Memory Hierarchy Calibration Based on Real Hardware In-order Cores for Accurate Simulation

Quentin Huppert[1], Timon Evenblij[2], Manu Perumkunnil[2], Francky Catthoor[2], Lionel Torres[1] and David Novo[1]

[1]*LIRMM, University of Montpellier, CNRS, Montpellier, France*

[2]*imec, Leuven, Belgium*

{quentin.huppert, lionel.torres, david.novo}@lirmm.fr, and {timon.evenblij, manu.perumkunnil, catthoor}@imec.be

***Abstract*—Computer system simulators are major tools used by architecture researchers. Two key elements play a role in the credibility of simulator results: (1) the simulator's accuracy, and (2) the quality of the baseline architecture. Some simulators, such as gem5, already provide highly accurate parameterized models. However, finding the right values for all these parameters to faithfully model a real architecture is still a problem. In this paper, we calibrate the memory hierarchy of an in-order core gem5 simulation to accurately model a real mobile Arm SoC. We execute small programs, which we design to stress specific parts of the memory system, to deduce key parameter values for the model. We compare the execution of SPEC CPU2006 benchmarks on the real hardware with the gem5 simulation. Our results show that our calibration reduces the average and worst-case IPC error by 36% and 50%, respectively, when compared with a gem5 simulation configured with the default parameters.**

***Index Terms*—computer system simulation, memory hierarchy**

I. Introduction

Architectures become increasingly complex to improve the performance and energy efficiency of modern computer systems. Parameterized simulators are a major tool driving this progress in computer architecture research. They allow quick iterations to test new architectures without having to fabricate real hardware. These architectures are often evaluated with respect to a baseline. Consequently, the relevance of the simulated results strongly depends on the faithfulness of that baseline.

There are two different sources of error leading to a flawed baseline: (1) errors in the simulation model (e.g., cache-bank conflicts are not modeled), and (2) errors in the model parameterization (e.g., the simulator is configured with a single-bank cache while the real reference architecture includes multiple banks). While continuous simulator development effort aims to remove the first source of errors, the second source is often neglected and remains problematic. Typically, the translation from documentation to simulator parameters is not obvious and some key parameters are not even disclosed publicly.

In this paper, we investigate the timing errors in model parameterization on the gem5 [1] simulator, which is being widely used in industry and academia. Previous work on gem5 simulation validation has mostly focused on the overall simulation error against real Arm [2], [3] and x86 [4] platforms. They all run small programs, named *microbenchmarks*, on gem5 and the reference architecture to analyze the simulation error of different components independently. However, none of them discusses the systematic design of such microbenchmarks in sufficient detail.

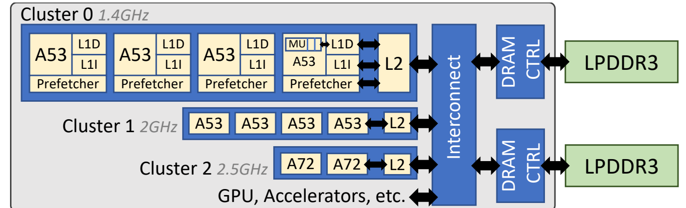

Fig. 1. Memory system organization of the MediaTek Helio X20 SoC.

To address this issue, in this work, we delve into the design of microbenchmarks for the calibration (i.e., finding the correct parameter instantiation) of the memory hierarchy of a gem5 model against a real hardware architecture. Importantly, we propose to use non-intrusive monitoring (i.e., hardware performance counters) to validate the microbenchmark intended execution. In this initial work, we concentrate on the memory hierarchy as a recent study shows that data movement dominates system performance and energy consumption in modern computing systems [5].

We make the following key contributions: (1) we describe the design microbenchmarks to calibrate the memory hierarchy of a parameterized simulator; (2) we calibrate a gem5 simulation of the Cortex-A53 core in the MediaTek Helio X20 SoC; and (3) we evaluate the simulation error running the *SPEC CPU2006* benchmarks before and after calibration.

II. Background

This section provides a brief background on modern *System-on-Chip (SoC)* memory hierarchy and the computer system simulator considered in this work.

