Racing to Hardware-Validated Simulation

Almutaz Adileh[†] Cecilia González-Álvarez[‡][*] Juan Miguel de Haro Ruiz[≀][*] Lieven Eeckhout[†]

[†]Ghent University, Belgium [‡]Nokia Bell Labs, Belgium [≀]Barcelona Supercomputing Center, Spain

***Abstract*—Processor simulators rely on detailed timing models of the processor pipeline to evaluate performance. The diversity in real-world processor designs mandates building flexible simulators that expose parts of the underlying model to the user in the form of configurable parameters. Consequently, the accuracy of modeling a real processor relies on both the accuracy of the pipeline model itself, and the accuracy of adjusting the configuration parameters according to the modeled processor. Unfortunately, processor vendors publicly disclose only a subset of their design decisions, raising the probability of introducing specification inaccuracies when modeling these processors. Inaccurately tuning model parameters deviates the simulated processor from the actual one. In the worst case, using improper parameters may lead to imbalanced pipeline models compromising the simulation output. Therefore, simulation models should be hardware-validated before using them for performance evaluation. As processors increase in complexity and diversity, validating a simulator model against real hardware becomes increasingly more challenging and time-consuming.**

In this work, we propose a methodology for validating simulation models against real hardware. We create a framework that relies on micro-benchmarks to collect performance statistics on real hardware, and machine learning-based algorithms to fine-tune the unknown parameters based on the accumulated statistics. We overhaul the Sniper simulator to support the ARM AArch64 instruction-set architecture (ISA), and introduce two new timing models for ARM-based in-order and out-of-order cores. Using our proposed simulator validation framework, we tune the in-order and out-of-order models to match the performance of a real-world implementation of the Cortex-A53 and Cortex-A72 cores with an average error of 7% and 15%, respectively, across a set of SPEC CPU2017 benchmarks.

I. Introduction

Computing systems are subject to continuous optimization to meet the ever-increasing performance, power-efficiency, reliability, and security demands. The high cost and turnaround time for prototyping computing systems lead computer architects to use processor simulators for swift and cost-efficient evaluation of novel ideas. Simulators rely on detailed models to account for the impact of the various processor components on overall performance. General-purpose processor simulators often expose these models to users in the form of configurable parameters. Simulation accuracy is thus a function of *both* how accurate the timing models are, and how accurately a user can tune the configuration parameters to model a real-world processor. According to Black and Shen's taxonomy [1], the former relates to *abstraction* accuracy whereas the latter relates to *specification* accuracy.

[*]This work was done while at Ghent University as a post-doctoral researcher (Cecilia González-Álvarez) and Erasmus+ exchange student (Juan Miguel de Haro Ruiz).

Modern simulators vary according to their modeling precision and level of abstraction. Each level is intended to drive processor assessment at a different stage of the processor design cycle. In general, high-precision modeling of timing and microarchitectural details induces long simulation times. Cycle-accurate simulators, such as [2]–[7], model the various processor components on a cycle-by-cycle basis. This level of accuracy comes at the cost of prohibitively long simulation times. In contrast, functional simulators, such as [8]–[10], are faster (by at least one order of magnitude) because they only model processor functionality without modeling its timing behavior. Sniper [11] and ZSim [12] are widely-used simulators that provide additional points on the speed-accuracy trade-off curve. More specifically, Sniper takes a unique approach to cycle-level simulation by relying on detailed high-abstraction models for all the processor components without having to simulate each component at each simulated cycle. By doing so, Sniper provides simulation accuracy comparable to detailed cycle-accurate simulators at a much higher simulation speed [13].

Modeling a real-world processor goes beyond the timing model. General-purpose simulators provide their users with a set of configurable parameters to model a processor of interest. Accurate simulation requires tuning the configuration parameters according to the modeled processor, in addition to the accurate timing model. Deviations between the settings of the model parameters and their counterparts in the real processor lead to incorrect findings. In a worse scenario, making errors when setting the configuration parameters can result in simulating an unbalanced processor design, and leaves the user with incorrect conclusions. Unfortunately, commercial processor vendors limit the amount of information they publicly disclose concerning their products. The scarcity of disclosed processor information encumbers the process of tuning the model according to a real-world processor. Thus, simulator users resort to best-effort estimates or rely on dated processor information to fill the missing model parameters. In an attempt to limit the margin of simulation error and raise the confidence in their results, simulator users validate their models against real-world processors [14]–[17]. The increasing complexity and diversity in modern-day microprocessors calls for a systematic approach to validate simulators against processors of interest.

In this paper, we propose a methodology for systematically validating simulators against real hardware. We rely on a machine learning-based racing algorithm [18] that compares the performance results of an actual processor with the performance of numerous processor configurations in simulation.

Every step of the algorithm eliminates the statistically unlikely configurations as new configurations with a lower error are evaluated. Within a finite number of iterations, our methodology generates a configuration that minimizes the error between the simulator and the real hardware. We advocate using a set of targeted (i.e., each stressing a specific processor component) micro-benchmarks because they allow quick evaluation of numerous configurations. Our methodology can be used to tune and validate any simulator, regardless of its precision, against any real-world processor.

We showcase the effectiveness of our methodology by demonstrating brand new support for the ARM ISA to the Sniper simulator [13], and by validating Sniper against actual ARM cores. We overhaul Sniper's front-end to decode ARM AArch64 binaries. We also adjust the timing contention models in Sniper to reflect the publicly disclosed information for existing in-order and out-of-order processors. We compare the in-order and out-of-order ARM models against the Cortex-A53 and Cortex-A72 cores, respectively. We show that by relying only on publicly available information, the error of the timing models could reach 33% and 45% on average across a set of SPEC CPU2017 benchmarks, relative to the Cortex-A53 and the Cortex-A72, respectively. Using our validation and tuning methodology, we manage to adjust the missing configuration parameters to limit the average error to less than 7% and 15% for the Cortex-A53 and Cortex-A72, respectively.

We make the following contributions in this paper:

- We propose a simulator validation methodology that leverages machine learning to automatically fine-tune unknown hardware simulation parameters to match real hardware measurements for a set of targeted microbenchmarks.
- We overhaul the Sniper simulator to provide support for the ARM AArch64 ISA. This includes replacing the Sniper front-end and the development of novel timing contention models for in-order and out-of-order ARM cores. We publicly release Sniper-ARM at `http://snipersim.org/`.
- Using the proposed simulator validation methodology, we validate Sniper-ARM against real hardware, including an in-order Cortex-A53 core as well as an out-of-order Cortex-A72 core. We report average simulation errors of 7% and 15% for the Cortex-A53 and Cortex-A72 cores, respectively, for a set of SPEC CPU2017 benchmarks.

