Validating a Timing Simulator for the NGMP Multicore Processor

Javier Jalle[†,‡], Jaume Abella[†], Luca Fossati[§], Marco Zulianello[§], Francisco J. Cazorla[⋆,†]
[†] Barcelona Supercomputing Center (BSC). Barcelona, Spain
[‡] Universitat Politècnica de Catalunya (UPC), Barcelona, Spain
[§]European Space Agency
[⋆] Spanish National Research Council (IIIA-CSIC). Barcelona, Spain.

***Abstract*—Timing simulation is a key element in multicore systems design. It enables a fast and cost effective design space exploration, allowing to simulate new architectural improvements without requiring RTL abstraction levels. Timing simulation also allows software developers to perform early testing of the timing behavior of their software without the need of buying the actual physical board, which can be very expensive when the board uses non-COTS technology. In this paper we present the validation of a timing simulator for the NGMP multicore processor, which is a 4 core processor being developed to become the reference platform for future missions of the European Space Agency.**

I. Introduction

Timing simulators are used to test processor designs, from RTL to system level. From a software perspective, the ideal timing model of the hardware should provide the exact latency for each instruction. However, the latency of an instruction, and thus the execution time of a task running in a processor, completely depends on many factors, including the hardware implementation: pipeline, input-data, memory hierarchy, data dependencies and execution history. For instance, every instruction has to be fetched, which generates a request to the instruction cache that may result on a cache hit or a cache miss depending on the execution history, i.e. if that address was fetched by a previous instruction.

Timing details of the processor are usually provided in the specifications/datasheets on a high level of abstraction using metrics such as processor frequency. Also required details from the board are given with similar metrics, such as memory frequency and column access latency for memories. In the best case, some information about the instruction and cache latencies is provided; however, many details needed for accurate timing simulation related to pipelining, data and structural dependencies, etc. are usually omitted. Moreover, latencies of interfaces (e.g., to access memory) are poorly documented and depend on complex timing interactions of several components such as the processor, the board and the memory chips. This is not a problem to obtain an idea of the performance of the processor, but it is not enough to derive accurate timing simulators.

In this paper, we focus on the space domain and the Cobham Gaisler Next Generation Multi-Processor (NGMP), a.k.a. GR740, multicore processor [10]. The NGMP is envisaged as the future computing platform for missions carried out by the European Space Agency (ESA). Such a processor for the space domain remains in service for decades and its design is refined over the years, responding to the new needs of the industry.

In order to test the accuracy of a timing simulator, its timing estimates have to be compared with values coming either from RTL simulation or execution on a board. However, in case the simulator does not provide accurate values, this kind of test does not provide any information about which part of the model is inaccurate. For instance, if the simulator has a bus latency that does not correspond with the reality, every miss in the L1 cache that accesses the bus will introduce an error on the estimate. This error accumulates, increasing overall inaccuracy. A methodology that allows to test the behavior of each resource separately is needed. This process allows not only to validate the behavior of the simulator but also to leverage certain timing parameters to provide better estimates. For instance, in the previous case described, if the methodology analyzes the latency of the different types of requests in the bus, it can infer their latency and thus, properly adjust the simulator's timing parameters to model it.

Contribution. In this paper we show the validation methodology for a timing simulator developed by us for the Cobham Gaisler NGMP processor [10]. We show the structure of the simulator and the procedure to validate its behavior in terms of instruction timing, memory hierarchy, interconnect access and multicore interference. We show how the simulator provides execution time estimates which have less than 3% error at cycle level for real applications and widely accepted benchmarks. Through this particular case study we provide a methodology that can be applied to other embedded processors considered in domains such as automotive, avionics, railway and the like.

The paper is organized as follows: Section II describes the processor and the simulator. Section III explains the methodology, which is applied in Section IV with the final accuracy results. Related work and conclusions are in Sections V and VI.

II. Timing simulator overview

We build a model of the NGMP simulator upon the SoCLib [20] simulation environment, which we properly modify to resemble the NGMP processor. The purpose of the simulator is to have a cycle accurate simulator environment, in order to correctly measure the execution cycles. However, our goal is not to model all the complexity of the processor, because there is a tradeoff between complexity (or simulation time) and accuracy. In case some accuracy has to be traded off for a more simple model, simplicity is achieved by modelling the most common case.