The reference hardware architecture. We use the MediaTek Helio X20 SoC as the reference architecture for this work. Figure 1 illustrates the main blocks of this modern Arm big.LITTLE-based architecture [6] optimized for mobile computing. It contains three clusters exhibiting different performance vs. energy efficiency trade-offs. The first cluster is composed of two high-performance out-of-order Cortex-A72 CPUs. The other two clusters include four energy-efficient in-order Cortex-A53 CPUs, which have been optimized for two different frequency ranges. Every core of the system has its own private first level cache divided into instructions (L1I) and data (L1D). All the cores in a cluster share the same L2 cache memory. The L2 caches of each cluster are connected to each other and to the main memory via a cache coherent interconnect. The main

memory system is composed of two memory controllers, each connected to an off-chip LPDDR3 memory. In this paper, we calibrate the timing of the memory hierarchy as experienced by a single Cortex-A53 CPU.

The detailed memory hierarchy simulator. The gem5 simulator [1] is an open-source system-level and processor simulator widely utilized in academic research and in industry by companies such as Arm Research. Concretely, we use the gem5 *High Performance In-order (HPI)* CPU to model the Cortex-A53 clusters shown in Figure 1. The HPI CPU model was introduced by Arm to model modern in-order Armv8-A CPUs [7]. However, as we show in the paper, its default parameterization does not model a Cortex-A53 CPU accurately and it is not obvious how to calibrate the model given the huge set of parameters to tune.

III. Microbenchmark-based timing calibration

A gem5 simulation includes hundreds of parameters to accurately model real architectures. We start by composing a gem5 model based on the memory sub-system of the MediaTek Helio X20 but including only a single Cortex-A53 CPU (see Figure 1). Our model includes existing gem5 modules, such as the mentioned *HPI CPU*, the *cache*, the *XBar*, and the *SimpleMemory* modules that respectively model the CPU, the cache memories, the interconnect, and the main memory.

To identify the key simulator parameters that describe the memory hierarchy, we study the path that a memory request follows during simulation. When a memory instruction is executed by the *Memory Unit* (MU), the instruction is issued to a *Load/Store Queue (LSQ)* and a memory request is generated. The instruction stays in the *LSQ* until the memory request is fully executed. Two parameters are important: the maximum number of memory accesses and the size of the *LSQ*. Once a memory requests is issued, it goes through the different levels of the memory hierarchy depending on where the data sits. Using the generic gem5 cache module, our model instantiates 2 cache levels with different parameter values for each level. For the caches, we select five key parameters: size, associativity, data access latency, replacement policy and clusivity, which defines the inclusion policy. If a request misses both cache levels, it goes to the main memory through the interconnect, which the *XBar* module models as a fixed latency. Similarly, the *SimpleMemory* module models accesses to the main memory as fixed latency. We select all these key parameters, as listed in Table I, for proper instantiation.

We now need to find the correct values for each parameter to best match the timing of the real hardware platform memory system. The first source of information is public first-party documentation. We use the Cortex-A53 technical reference manual [8] and the SoC functional specification documentation [9] to find all the parameters highlighted in bold in Table I. However, several parameters that are key for an accurate timing simulation are not available in the documentation (e.g., the number of parallel access that can be processed by a cache, or the memory controller buffering latency).

TABLE I
List of key parameters with default and calibrated values.

gem5 Module	gem5 Parameter	Default	Calibrated[a]
HPI	executeMaxAccessesInMemory	2	**3**
	executeLSQTransfersQueueSize	2	**3**
	enableIdling	True	False
	srcRegsRelativeLats[b]	[2]	[0]
HPI_DCache	size	32KB	**32KB**
	data_latency	1	2
	assoc	4	**4**
	replacement_policy	LRURP	**RandomRP**
	clusivity	incl	**excl**
	writeback_clean	False	**True**
HPI_L2	size	512KB	**512KB**
	data_latency	13	10
	assoc	16	**16**
	replacement_policy	LRURP	**RandomRP**
Xbar	forward_latency	4	100
SimpleMemory	latency	30ns	30ns

[a]Bold values are extracted from documentation.
[b]Parameter from the HPI_DefaultMem64 submodule.