II. Background and Motivation

Black and Shen [1] describe several potential causes of simulation errors. Aside from accidental bugs in simulator code, two main sources of error relate to modeling and abstraction errors. This work makes contributions that address both the abstraction and the modeling sides of simulation.

A. Hardware-Validated Simulation

Due to the significant role of simulation in processor architecture research and development, simulator accuracy must be scrutinized. Early work by Desikan et al. [14] shows cases where the expected error in simulation exceeds the benefits they expect from architectural techniques. As a result, follow-on work strives at understanding the sources of error in commonly used simulators and advocates validating simulators against real hardware, see for example [16], [19]–[23]. Real hardware is the golden reference according to which simulator accuracy can be judged. Using a simulator to assess the performance of a processor hinges on first establishing that the simulator accurately models real hardware.

Simulator validation is a challenging task. It is error-prone, time-consuming and tedious. Processor vendors offer a wide variety of designs to meet the market demands. One of the main challenges to simulator validation is the lack of the publicly disclosed information on the processor being modeled. For this reason, Desikan et al. [14] restricted their validation to the Alpha 21264 processor whose microarchitecture is disclosed in considerably more detail than other processors [24], [25].

This specification challenge (or lack thereof) triggered Walker et al. [17] to propose a methodology to evaluate the sources of error in simulation relative to real hardware using the `gem5` simulator and the configuration parameters that come with it. Their proposed methodology employs clustering and regression analysis to understand the relation between hardware and simulator performance events and their association to the error in performance. Unfortunately, their methodology is cumbersome and does not guarantee an easy identification of the sources of error in the model. For example, each clustering phase could lead to a different set of performance events, and can also differ from the findings of the regression analysis. Finding the exact source of error remains challenging in spite of this analysis. More importantly, even after identifying modeling errors in the simulator, this technique does not provide a systematic solution for how to fix it. If there is a specification error in the model due to a lack of disclosed information, this approach does not help find the correct specification.

Other proposed validation strategies focus on embedded processors. The methodology of Lattuada and Ferrandi [26] identifies the timing errors of a simulator using a recursive analysis of the application traces. They demonstrate their approach for the TSIM simulator[1] and the LEON3 processor, and focus only on finding timing errors, rather than exploring and proposing the best simulator configuration as this work does. Jalle et al. [27] show a validation methodology tailored for an NGMP embedded processor. However, while they methodically describe every step for identifying timing errors, the final configuration of the simulator is performed manually.

In contrast with all the previous approaches, we propose a new validation methodology that automatically explores all unknown parameter configurations to minimize simulation error.

[1]https://www.gaisler.com/index.php/products/simulators/tsim

B. Sniper

Without loss of generality, we select the Sniper simulator [11], [13] to showcase our validation methodology. We observe that although x86 and ARM ISA are widely recognized as the two most popular ISAs, most modern simulators support only the x86 ISA, see for example `gem5` [2], MARSS [6], Multi2Sim [7], Sniper [13], ZSim [12], PTLSim [4] and MaxSim [28]. Apart from several functional simulators [8], [10], including ARM Fast Models[2], `gem5` is the only widely-used cycle-level timing simulator that supports the ARM AArch64 ISA.

Sniper provides a unique speed versus accuracy trade-off compared to other cycle-level simulators. It relies on detailed timing models of the processor pipeline to perform accurate cycle accounting, while eluding the need to simulate every component in each cycle. This allows Sniper to perform fast simulation of unmodified applications at the expense of a minor degradation in accuracy relative to a cycle-by-cycle simulator. Sniper's high accuracy and high simulation speed renders it appealing for performance analysis for academic research and at early design stages in industry.

In this work, we augment Sniper with a brand-new front-end to support the ARM AArch64 ISA. We develop the necessary back-end timing contention models in Sniper to provide modeling capability for both in-order and out-of-order ARM cores. We use these core models to showcase the effectiveness of our methodology at eliminating abstraction and specification errors.

III. Validation Methodology

We propose a systematic approach to hardware-validated simulation. Our methodology includes both an automated part and a human element that, similar to all validation work, is required to inspect and contrast the peculiarities of the simulator against real hardware. Our approach has two advantages over prior work. First, it simplifies the validation task by exploiting targeted micro-benchmarks to isolate errors. Second, it automates the process of selecting the best configuration when identifying a problem in the model.

A. Overview

Figure 1 provides an overview of the proposed simulator validation methodology. Steps #1 through #3 are non-iterative, i.e., they are done only once at the beginning of the validation task. Step #1 involves human intervention to gather all publicly available information on the modeled processor from reliable sources and plugging them into the timing model of the simulator. We note that there exist other parameters that are not necessarily provided as part of the technical reference manuals or given just as approximate values, e.g., cache access latencies. In step #2, we estimate the access time of the L1 data and instruction caches in addition to the L2 cache using the lmbench micro-benchmarks [29], and plug them into the timing models as well. In step #3, the remaining unknown configuration parameters are set into the timing model using the best guess of the user.

[2]https://developer.arm.com/products/system-design/fast-models

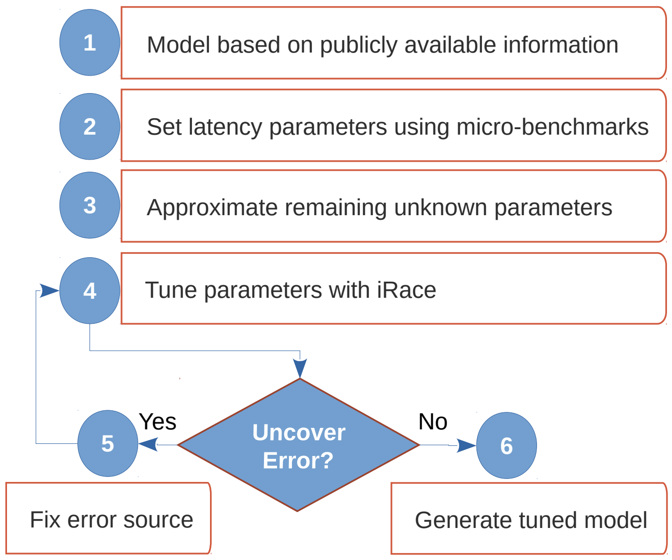
1
Model based on publicly available information
2
Set latency parameters using micro-benchmarks
3
Approximate remaining unknown parameters
4
Tune parameters with iRace
Uncover Error?
Yes
No
5
6
Fix error source
Generate tuned model

Fig. 1. Overview of the proposed simulator validation methodology.