A. The NGMP processor

In the space domain, the NGMP architecture [10], whose latest implementation is the GR740 [11], is an architecture considered by the European Space agency for its future missions. The NGMP has four cores, each comprising an in-order pipeline. The pipeline consists of 7 stages: *fetch*, *decode*, *register*, *execute*, *memory*, *exceptions* and *commit*. The instruction and data caches are accessed in the *fetch*

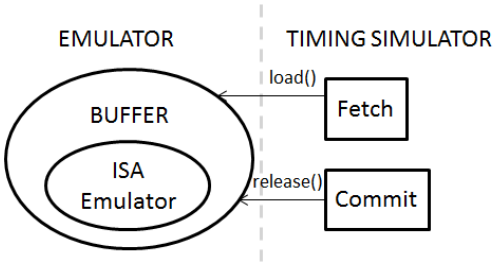
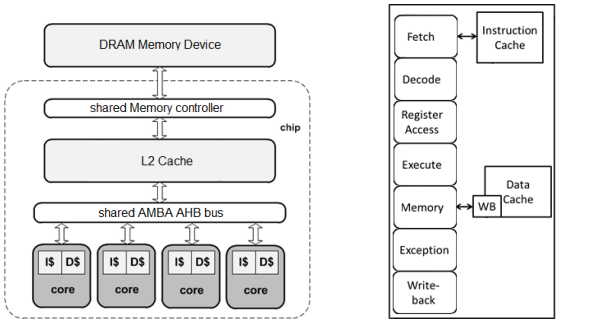


Fig. 1: Simulator structure



(a) Multicore architecture (b) Single core details

Fig. 2: Simulator architecture

and *memory* stages respectively. Floating point operations are performed in a separate functional unit.

Every core comprises a L1 instruction cache and data cache, each one with 4 ways, 16KiB (32 bytes line). The write policy for these caches is write-through, write no allocate. All cores are connected using a 128bit AHB AMBA bus arbitrated by a round-robin policy. The bus connects the processors with the L2 cache shared by all the cores in the processor. It is a 4 way, 256KiB (32 bytes line). The write policy is copy-back, write allocate. Some write-buffers exist somewhere, not specified, in the memory hierarchy of the processor.

B. Simulator architecture

The simulator is conceptually designed to separate the functional part from the timing behavior, which creates two different design spaces: (1) the *functional emulator* (emulator from now on) and (2) the *timing simulator*. The simulator structure is shown in Figure 1. The emulator part executes the instructions according to a particular Instruction Set Architecture (ISA) and provides all the information about an instruction like the instruction address, registers, type, results and memory address in case it is a memory operation. The timing simulator (simulator from now on) simulates the timing behavior of the instruction for a given hardware implementation, for instance, if it is a cache hit or miss and the delay introduced by the bus access in case it has to reach a higher level cache or the memory. The architecture of the simulator serves two purposes: 1) to resemble the actual processor structure which helps to have a more accurate modeling of its behavior and 2) to ease implementation of new hardware features such as bigger caches.

The simulator architecture, as shown in Figure 2a, consists of 4 cores with their respective private caches, connected through a bus to a shared L2 cache and the shared memory. The internal structure of the core is depicted in Figure 2b, which comprises the 7-stage pipeline with private caches and a write-buffer. Both pipelines, integer and floating point, are embedded in the execution stage, that assigns latencies to instructions. The timing of an instruction consists of the instruction latency, that is defined by the pipeline, and the memory hierarchy latency for the instruction access and data access, in case of load and store operations. The memory hierarchy latency includes a third contributor which is the multicore contention, which involves the effect that the rest of the cores can have on the timing of the core being analyzed, such as waiting for the bus because it is being used by another core.

C. Simulator parameters

The simulator allows parameterizing all hardware features that i) either allow fine tuning of the timing behavior, such as cache hit and miss latencies or ii) allow design-space exploration for new hardware features, such as cache size to test bigger cache designs. In particular, there are several parameters that allow fine-tuning the behavior of the simulator:

Instruction latencies: To assign a latency to each instruction, we classify the instructions into different types, having each type its own latency assigned. The contribution of an instruction to the execution time is caused solely by the latency of that instruction type if we consider always cache hits and no data-dependencies. In our case, the NGMP features a back-to-back execution pipeline in which all data dependencies are resolved. We classify the instructions into: Integer short: add, sub, cmp, ...; Integer long: mul, div, ... ; Branch: br, jmp, ...; Load; Store; Floating point short: fadd, fsub, ...; Floating point long: fmul, fdiv, ...; Special: Other instructions, traps, ...