To discover the hidden key parameters, we propose to execute microbenchmarks on the real hardware. A microbenchmark is a small program that we design to target the extraction of specific missing parameters. We start again from the behavior of a memory request. The latter departs from the MU of the processor core and accumulates delay as it travels deeper in the target memory hierarchy (see Figure 1). The sum of these individual delays constitutes the access time of the request, as illustrated in Figure 2. Importantly, the path that a particular memory request follows depends on a set of conditions, such as where the data sits in the memory hierarchy or the current state of the microarchitecture (e.g., a resource is busy with a prior request). Thus, we exploit these dependencies to systematically design our microbenchmarks with the following structure:

1) *Data pinning.* The microbenchmark initializes data in a targeted level of the memory hierarchy and uses a fixed access pattern to force all memory requests to follow a predetermined path of the delay model. The data can be pinned in the different cache levels as well as in the main memory. We set the location of data by controlling the size of an array we repeatedly access (e.g., if the array is small enough to fit in the cache, consecutive accesses will not miss that cache) or by using specific cache flushing instructions (e.g., `dc civac` in Armv8 ISA). By varying the pinned location of the data in different microbenchmarks, all the paths from the delay model can be covered by at least one microbenchmark.

2) *Memory request dependency.* Multiple accesses to memory hierarchy can have an effect on each other. Contention conflicts or data movement can occur and modify the path

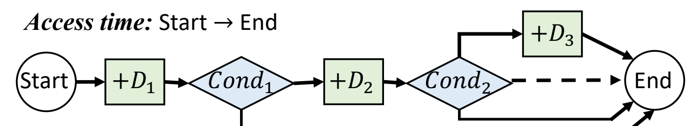

Fig. 2. The delay of a memory request increases as it deepens in the hierarchy.

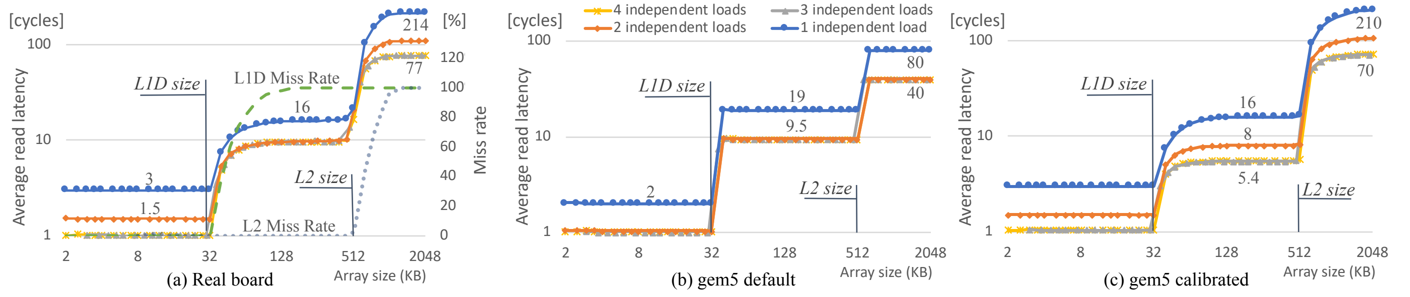

Fig. 3. Proposed microbenchmark running on (a) the real board and on the gem5 simulator with two configurations: (b) default and (c) calibrated.

through the delay model (e.g., the L2 cache bank is still busy with a previous request). By controlling the dependency between consecutive memory requests, the microbenchmark avoids or triggers such conflicts for further characterization.

3) *Access time measurement.* Once the memory access pattern has been fixed, the microbenchmark iterates the pattern many times in a loop, such that the overall execution time interval is large enough to average out transient effects. Furthermore, we unroll the body loop multiple times to minimize the impact of the iterator count and the conditional check instructions.

Importantly, the delay path that the microbenchmark takes during execution is not always obvious from its source code. Accordingly, we use *Hardware Performance Counters (HPC)* as a non-intrusive monitoring infrastructure to verify the execution of our microbenchmarks [10]. HPCs are present in all modern architectures and count events that occur in the hardware without disturbing the running application. Thus, we use the following HPCs to validate and correct our microbenchmarks:

- Total number of memory accesses: For example, a different number of memory accesses than expected could indicate that extra accesses are generated by the MMU or that accesses have been optimized away by the compiler.
- Cache misses at specific cache levels: For example, if we target the L2 access latency, the corresponding L1 cache miss rate should be close to 100%. Otherwise, our programmed memory access pattern has been overruled (e.g., by data prefetching).

Once the HPC values are consistent with the intended execution pattern, we run the microbenchmark to extract the targeted parameters. For that, we use again HPCs to restrict the timing measurement to the desired region of interest.

IV. Reference SoC Characterization

In this section we illustrate the microbenchmarking process with a concrete example. We design a microbenchmark to extract some of the memory hierarchy timing parameters mentioned in the previous section. We run the benchmark in a single Cortex-A53 CPU of the MediaTek Helio X20 SoC.