Step #3 is the source of specification errors that validation techniques try to minimize. Step #4 is a distinctive step in our methodology that targets this particular issue. Instead of guessing the best configuration, we prepare a list of all the configuration parameters that require a best guess in step #3. We pair each parameter in the list with all the candidate values it could take. For example, a list may contain all the L2 cache address hashing techniques (as implemented in the simulator), the numerous data prefetchers that are possible at each cache level, configuration parameters for each prefetcher, etc. The list of unknown parameters grows with the model precision and the complexity of the real processor. In step #4, we pass the list to a parameter tuning tool that is responsible for selecting the configuration that minimizes a pre-defined cost function. The cost function in the case of hardware validation is the performance prediction error of the simulation model.

Step #4 searches for the best configuration based on a set of micro-benchmarks. We implement a framework that automates this step. The framework automates collecting performance measurements on real hardware for each micro-benchmark. It also generates a trace of the same instructions used for real-hardware measurement for each micro-benchmark. This framework contains the necessary scripts to launch a simulation for each trace using Sniper, collect the performance metric of choice, compare it with the hardware-obtained results, and return the estimated error for that particular simulation. The configuration tuning tool uses this script to launch the necessary simulation experiments for all the traces using configurations derived from the list generated in step #3. The tool finds the configuration that minimizes the simulator's performance prediction error. The algorithm stops after finishing a pre-configured number of optimization rounds. We provide further explanation on the choice of micro-benchmarks and the parameter tuning algorithm in Sections III-B and III-C, respectively.

Step #5 evaluates the error in simulation after the parameter tuning process. Each of the micro-benchmarks we use in step #4 stresses a particular component of the processor, and can thus expose modeling errors related to that component. Step #5 checks whether the modeling of certain processor components, as suggested by high errors for their respective micro-benchmarks, requires further optimization in the simulator. As the complex interaction among processor components determines its overall performance, it is possible for step #4 to configure other components properly, resulting in a low overall modeling error (when measured across all micro-benchmarks) while masking the error in one specific component. For example, we have seen scenarios where step #4 manages to improve the overall modeling accuracy, while a couple micro-benchmarks in step #5 reveal that there is still room for improving the indirect branch model.

Step #5 recommends performing an extra optimization round to focus on the modeling of a particular component. This step emphasizes the importance of accurately modeling all components and not relying only on a seemingly low error in overall performance estimation [16], [17]. For optimizations targeting a specific component, we recommend including metrics that are relevant to that component in the cost function of the tuning algorithm. For example, instead of using the Cycles-Per-Instruction (CPI) error only, a weighted cost function that includes both the branch misprediction rate and the CPI can be used. Fixing the model of a particular component may involve development work as well, depending on how complete the simulation framework is. For example, if the simulator implements a simple stride data prefetcher, a problem in modeling part of the memory subsystem may indicate that the hardware is using a different prefetcher. Another prefetcher should then be implemented in the simulator and provided as a configuration choice to the user. The tuning algorithm can then take that prefetcher and its possible configurations into consideration as it identifies the most accurate model.

B. Targeted Micro-Benchmarks

Our approach differs from recent work by Walker et al. [17] which relies on clustering techniques to group applications with similar modeling error, and then clustering performance-related events according to their impact on error. This is followed by regression analysis in an attempt to establish the relation between model error and a particular event, from which a candidate reason (or few reasons) for the modeling error can be identified. As mentioned earlier, our strategy is to isolate the modeling error in each component by relying on a suite of micro-benchmarks [30], each of which target a specific processor component. By doing so, identifying the sources of error in modeling becomes easier. For example, micro-benchmarks that target the first cache level show whether the L1 cache is modeled accurately or not. Similarly, benchmarks that target direct and indirect branch behavior can expose errors in the branch predictor model.

The micro-benchmark suite that we use for the tuning phase contains a set of 40 micro-benchmarks that can be classified into five categories: (1) control flow, (2) data-parallel and floating-point operations, (3) execution with stress on inter-instruction dependencies, (4) memory operations stressing various levels of the hierarchy, and (5) store-intensive operations [30]. Table I provides a list of the micro-benchmarks and the number of dynamically executed AArch64 instructions for the main loop. The control flow benchmarks stress the branch unit in various scenarios such as easy-to-predict branches, heavily biased branches, randomized flow, branches with large flush penalty, indirect branches, etc. The data-parallel benchmarks evaluate cases with data parallel loops that involve double and float operations and conversions. The complexity of the computations involved varies across the benchmarks as well. The benchmarks focusing on the execution units involve integer and floating-point operations that vary in complexity. Each of these benchmarks involve chains of dependencies of variable length. The benchmarks that stress the memory hierarchy involve access to data sets that reside at various levels of the hierarchy, access with plenty of conflict misses, linked list traversal at different cache levels or in memory, stressing instruction cache misses, and load-store dependencies. In general, these benchmarks provide a diverse substrate to comprehensively evaluate the accuracy of the simulation model, and to increase the probability of finding an accurate configuration using the proposed tuning algorithm.

In addition to their ability to isolate errors in individual components of the processor, micro-benchmarks are an enabler for the automatic validation and tuning process. Each round of tuning involves thousands of simulation runs to make a comprehensive test that touches all the benchmarks with a statistically significant set of all the possible configuration permutations. The relatively small number of instructions, compared to SPEC or PARSEC, allows evaluating tens of thousands of configurations within a span of a few hours.

C. Racing

We leverage algorithms used in machine learning to automatically validate a simulator against real hardware. More specifically, we rely on irace [31] to tune the configuration parameters to reduce the specification errors in simulation. irace is an implementation of iterated racing [18] in the widely used statistical software package R. The algorithm takes the configurable parameters along with their candidate values as its input. In most cases, evaluating all possible permutations of configuration parameters is computationally unfeasible. Therefore, the algorithm stops after a configurable maximum number of trials. To make the most out of the maximum number of trials, racing algorithms depend on fast elimination of configurations that can be statistically proven to be inferior to others.