Memory hierarchy: A set of important parameters is the one devoted to latencies of the complete memory hierarchy. These parameters include: IL1/DL1, L2 hit/miss latencies, as well as bus latencies and memory latencies. Another set of parameters includes the effect that the rest of contenders in the processor can have in the timing of the core analyzed due to contention on the access to shared resources. This is particularly taken into account on the processor bus, the L2 cache and the memory controller.

III. Methodology

Our validation methodology is based on *microbenchmarks* [19][7], a set of application kernels that exercise certain parts of the processor with a tow-fold objective. Providing confidence on the fact that the timing simulator is accurate by comparing their execution time on the simulator and the real board, and if the execution time is different, it helps adjusting the timing parameters of the simulator to improve accuracy for the specific components used by the microbenchmark.

A. Microbenchmark structure

We use *microbenchmarks* or Resource Stressing Kernels (RSK) comprising a single loop with instructions of the same type that are chosen to stress a certain hardware resource or a set of them. The reason to use a loop is to avoid IL1 (instruction L1) misses and exercise some latencies or certain behavior in the processor long enough to be able to minimize (1) the overhead of instrumenting and measuring the execution time and (2) artifact interferences such as memory refreshes that occur seldom over time and cannot be controlled. Each loop iteration contains N instructions or operations that have

```
1: for i = 0 to iterations - 1 do
2:     ld [0x00000000], $0
3:     ld [0x00001000], $0
4:     ld [0x00002000], $0
5:     ld [0x00003000], $0
6:     ld [0x00004000], $0
7: end for
```

Fig. 3: RSK example with *loads* with 4K offset

a certain behavior, such as an addition or an access to the L2 cache. An example of a RSK is shown in Figure 3.

By doing this, every benchmark we run, except for the specific loop branch instructions, has exactly the same overhead or structure. This simplifies getting a better prediction of the time increment caused by the instructions in the loop and not the rest of the benchmark. We assume that there is enough hardware and software support to be able to measure the execution time of the main loop with a certain degree of accuracy. The number of iterations of the loop helps adjusting the granularity at which execution time can be measured, since the behavior of a very short loop might not be captured by a coarse-grain performance monitoring counter and, on the opposite end, a very long loop might saturate fine-grain counters.

B. Description of the methodology

The first step is factoring out the execution time of the loop overhead in both, the timing simulator and the reference processor. This overhead is removed from all the execution time readings to obtain the exact latency of the operations performed in the loop body. The latency of the body helps tuning the latencies of the different parts of the processor with very high accuracy. The loop overhead is measured with a loop containing only the loop-related instructions, which include roughly an arithmetic instruction to compute the loop index and a branch instruction, and also possibly a comparison instruction in some ISA. The latencies of these instructions can be obtained from the datasheets, if available. Otherwise, they can be derived easily. For that purpose, we use a loop containing N times the instruction whose latency is to be measured, that is programmed in a way that IL1 misses can only occur in the first loop iteration, i.e. the loops fits in the IL1. This way we can approximate the latency of the instruction analyzed very accurately, since the overhead of the loop is minimized. Once each instruction latency is obtained we can set up the simulator with the appropriate latency for each instruction type.

In terms of implementation, we proceed as follows. We (automatically) create a set of RSK, each one having a different instruction type under analysis in the loop. All instructions are forced to incur cache hits in order not to include the memory hierarchy latency on the execution time. First, we measure the execution time of the empty-loop RSK. This execution time needs to be substracted from the execution time of any of the RSK that we produce to analyze any instruction. Then, the execution time difference between the specific RSK and the empty-loop RSK is divided by $N \cdot M$, where N is the number of instances of the instruction under study in the loop and M the number of iterations carried out. In order to validate the behavior of each instruction type, the execution time of the loop obtained in the simulator and the real board have to match for each type. Figure 4 shows the results of the instruction

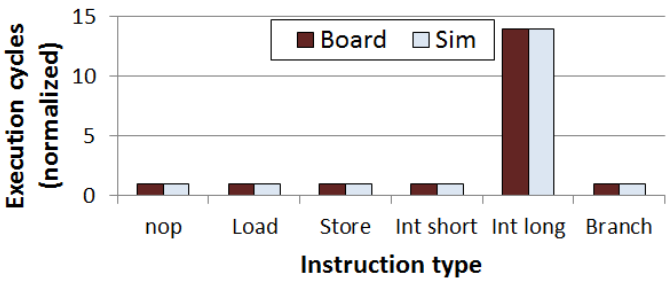


Fig. 4: Instruction timing

validation test for the different instruction types. All results are normalized with respect to the *nop* case.