Figure 4 shows an instance of the example microbenchmark. The program generates `MAX` sequential load requests to a contiguous region of memory named `array` of size `N`. Considering that the granularity of data transfers in the memory subsystem is a cache line (i.e., 64 bytes), the microbenchmark should only access a single word of each cache line in `array` to avoid cache line locality. In line 2, we initialize each element in `array` allocated to a first word of a cache line. The initialized elements include the address of another randomly chosen element also allocated to a first word of a cache line. By not repeating any address, we close a circular chain of references. Thereby, we can generate an unlimited sequence of random read requests to different cache lines by iterating over `array` with a pointer (i.e., pointer chasing) as shown in lines 6–8. We randomize the sequence to avoid triggering data prefetching. Furthermore, the microbenchmark should traverse the array many times for cold start misses to become negligible. Accordingly, we select `MAX` to be several orders of magnitude larger than `N`.

We can control the size of `array` with `N`. For instance, if `array` is smaller than L1D, then `array` resides in L1D throughout the whole execution and every load request takes the access time of L1D. Thus, we can extract L1D access time by dividing the time spent in the region of interest (see Figure 4) by `MAX` (i.e., the number of load requests). Instead, if `array` is much larger than L1D but still smaller than L2, then the accesses to `array` always miss L1D and hit L2, which allows us to measure L2 access time. The same process can be repeated with an even larger `N` to measure the main memory access time. To measure the time spent in the region of interest and/or verify the access pattern, we initialize the HPCs at the begin of the region of interest and read them at the end. Figure 3(a) shows the average access time measured with the microbenchmark on the real board for different sizes of `array`. The figure also shows the L1 and L2 cache miss rates read from the HPCs.

We also use variations of the microbenchmark to measure the level of parallelism in the memory hierarchy. The load requests resulting from the same pointer-chasing operation (e.g., line 7 in Figure 4) are data dependent: the load request of iteration i should be completed before issuing the load request of iteration $i+1$. However, we can create opportunities

```
1 long long int array[N], *pointer_1, *pointer_2, i;
2 init_ptr_chasing(array); // Randomly linked cache lines
3 ptr_1 = array[0]; // Init 1st pointer to a cache line
4 ptr_2 = array[8]; // Init 2nd pointer to another line
5 start_hw_perf_counters(); // Reset HPC
6 for(i = 0; i < MAX/2; i++){ // Region of Interest
7     ptr_1 = *ptr_1;
8     ptr_2 = *ptr_2;}
9 read_hw_perf_counters()(); //Read HPC
```

Fig. 4. C code of a typical microbenchmark.

for memory-level parallelism by having multiple independent pointer-chasing operations. For example, Figure 4 shows a case for two independent load requests. Accordingly, we vary the number of independent requests from one to four as shown in Figure 3(a). We discover that the performance of the L1D and main memory improves linearly with the level of parallelism, saturating after three parallel requests. Instead, the performance of the L2 saturates after only two parallel requests. Therefore, our microbenchmark shows that although the Cortex-A53 can generate up to three outstanding memory requests, the L2 can only handle up to two requests in parallel.

V. Model calibration

Figure 3(b) shows the average access times of running our microbenchmark on the gem5 model using the default parameters. We observe that the L1 and main memory access times are lower than those of the real hardware, while the opposite is true for the L2 access time. Furthermore, we find that the number of outstanding requests in gem5 is only two instead of the three indicated in the reference manual and confirmed by our microbenchmark. Accordingly, we calibrate the gem5 model by combining the information found in the documentation with the timing values discovered via the execution of our microbenchmark. Table I shows the default and calibrated values for the key timing parameters. Importantly, Figure 3(c) shows that the new average access times of the calibrated gem5 model are practically identical to those of the real hardware. This demonstrates that by a proper systematic parameter calibration, the simulation models can become quite accurate. We also observe that the L1 and the main memory respond very similarly to parallel requests. However, this is not the case for the L2. In gem5, the L2 cache model is not able to replicate under any parameterization the contention exhibited by the real architecture. To reduce that error, we would need to create a new contention-aware cache model, similar to the one introduced by Evenblij et al. [11]. However, this falls out of the scope of this paper as it corresponds to a modeling error instead of a parameterization error.