The iterated sampling algorithm involves three main steps, as demonstrated in Figure 2: (1) sampling new configurations

TABLE I
MICRO-BENCHMARKS USED IN OUR METHODOLOGY ALONG WITH THEIR NUMBER OF DYNAMICALLY EXECUTED AARCH64 INSTRUCTIONS.

Memory Hierarchy														
MC	MCS	MD	MI	MIM	MIM2	MIP	ML2	ML2_BWld	ML2_BWldst	ML2_BWst	ML2_st	MM	MM_st	M_Dyn
1.8M	115K	33K	22M	5.25M	214K	66M	131K	3.15M	107K	8.4K	164K	1.05M	1.97M	1.5M
Control Flow														
CCa	CCe	CCh	CCh_st	CCl	CCm	CF1	CRd	CRf	CRm	CS1	CS3			
82K	657K	2.6M	157K	1.38M	656K	1.27M	599K	133K	399K	58K	34.5M			
Data Parallel														
DP1d	DP1f	DPcvt	DPT	DPTd										
5.2M	5.2M	36.7M	542K	1.18M										
Execution														
164K	451K	5.24M	65K	328K										
Store Intensive														
STL2	STL2b	STc												
4K	1.12M	400K												

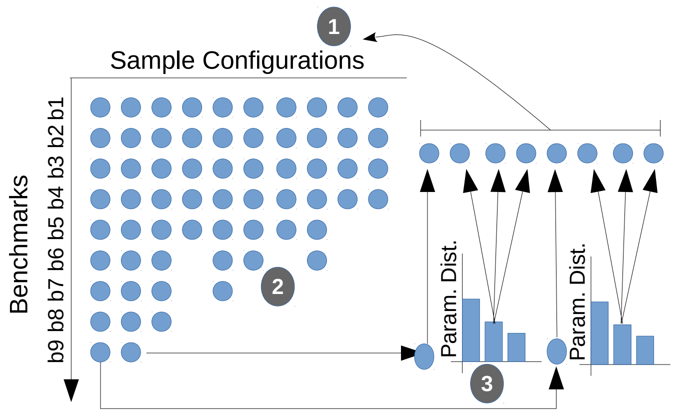

Fig. 2. An overview of the irace parameter tuning algorithm.

according to a specific distribution, (2) choosing the best configurations out of the sampled ones via a racing technique, and (3) updating the distributions to bias future configuration sampling towards the best ones. The process repeats for a maximum number of trials.

Each configuration parameter is associated with a sampling distribution that determines the probability of selecting a certain value for that parameter. Initial sampling assumes all values have equal weights. As the algorithm starts finding winning configurations, it updates the distributions associated with each parameter. Updating the sampling distribution involves biasing the weights to increase the probability of selecting the right value for each parameter.

Once a set of new configurations is sampled in step #1, step #2 finds the best configurations from the sampled ones. In Figure 2, the set of sampled configurations is represented by filled circles. The racing algorithm starts evaluating the configurations on the set of micro-benchmarks in turn, shown along the vertical axis in Figure 2. As the figure shows, the whole set of configurations is evaluated against the first few benchmarks; the algorithm then starts making statistical tests to quickly eliminate the configurations that perform worse than at least one other configuration. The algorithm keeps evaluating survivor configurations against the micro-benchmarks, while increasing the frequency of elimination. The final set of survivor configurations are used to update the parameter sampling distributions in step #3 and are also propagated among the newly sampled configurations. The process repeats until the maximum number of optimization trials is reached.

In the context of our framework, irace takes the following inputs: (1) a set of parameters and a list of possible values each parameter can take, (2) a set of micro-benchmarks to simulate in Sniper for performance assessment, (3) the performance values measured on real hardware for each micro-benchmark, and (4) a tool to calculate the desired error metric being minimized, i.e., CPI error in our case. The list of parameters include all the parameters that the user has to make a best-effort estimate for due to lack of reliable disclosed information. The user has to manually provide this list. The second input to the tool comes in the form of instruction traces of every micro-benchmark in the suite. The benchmark traces are generated on the real hardware platform only once. Similarly, the third input requires evaluating the performance of each micro-benchmark on the real hardware platform only once. For example, if the user tries to model the ARM Cortex-A72 core, the tuning algorithm expects the performance results for each of the benchmarks on a real Cortex-A72 processor. Once these inputs are ready, irace launches several simulation experiments in parallel by sampling the configurations and evaluating them for each benchmark, as explained earlier in Figure 2. As mentioned earlier, we implement a framework to facilitate trace generation and performance number collection on real hardware, in addition to scripts that help irace launch simulations, extract performance measures from the simulations, and calculate the error cost that irace uses to determine which configurations to survive and which to eliminate. The final output of the algorithm is the best candidate configurations given the accuracy of the simulator model.

Several factors determine the turnaround time of irace. The higher the number of configuration parameters and the number of possible values per parameter, the longer it takes to find an optimal configuration. However, because exploring all possible permutations is computationally intractable, the user can define criteria to terminate the tuning process, e.g., a minimum number of remaining survivor configurations, a maximum finite time, or a maximum number of iterations. In this work

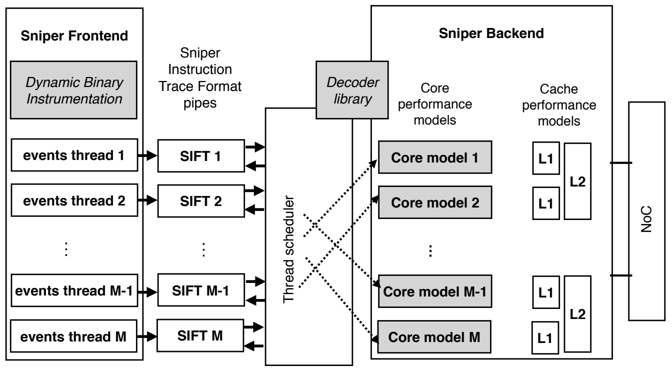

Fig. 3. An overview of the main components that were modified in Sniper to support the ARM AArch64 ISA and to model the in-order Cortex-A53 and out-of-order Cortex-A72 cores.

we instruct irace to search for an optimal configuration with a maximum budget of 100 K trials. The length of the simulated application also determines the duration of the simulation, and consequently the tuning process time. We take this factor into account by selecting a suite of micro-benchmarks that can be simulated in subseconds. Fortunately, the experiments in irace can be parallelized. Hence, the duration of the tuning algorithm can be expedited using a high core count and processor operating frequency to run irace iterations. For this work, we run irace on a 12-core processor with hyper-threading (i.e., 24 hardware contexts) running at 2 GHz. Budgets of 10 K and 100 K trials finish on this platform in about seven hours and two days, respectively.