For branch instructions, in case there is a different latency for taken and non-taken branches, the RSK can be adjusted to use non-taken or taken branches by setting the branch address to the exactly sequential instruction so that the control flow is exactly the same and they can be modeled separately. In our case, both cases behave equally. For input-dependent instruction latencies, as it might be the case for divisions or multiplications, several tests covering the different cases can be used to profile the different instruction latencies of that instruction type.

As next step, we address the memory hierarchy by testing all types of hits and misses on it. For that purpose, the instruction loop generates hits or misses on the instruction, data L1 and L2 caches. The same procedure followed here can be extended for processors with more hierarchy levels. All caches in our processor have LRU replacement policy, which is the most common in practice. To generate misses we perform $W + 1$ accesses, being W the number of cache ways, to the same set on each loop iteration. For instance, in the NGMP, caches have 4-ways so performing 5 accesses to the same set for different addresses causes that the first 4 accesses fill the 4-ways of the set and the 5th one evicts the 1st one from the set. When the access sequence repeats, the 1st address misses and evicts the 2nd address, which misses in turn and evicts the 3rd address and so on and so forth. The memory accesses systematically evict subsequent data to be accessed next and thus all accesses miss in the cache systematically. An easy way to access the same set is making cache accesses have an address offset equal to the way size of the cache, i.e., for the 16K L1 cache, 16K/4 ways = 4K. The resulting RSK is shown in Figure 3. By doing this we guarantee that we miss on that cache, but we hit on the bigger next level cache. To generate hits on a certain level of cache, we use the same procedure, making a loop that misses on the previous level. In the case of the L1 caches, accesses to the same address generate hits. For instruction misses, we use five branches physically separated by the required offset that jump sequentially. For data misses we use load and store operations with the given address offset. This benchmark structure is particularly devised for LRU and FIFO replacement policies. For other types of replacement policies similar structures can be built to produce systematic cache misses.

Figure 5a shows the results of the validation for load hit and misses in both, instruction (IC) and data (DC) L1 caches, normalized w.r.t. the execution of *nop* operations. Store operations in the L1 data cache, which is write-through, behave exactly the same whether they hit or miss since data are

forwarded to the next level anyway. There is a source of inaccuracy from our model that can be seen on this figure. Store operations are not exactly modelled due to the presence of one or more write-buffers that are not accurately described in the documentation.

In the L2 cache, we have four different types of accesses: loads and writes that either hit or miss. Loads can be instruction or data requests, since the L2 cache is unified. Figure 5b shows the values obtained for the different RSKs that generate each type of request. We can observe again the difference for store operations. The L2 cache case is more complicated, since according to the manual [10], L2 latencies are variable and depend on previous requests that were accessing the L2 cache, which can come from any of the 4 cores. However, this behavior is complex to model or test, since requires cycle-level control of the contention on the L2 cache which is impractical in reality. In this case, we choose to model a fixed latency, which reduces the accuracy but simplifies the L2 model.

As last step, we address the multicore contention on the shared bus. To that end, we use a recently published method [15] that allows to derive the latency caused by the contention on the bus. This method allows us to expose the interference that the cores generate for different types of request in a given shared resource. We use two different RSK at the same time, a sensitive RSeK and a stressing RStK. Both RSK perform continuous accesses to the shared resource under analysis by using the same technique presented before to miss in the appropriate cache levels. The RSeK runs on the core under analysis and the RStK run on the rest of cores, thus creating a high contention scenario. The method explained in [15] shows that round-robin arbitration behaves as a time-multiplexed scheme under a high contention scenario. In this situation, the access time of the RSeK accesses with respect to the time-multiplexed window can be varied by inserting a variable amount of nops between RSeK accesses to the shared resources. For each access time, the interference experienced has a different value and follows a sawtooth behavior. This sawtooth exposes the maximum interference delay on the bus as the frequency of the sawtooth. Similar analysis can be carried out in the memory, however, in our case, the interference occurs mostly on the bus because the bus serializes the traffic by stalling accesses until the request being processed is served.

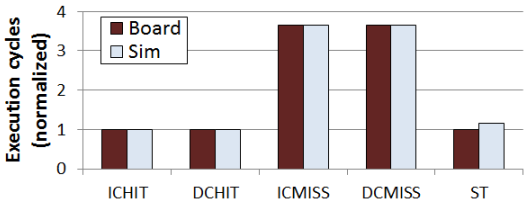
After the contention on the bus is properly adjusted, we run tests based on a RSK that uses a large fraction of the L2 cache on each core to create interference between the cores. These can be adjusted to interfere only on the bus or the memory by exceeding L1 and L2 capacity respectively only when run together. For instance, given N cores, each RSK can be designed so that the cache space used, C_i, matches the following constraint: $\frac{L2size}{N} < C_i \leq L2size$. Results of this part are omitted due to space constraints and will be included in the final version of the paper.

