peripherals, accelerators, and multicore platforms have included a Playstation One simulator, an efficient Mandelbrot set generator, electronic voting system, digital TV platforms, and so on.

Student feedback: every semester, students respond to a course feedback survey. Our feedback has generally been positive, but recurring complaints include a higher-than-expected course load (ours is a two-credits course), and the aforementioned infrastructure issues. Many students like the practical nature of the course, particularly for advanced concepts such as cache modeling and superscalar processors. Following are two quotes from student evaluations: "The challenge of implementing the hardware system used in the theory course is very interesting and improved my understanding of Computer Architecture." "Projects are interesting and instigating, mostly by being open ended."

4. Conclusions

This paper presented a set of experiments covering 10 distinct topics on Computer Architecture that we have been using at University of Campinas. All experiments were based on ArchC, an Architecture Description Language, allowing the students to strengthen their knowledge from the theory classes.

We found that they way we designed the experiments allowed us to focus on the Computer Architecture concepts rather than a specific hardware description language or low level simulator.

References

[1] ArchC project website. `http://www.archc.org`.

[2] MiBench embedded benchmark suite. `http://vhosts.eecs.umich.edu/mibench//`.

[3] SystemC project website. `http://accellera.org/downloads/standards/systemc`.

[4] R. Azevedo, S. Rigo, and G. Araujo. *Projeto e Desenvolvimento de Sistemas Embarcados Multiprocessados*, pages 331–386. Ed. PUC-Rio, 2006.

[5] R. Azevedo, S. Rigo, M. Bartholomeu, G. Araujo, C. Araujo, and E. Barros. The archc architecture description language and tools. *Intl. J. Par. Program.*, 33(5), 2005.

[6] L. Cai and D. Gajski. Transaction level modeling: an overview. In *CODES+ISSS*, 2003.

[7] L. Duenha, G. Madalozzo, T. Santiago, F. Moraes, and R. Azevedo. Mpsocbench: A benchmark for high-level evaluation of multiprocessor system-on-chip tools and methodologies. *J. of Parallel and Distributed Comp.*, 95, 2016.

[8] J. Elder and M. D. Hill. Dinero IV trace-driven uniprocessor cache simulator. `http://www.cs.wisc.edu/~markhill/DineroIV/`, 2003.

[9] M. Garcia, R. Azevedo, and S. Rigo. Optimizing simulation in multiprocessor platforms using dynamic-compiled simulation. In *WSCAD-SSC*, pages 80–87, 2012.

[10] M. Guedes, R. Auler, E. Borin, and R. Azevedo. An ArchC approach for automatic energy consumption characterization of processors. In *RSP*, pages 57–63, 2012.

[11] J. L. Hennessy and D. Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann, 5th ed., 2011.

[12] Joint Task Force on Computing Curricula, ACM and IEEE. *Curriculum Guidelines for Undergraduate Degree Programs in Computer Science*. ACM, 2013.

[13] F. Kronbauer, A. Baldassin, B. Albertini, P. Centoducatte, S. Rigo, G. Araujo, and R. Azevedo. A flexible platform framework for rapid transactional memory systems prototyping and evaluation. In *RSP '07*, pages 123–129, 2007.

[14] B. Nichols, D. Buttlar, and J. P. Farrell. *Pthreads Programming*. O'Reilly, 1996.

[15] D. Patterson and J. L. Hennessy. *Computer Organization & Design: The Hardware/Software Interface*. 5th ed., 2013.

[16] S. Rigo, R. Azevedo, P. Centoducatte, and G. Araujo. Uma nova abordagem para um curso de projeto de sistemas computacionais. In *WEAC*, 2006.

[17] S. Rigo, R. Azevedo, and L. Santos. Electronic system level design: An open-source approach, 2011.

[18] S. Rigo, M. Juliato, R. Azevedo, G. Araujo, and P. Centoducatte. Teaching computer architecture using an architecture description language. In *Proc. workshop on Comp. architecture education (WCAE)*. ACM, 2004.

[19] N. Ventroux, A. Guerre, T. Sassolas, L. Moutaoukil, G. Blanc, C. Bechara, and R. David. SESAM: An MPSoC simulation environment for dynamic application processing. In *Intl. Conf. on Comp. and Information Technology*, 2010.

[20] P. Viana, E. Barros, S. Rigo, R. Azevedo, and G. Araujo. Modeling and simulating memory hierarchies in a platform-based design methodology. In *DATE '04*, page 10734, 2004.

[21] H. Wagstaff, M. Gould, B. Franke, and N. Topham. Early partial evaluation in a jit-compiled, retargetable instruction set simulator generated from a high-level architecture description. In *DAC*, May 2013.