IV. Sniper-ARM Simulator

We demonstrate our simulator validation methodology by implementing the necessary components to support the ARM AArch64 ISA in Sniper, along with the timing models for in-order and out-of-order processors. We use our methodology to configure the in-order microarchitecture according to the ARM Cortex-A53 processor [32] and the out-of-order microarchitecture according to the Cortex-A72 processor [33].

A. Modifications to Sniper

Figure 3 depicts a high-level overview of the main components of Sniper that need modification to (1) support the ARM AArch64 ISA, and (2) model in-order and out-of-order ARM core types.

The simulator front-end is responsible for instrumenting the ARM code, decoding the instructions, and feeding the timing model with a dynamic stream of instructions. This component is necessary to simulate unmodified AArch64 code. Sniper was originally built around Pin [34], a dynamic binary instrumentation tool for x86 processors. Sniper relies on the *X86 Encoder Decoder (XED)* libraries to provide a stream of micro-operations to the back-end of the pipeline. To support AArch64, we replace the front-end interface with DynamoRIO [35], which is a dynamic binary instrumentation tool that provides similar functionality to Pin for the ARM AArch64 ISA. We also replace the x86 decoder libraries with Capstone [36].

Note that because DynamoRIO is a dynamic binary instrumentation tool and does not perform cross-ISA instrumentation, we need to run the Sniper-ARM front-end on a real ARM processor. However, we provide a solution that allows porting the traced ARM code and simulate it on x86 server machines. We integrate the DynamoRIO tool with Sniper's front-end to allow for recording instruction traces in the Sniper Instruction Trace Format (SIFT), a format readable by Sniper's back-end. This way, a representative part of the benchmark needs to be recorded only once, and can be re-used for all future timing simulations as we evaluate different ARM core microarchitecture configurations on x86 servers.

The second part that we modify is the contention model in Sniper's timing model. The contention model defines the functional units in the processor and assigns every instruction to its corresponding functional unit. The contention model is responsible for ensuring that a functional unit is available when issuing an instruction. A unit may be unavailable in a given cycle if it is occupied by other instructions, another instruction has precedence to use the unit if issued at the same cycle, or if the issue slot for this unit is occupied. Moreover, the contention model verifies that instructions issued in the same cycle are compatible, or can be dual-issued. We model the contention models for the in-order and out-of-order cores according to the information disclosed in the technical reference manuals for the Cortex-A53 and Cortex-A72 processor cores, respectively.

Finally, the last part of the timing model that we modify in Sniper is the core timing model. This part of the model defines the general organization of the pipeline and its configuration parameters, including the cache hierarchy (number of cache levels, the respective sizes, etc.), the reservation stations and their sizes, the ROB, processor frequency, etc. Where appropriate, we augment the existing models with more options to mimic potential microarchitectural features that are usually not disclosed. As an example, we implement mask-based, xor-based, and Mersenne modulo [37] address hashing for cache indexing. In addition to generic pipeline components, the core timing model also includes specialized structures that are not necessarily implemented in every processor. For example, the branch prediction unit is a specialized component that can be totally different across different processors. Similarly, data prefetchers and their configurations at the various levels of the hierarchy vary among different processors. Whereas few generic pipeline configuration parameters may be disclosed, the configuration of specialized components are usually not disclosed at all. These components are ideal candidates for automated tuning. For these components, we implement an assortment of techniques, each of which is further parameterized. We feed these components into the tuning algorithm to automatically determine the component and configuration that best matches the real hardware measurements.

Sniper features a couple hundred configuration parameters. Several of these parameters are necessary to define the

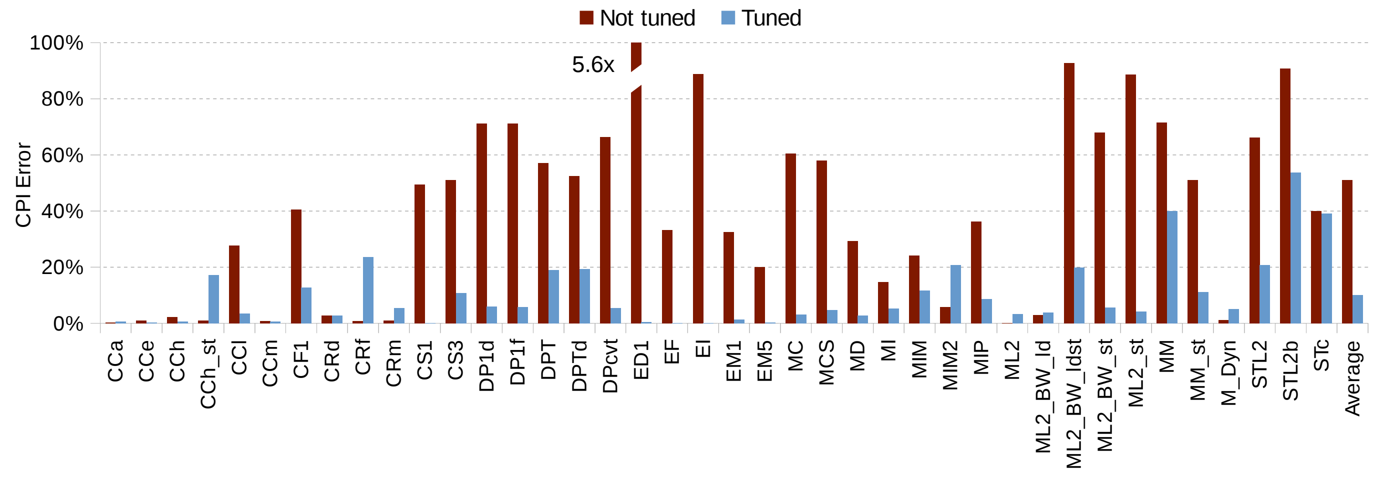

Fig. 4. CPI prediction errors for the microbenchmarks for the Cortex-A53 processor before and after tuning.

simulation environment, e.g., application scheduling, and are not involved in the core timing model. We count about a hundred parameters that define the simulated processor. Out of these parameters, we identify 64 parameters that cannot be accurately adjusted using publicly disclosed information or via latency estimation using lmbench. These parameters are passed to irace. The parameters vary in the number of values they can possibly take. There are parameters that require a binary true or false value such as whether to use a prefetcher or not, or whether to prefetch after a prefetch hit. Other parameters can take on a relatively large number of possibilities. For example, parameters that define the number of entries in the load/store unit or the size of the active instruction window. irace can take a range of values (e.g., 16 to 164) but to avoid wasting irace's budget, these parameters are given a limited set of discrete values. Other parameters can assume a discrete set of parameters to select a particular feature, e.g., which branch predictor to use. The list of Sniper parameters we pass for tuning includes pipeline and cache hierarchy configuration parameters. For example, we pass parameters for the reservation station configuration, branch misprediction penalty, window size, cache bandwidth configurations, victim cache entries, serial and parallel tag and data access in cache, among others. Other parameters tune system configurations, e.g., the main memory organization, and access latency and bandwidth modeling parameters.