IV. Validation Results

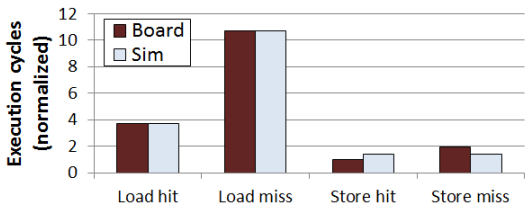
In this section we show how the simulator performs with different benchmarks and real applications to show its accuracy.

A. Experimental Setup

The processor we use as reference, a.k.a. *board*, is the only commercially available version of the NGMP, the GR-N2X [12] board. We collect the values from the available Performance Monitoring Counters (PMC) with the also commercially available GRMON [8] tool. The *simulator* already provides all the execution time values as output, without requiring any tool. The GR-N2X has an ASIC implementation of the NGMP architecture [10], whose latest implementation is the GR740 [11].



(a) L1 caches access latencies



(b) L2 cache accesses

Fig. 5: Memory hierarchy timing

B. Accuracy Results

For the validation with general benchmarks we use the EEMBC Autobench suite [18], which mimics some real-world automotive critical functionalities.

As real applications we use some software provided by the ESA. As payload applications we use the On-board Data Processing (OBDP) and DEBIE benchmark. OBDP contains the algorithms used to process raw frames coming from the state-of-the-art near infrared (NIR) HAWAII-2RG detector, already used on real projects, like the Hubble Space Telescope to detect cosmic rays. DEBIE is the software that controls an instrument, which was carried on PROBA-1 satellite, to observe micro-meteoroids and small space debris by detecting impacts on its sensors, both mechanically and electrically. As control application we use the Attitude and Orbit Control System (AOCS) from the EagleEye project [4] and the thruster vector control of the Vega launcher (VEGA). AOCS contains the Guidance and Navigation Control system from the spacecraft in charge of the correct position and orbit of the spacecraft. It is one of the most critical systems of a spacecraft, since a wrong position or orbit could mean the complete loss of the spacecraft, due to loss of power (not pointing to the sun for solar powered spacecrafts) or communication (antennas are directional and have to be properly oriented).

Figure 6 shows the accuracy, at cycle-level, for EEMBC automotive benchmarks and the four ESA applications. In this chart we can clearly see, that our simulator offers accurate results, with 3% of error on average. The lost in accuracy is caused by the design choices of not exactly modeling some of the behaviors. For instance the variable integer long instruction latency, variable L2 miss latency and the write-buffer. Also the difficulties to properly control and measure the contention in multicore workloads, makes their modeling difficult.

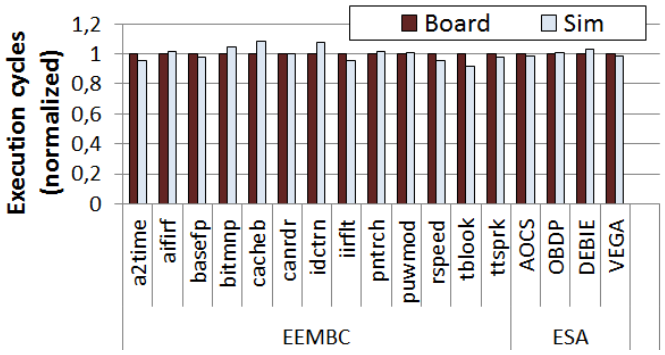


Fig. 6: EEMBC and ESA benchmarks accuracy results.

Our simulator is able to reach up to 1 MIPS. If we compare it with the performance of the actual board (around 100 MIPS for the benchmarks), we observe that we obtained accurate timing predictions for a tradeoff of performance. There is still room to improve the execution time of the simulator by tuning its code, however this is beyond the scope of this paper.

V. Related Work

Computer architecture research heavily relies on simulators since they allow the architect to quickly evaluate the performance of a wide range of architectures [21]. We can differentiate two main types of simulators. On the one hand, we have emulators that focus on the functional simulation and speed is their main design goal [2]. On the other hand, we find simulation infrastructures that include both, an emulator and a timing simulator. In this case, the speed of the simulator and the accuracy of its behavior are the metrics to optimize that usually come as a tradeoff [14]. A commercial option for simulating the LEON architecture are the GRSIM/TSIM simulators from Gaisler [9].