VI. Application-level evaluation and conclusions

To evaluate the application-level accuracy of our microbenchmark-based calibration, we run 20 of the 29 programs in the SPEC CPU2006 suite on the real hardware and on two gem5 models: the default and the calibrated model as described in Table I. We only exclude the programs that we were not able to port to the Android-based MediaTek Helio X20 SoC. In gem5, we use system call emulation and follow a standard SimPoint methodology [12] with 100 million instructions per slice and max K set to 30.

Figure 5 shows the *Instruction Per Cycle (IPC)* error normalized to the real hardware execution. We sort programs in ascending number of *Misses Per Kilo-Instruction (MPKI)*, which is a proxy for memory intensity. The figure shows that the proposed memory hierarchy calibration reduces the average and worst-case IPC error by 36% and 50%, respectively. We also observe that the calibration does not always reduce the IPC error. Actually, 11 out of 20 programs have a higher IPC error

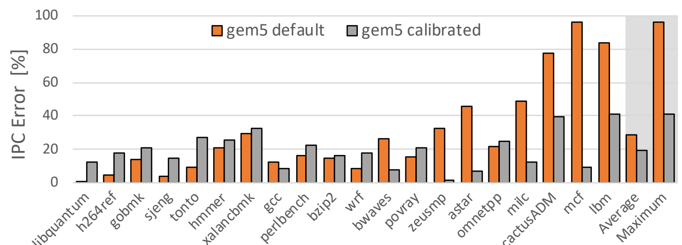

Fig. 5. Normalized IPC error of the default and the calibrated gem5 models with respect to a real hardware execution.

after calibration. This is not so surprising, as we only reduce the parameterization error in the memory hierarchy while all the other sources of error still remain after our calibration. When errors contribute to the IPC in opposite directions (i.e., the errors cancel each other for that particular execution), the removal of an error source can lead to a higher IPC error despite the improvement in modeling accuracy. Importantly, we see that the calibrated model consistently reduces the IPC error in applications of higher memory intensity (i.e., the contribution of the memory hierarchy error to the IPC is higher).

In conclusion, we presented a systematic method for instantiating the key timing parameters of the memory hierarchy in a gem5 model of a real modern mobile SoC. We described the design of microbenchmarks for that purpose and we showed significant improvements in IPC accuracy of the resulting calibrated model running full applications. In future work, we will extend the scope of our calibration to the data prefetching, out-of-order CPUs, and multicore systems.

References

[1] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness *et al.*, "The gem5 simulator," *ACM SIGARCH Computer Architecture News*, vol. 39, no. 2, 2011.

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

[3] B. Anastasiia, G. Rafael, O. Luciano, and G. Sassatelli, "Accuracy evaluation of gem5 simulator system," in *Proceedings of ReCoSoC*, 2012.

[4] A. Akram and L. Sawalha, "Validation of the gem5 simulator for x86 architectures," in *Proceedings of PMBS*, 2019.

[5] A. Boroumand, S. Ghose, Y. Kim, R. Ausavarungnirun, E. Shiu *et al.*, "Google workloads for consumer devices: Mitigating data movement bottlenecks," in *Proceedings of ASPLOS*, 2018.

[6] P. Greenhalgh, "big.LITTLE technology: The future of mobile," *Arm Limited, White Paper*, p. 12, 2013.

[7] A. Tousi and C. Zhu, "Arm research starter kit: System modeling using gem5," 2017.

[8] Arm, "Arm Cortex-A53 MPCore processor technical reference manual," https://developer.arm.com/documentation/ddi0500/d/, [Accessed: Dec-20].

[9] Mediatek, "MT6797 LTE-A smartphone application processor functional specification for development board," https://www.96boards.org/documentation/consumer/mediatekx20/additional-docs/docs/MT6797_Functional_Specification_V1_0.pdf, [Accessed: Dec-20].

[10] D. Terpstra, H. Jagode, H. You, and J. Dongarra, "Collecting performance data with PAPI-C," *Tools for High Performance Computing*, pp. 157–173, 2009.

[11] T. Evenblij, M. Perumkunnil, F. Catthoor, S. Sakhare, P. Debacker, G. Kar, A. Furnemont, N. Bueno, J. I. Gómez-Pérez, and C. Tenllado, "A comparative analysis on the impact of bank contention in STT-MRAM and SRAM based LLCs," in *Proceedings of ICCD*, 2019.

[12] T. Sherwood, E. Perelman, H. Greg, and B. Calder, "Automatically characterizing large scale program behavior," *Proceedings of ASPLOS*, 2002.