B. Fine-Tuning the Timing Model

We now briefly highlight the added value of our proposed validation methodology to improve the timing model in Sniper, and, to our own surprise, uncover coding bugs in the decoder libraries we use. The discussion we provide in this section shows results for the in-order Cortex-A53 core validation effort. Validating the out-of-order Cortex-A72 core model proceeds similarly.

Our initial configuration of the in-order core uses all the information we could gather regarding the Cortex-A53 from the technical manual in addition to the cache latency parameters derived using lmbench. This is equivalent to completing step #3 in our proposed validation methodology, see Figure 1. Figure 4 reports absolute CPI errors for the initial Cortex-A53 model for the complete set of micro-benchmarks: the average error approaches 50% with errors reaching up to more than 5×, see for example the ED1 micro-benchmark.

At this point of the validation process, it is not clear whether the errors are to be attributed to abstraction errors in the new model, bugs in any of the used libraries, or specification errors (inaccurate parameter settings). However, a round of parameter tuning in step #4 of our validation methodology, alleviates the impact of inaccurate parameter tuning. Indeed, after the first tuning round (results not shown), the average error drops significantly, to about 33% across all micro-benchmarks, and the outlier errors in ED1 are trimmed to around 33% as well.

Beyond this point, we seek to improve the average error even further. We observe after the first tuning round that several micro-benchmarks still suffer from a high CPI prediction error. A couple control-flow intensive benchmarks still indicate high errors as in CS1, which simulates a `case` statement that benefits from indirect branch support. A recent work acknowledges a similar modeling issue [16]. We augment our model with indirect branch support and provide further flexibility of choice for the tuning algorithm to fine-tune the predictor configuration. irace did not significantly improve the modeling accuracy for the floating-point and data-parallel applications due to errors in modeling the timing and contention of the arithmetic instruction execution units. We also identify relevant bugs in the Capstone decoder library that led to errors in modeling dependencies across instructions. To alleviate the impact of modeling errors in micro-benchmarks that stress the memory subsystem, we provide the tuning algorithm with further options to use address hashing techniques to place and index blocks in caches, in addition to configurable prefetching options including stride [38] and GHB [39] prefetching. We also note that a couple memory-intensive micro-benchmarks access an uninitialized array, most of which are considered a cache miss by our model but are reported as hits on real hardware. We conjecture that in hardware the first access to an uninitialized OS page misses in the cache but further accesses to that page are recognized and the cache behavior is optimized to avoid the miss. Initializing the arrays prior to simulation dwarfs the error for these micro-benchmarks.

Figure 4 also shows the absolute CPI prediction error after fixing modeling errors and subjecting it to further rounds of parameter tuning using irace. The average absolute CPI prediction error is significantly reduced to around 10%. We conclude that the Cortex-A53 model in Sniper accurately models real hardware for the broad set of micro-benchmarks that stress different components in the processor.

V. Experimental Setup

We validate Sniper-ARM against the ARM Cortex-A72 and Cortex-A53. We model and validate these processors relying on the Firefly RK3399 development board, which is a platform that features a six-core 64-bit heterogeneous processor (i.e., big.LITTLE) [40]. The processor features one cluster of two 'big' high-performance Cortex-A72 cores, and another cluster with four 'little' low-power Cortex-A53 cores. Both cores include a 32 KB L1 data cache. The Cortex-A53 features a 32 KB L1 instruction cache, while the Cortex-A72 core features a 48 KB L1 instruction cache. The L2 cache is shared among the cores in each cluster and is coherent across the clusters. The 'big' cluster has a 1 MB L2 cache, and the 'little' cluster has a 512 KB L2 cache. The Cortex-A72 is configured to run at the maximum frequency of 1.99 GHz, while in the Cortex-A53 is configured to run at 1.51 GHz. Additionally, the board is equipped with 4 GB of DDR3 main memory. We use the Linux 16.04 Xenial operating system for this work.

We use two different sets of benchmarks in this paper. We rely on the complete set of micro-benchmarks proposed in microbench [30] during the parameter tuning phase. We show the accuracy of our approach at modeling the real hardware using SPEC CPU2017 benchmarks [41]. We evaluate a mix of both integer and floating-point benchmarks that are written in C/C++ (not Fortran) as Sniper implements methods for marking regions of interest in codes written in these languages. For each benchmark, we mark the code before and after the main loop of the benchmark. We compile the benchmarks on the Firefly board using the GNU C compiler v5.4.0. For the SPEC benchmarks, we compile and install the applications using the respective commands from SPEC. For the tuning process, we use *Perf* [42] on the board to gather all the relevant performance statistics. The tuning process reported in this work collects the number of dynamically executed instructions as well as the total number of cycles to calculate overall application CPI. We also generate instruction traces using the same exact part of the code. The traces are fed to Sniper's timing model for simulation.

We generate a representative code section for the SPEC benchmarks we use in this study. Contrasting between the CPI on the real hardware with that using our simulator requires using the same unit of work on both platforms. Due to the difficulty of isolating the performance of representative SimPoints [43] on hardware, we mark the main loops and we measure their performance. Due to the prohibitively long time it takes to simulate the SPEC CPU2017 benchmarks using their reference inputs, we rely on the train inputs and simulate few iterations up to the full execution of the application, reaching billions of simulated instructions for most benchmarks. Table II provides a lists of the applications and information on the starting location for evaluating each application as well as the total number of instructions evaluated. This information applies to both the hardware and the simulation platforms.

TABLE II
List of benchmarks and their dynamic instruction count.