Simulator validation has been done by measuring the experimental error with benchmarks that target certain hardware resources [13][6][17][16] or statistically using Monte Carlo methods for out-of-order processors [1]. The idea of using loops to test performance has already been used in [5]. Authors in [3] improve their simulator accuracy also comparing with the actual hardware showing the need for a simulator validation in order to get representative results with the simulator.

However, to the best of our knowledge this paper is the first attempt to adapt simulator validation processes to the case of multicores for safety-critical systems, and produce a methodology portable to other processors in the same domain.

VI. Conclusions

Accurate but fast timing simulators have been regarded as mandatory to enable software development even before delivering actual hardware, to save procurement costs during development and to perform early schedulability analysis. However, a methodology to tune and validate those simulators is needed.

In this paper we apply such methodology through the particular case of the NGMP multicore targeting the space domain. By developing appropriate microkernels we are able to characterize the timing of the main processor components and develop accurate – yet light – performance simulators. Our results for the NGMP show an average inaccuracy of only 3% w.r.t. a real board implementation and different applications and benchmarks.

Acknowledgements

The research leading to these results has received funding from the European Space Agency under contract NPI 4000102880 and the Ministry of Science and Technology of Spain under contract TIN-2015-65316-P. Jaume Abella has been partially supported by the Ministry of Economy and Competitiveness under Ramon y Cajal postdoctoral fellowship number RYC-2013-14717.

References

[1] W. Alkohlani et. al. Accurate statistical performance modeling and validation of out-of-order processors using monte carlo methods. In *IPCCC*, 2014.

[2] Fabrice Bellard. Qemu, a fast and portable dynamic translator. In *USENIX*, ATEC '05, pages 41–41, Berkeley, CA, USA, 2005. USENIX Association.

[3] Bryan Black et. al. Calibration of microprocessor performance models. *Computer*, 1998.

[4] Victor Bos et. al. Time and Space Partitioning the EagleEye Reference Mission. In *Data Systems in Aerospace*, DASIA, 2013.

[5] P. Bose et. al. Bounds-based loop performance analysis: application to validation and tuning. In *IPCCC*, 1998.

[6] P. Bose et. al. Challenges in processor modeling and validation [guest editors' introduction]. *Micro*, 1999.

[7] Francisco J. Cazorla et. al. Multicore OS benchmarks. Technical Report Contract 4000102623, European Space Agency, 2012.

[8] Cobham Gaisler. *GRMON2-User Manual Version 2.0.62, March 2015*.

[9] Cobham Gaisler. *GRSIM Users Manual 1.1.51, 2015*.

[10] Cobham Gaisler. *NGMP Preliminary Datasheet Version 2.1, May 2013*.

[11] Cobham Gaisler. *Quad Core LEON4 SPARC V8 Processor - GR740-UM-DS-D1 - Data Sheet and Users Manual, 2015*.

[12] Cobham Gaisler. *Quad Core LEON4 SPARC V8 Processor - LEON4-N2X Data Sheet and Users Manual Version 2.1*, 2015.

[13] R. Desikan et. al., D. Burger, and S.W. Keckler. Measuring experimental error in microprocessor simulation. In *ISCA*, 2001.

[14] M. Durbhakula et. al. Improving the accuracy vs. speed tradeoff for simulating shared-memory multiprocessors with ilp processors. In *HPCA*, 1999.

[15] G. Fernandez et. al. Increasing confidence on measurement-based contention bounds for real-time round-robin buses. In *DAC*, 2015.

[16] M. Moudgill et. al. Environment for powerpc microarchitecture exploration. *Micro*, 1999.

[17] M. Moudgill et. al. Validation of turandot, a fast processor model for microarchitecture exploration. In *IPCC*, 1999.

[18] Jason Poovey. *Characterization of the EEMBC Benchmark Suite*. North Carolina State University, 2007.

[19] Petar Radojković et al. On the evaluation of the impact of shared resources in multithreaded cots processors in time-critical environments. *ACM TACO*, 2012.

[20] SoCLib. -, 2003-2012. http://www.soclib.fr/trac/dev.

[21] J.J Yi et. al. Simulation of computer architectures: simulators, benchmarks, methodologies, and recommendations. *Computers, IEEE Transactions on*, 2006.