Benchmark	File name	Insn Count
mcf	psimplex.c, line 331	12 Billion
povray	povray.cpp, line 258	2.45 Billion
omnetpp	simulator/cmdenv.cc, line 268	10.8 Billion
xalancbmk	XalanExe.cpp, line 842	443 Million
deepsjeng	epd.cpp, line 365	14.9 Billion
x264	x264_src/x264.c, line 173	14.8 Billion
nab	nabmd.c, line 127	14.2 Billion
leela	Leela.cpp, line 62	10.3 Billion
imagick	wang/mogrify.cpp, line 168	13.4 Billion
gcc	toplev.c, line 2461	9 Billion
xz	spec_xz.c, line 229	10.8 Billion

VI. Validation Results

We show the accuracy of our hardware-validated simulation model against both the Cortex-A53 and Cortex-A72 cores. We further demonstrate how even best effort in tuning the model can lead to severe modeling errors, further motivating the need for an automated validation methodology as proposed in this paper.

A. Model Accuracy

Figure 5 reports the per-benchmark absolute CPI prediction errors for the in-order Cortex-A53 Sniper-ARM model compared to real hardware. Our Cortex-A53 simulation model follows the real hardware quite accurately. We report an average absolute CPI prediction error of 7%, and at most 16% for a single benchmark.

Similarly, Figure 6 shows the absolute CPI prediction error for the out-of-order Cortex-A72 core. We report an average absolute CPI prediction error of 15% with a couple outlier applications reaching up to approximately 30%. Further analysis reveals that for povray and x264, more than half the modeling error can be attributed to the prefetcher. For more than half of the benchmarks, the modeling error is less than 10%.

B. Impact of Modeling Errors

Minor modeling mistakes that come from specification errors can lead to significant errors in the estimated performance despite the best estimates from users. To demonstrate how significant the error can be in practice, we determine the worst processor configuration in very close proximity to the true optimum (as determined by our validation methodology). We

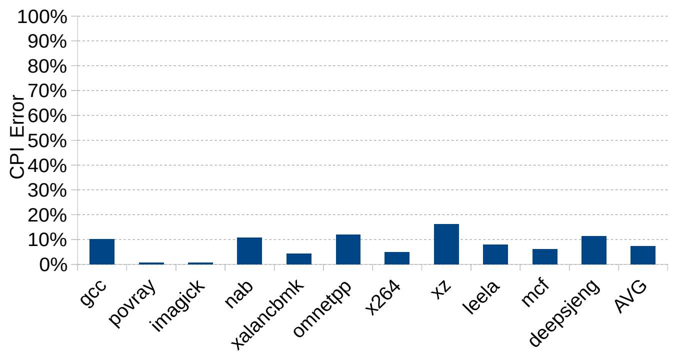

Fig. 5. Absolute CPI prediction error for the in-order Cortex-A53 Sniper-ARM simulation model compared to real hardware.

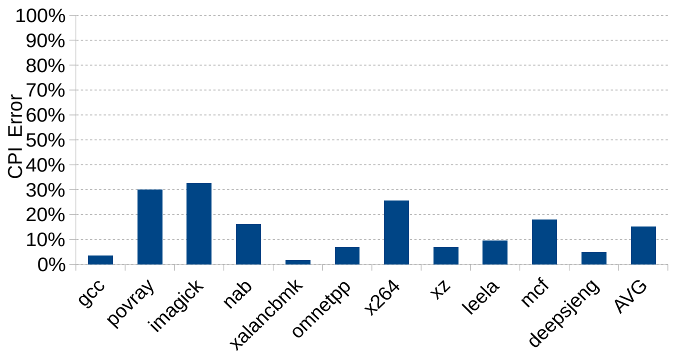

Fig. 6. Absolute CPI prediction error for the out-of-order Cortex-A72 Sniper-ARM simulation model compared to real hardware.

start from the optimum configuration and find the worst configuration that results from giving each configuration parameter a value that differs by a single step from the optimal. For example, if the possible values to configure the reservation station are 32, 40, 50, 64, and 70 entries, and irace determines that it should be 50 for minimum error, we evaluate the deviation of this parameter only to the values of 40 and 64 entries, leaving the rest of the possible values. We exhaustively search for the worst configuration that can be achieved with such a small deviation (including the deviation of multiple parameters simultaneously), and report the accuracy result for both the Cortex-A53 and Cortex-A72 models.

Figure 7 reports the absolute CPI error for the inaccurate Cortex-A53 model. The average error across all the benchmarks grows significantly from 7% to 34%. Similarly, individual applications yield errors that reach up to 67%. Relative to the best configuration in Figure 5, both the average error and highest error quadruple. Similarly, Figure 8 demonstrates the noticeable increase in CPI error for the Cortex-A72 model. The figure indicates that even with controlled deviation from an optimum configuration the average error reaches about 45%, i.e., the error increases by threefold compared to the average error of the accurately-tuned configuration.

Note that these results serve to provide an indication on how inaccurate parameter settings can lead to significant modeling errors, even when all the configuration values are within close proximity to the optimum. We expect worse modeling errors

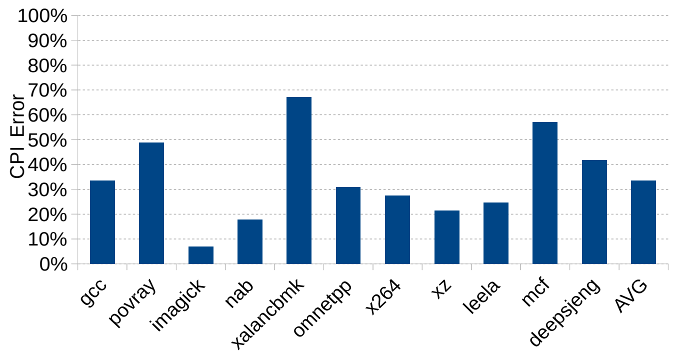

Fig. 7. Impact of close-to-optimum but inaccurate parameter settings on the Cortex-A53 model. Significant errors in performance evaluation can be seen even with reasonable parameter estimates.

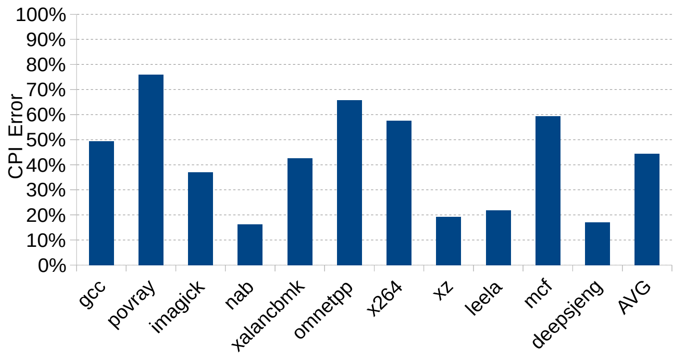

Fig. 8. Impact of close-to-optimum but inaccurate parameter settings on the Cortex-A72 model. Significant errors in performance evaluation can be seen even with reasonable parameter estimates.

when all possible parameter values are considered.

VII. Conclusions

Processor simulators are widely used in academia and industry for performance evaluation. Accurate assessment of the performance of a particular processor depends on how accurately the simulator models real hardware. However, validating processor simulators is a demanding endeavour, especially with the scarcity of publicly disclosed information on various aspects of the processor design. In this paper, we propose a validation methodology to alleviate the task of simulator validation. Our methodology relies on automated parameter tuning algorithms, based on iterated racing, to adjust the timing model of the simulator such that the error in performance evaluation is minimized. This approach vastly lessens the impact of *specification* errors resulting from lack of information on the real processor, and helps uncover *abstraction* modeling errors in the simulator. To demonstrate the effectiveness of our approach, we overhaul the Sniper simulator to provide support for the ARM AArch64 ISA, and we develop novel timing models for in-order and out-of-order ARM cores. We use our validation methodology to accurately model both an ARM Cortex-A53 in-order processor and an ARM Cortex-A72 out-of-order processor, with an average error of 7% and 15%, respectively, across a set of SPEC CPU2017 benchmarks.

ACKNOWLEDGEMENTS

We thank the reviewers for their constructive and insightful feedback. This work was primarily supported by the European Research Council (ERC) Proof-of-Concept Grant agreement No. 713632. Additional support is provided by the European Research Council (ERC) Advanced Grant agreement No. 741097, and the FWO projects G.0434.16N and G.0144.17N.

REFERENCES

[1] B. Black and J. P. Shen, "Calibration of microprocessor performance models," *IEEE Computer*, vol. 31, May 1998.

[2] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The gem5 simulator," *Computer Architecture News*, vol. 39, May 2011.

[3] T. F. Wenisch, R. E. Wunderlich, M. Ferdman, A. Ailamaki, B. Falsafi, and J. C. Hoe, "SimFlex: Statistical sampling of computer system simulation," *IEEE Micro*, vol. 26, Jul. 2006.

[4] M. T. Yourst, "PTLsim: A cycle accurate full system x86-64 microarchitectural simulator," in *Proceedings of the International Symposium on Performance Analysis of Systems and Software (ISPASS)*, Apr. 2007.

[5] D. C. Burger and T. M. Austin, "The SimpleScalar Tool Set," Computer Architecture News, 1997, see also `http://www.simplescalar.com` for more information.

[6] A. Patel, F. Afram, S. Chen, and K. Ghose, "MARSS: A full system simulator for multicore x86 CPUs," in *Proceedings of the Design Automation Conference (DAC)*, June 2011.

[7] R. Ubal, B. Jang, P. Mistry, D. Schaa, and D. Kaeli, "Multi2Sim: A simulation framework for CPU-GPU computing," in *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*, Sept 2012.

[8] P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. H. nad F. Larsson, A. Moestedt, and B. Werner, "Simics: A full system simulation platform," *IEEE Computer*, vol. 35, Feb. 2002.

[9] M. Rosenblum, E. Bugnion, S. Devine, and S. A. Herrod, "Using the SimOS machine simulator to study complex computer systems," *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, vol. 7, Jan. 1997.

[10] F. Bellard, "QEMU, a fast and portable dynamic translator," in *Proceedings of the Annual Conference on USENIX Annual Technical Conference (ATC)*, Jul. 2005.

[11] T. E. Carlson, W. Heirman, and L. Eeckhout, "Sniper: Exploring the level of abstraction for scalable and accurate parallel multi-core simulation," in *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis (SC)*, Nov. 2011.

[12] D. Sanchez and C. Kozyrakis, "ZSim: Fast and accurate microarchitectural simulation of thousand-core systems," in *Proceedings of the Annual International Symposium on Computer Architecture (ISCA)*, Jun. 2013.

[13] T. E. Carlson, W. Heirman, S. Eyerman, I. Hur, and L. Eeckhout, "An evaluation of high-level mechanistic core models," *ACM Transactions on Architecture and Code Optimization (TACO)*, Oct. 2014.

[14] R. Desikan, D. Burger, and S. W. Keckler, "Measuring experimental error in microprocessor simulation," in *Proceedings of the Annual International Symposium on Computer Architecture (ISCA)*, Jul. 2001.

[15] J. Gibson, R. Kunz, D. Ofelt, M. Horowitz, J. Hennessy, and M. Heinrich, "FLASH vs. (simulated) FLASH: Closing the simulation loop," in *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Nov. 2000.

[16] 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 the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, Mar. 2014.

[17] M. Walker, S. Bischoff, S. Diestelhorst, G. Merrett, and B. Al-Hashimi, "Hardware-validated CPU performance and energy modelling," in *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, April 2018.

[18] M. Birattari, T. Stützle, L. Paquete, and K. Varrentrapp, "A racing algorithm for configuring metaheuristics," in *Proceedings of the 4th Annual Conference on Genetic and Evolutionary Computation (GECCO)*, Jul. 2002.

[19] P. Bose, T. M. Conte, and T. M. Austin, "Challenges in processor modeling and validation," *IEEE Micro*, vol. 19, May 1999.

[20] R. Bell, Jr. and L. K. John, "Improved automatic testcase synthesis for performance model validation," in *Proceedings of the ACM International Conference on Supercomputing (ICS)*, Jun. 2005.

[21] F. Ryckbosch, S. Polfliet, and L. Eeckhout, "Fast, accurate and validated full-system software simulation of x86 hardware," *IEEE Micro*, vol. 30, Nov/Dec 2010.

[22] A. Butko, R. Garibotti, L. Ost, and G. Sassatelli, "Accuracy evaluation of gem5 simulator system," in *International Workshop on Reconfigurable Communication-centric Systems-on-Chip (ReCoSoC)*, Jul. 2012.

[23] M. A. Z. Alves, C. Villavieja, M. Diener, F. B. Moreira, and P. O. A. Navaux, "SiNUCA: A validated micro-architecture simulator." in *Proceedings of the IEEE International Conference on High Performance Computing and Communications*, Aug. 2015.

[24] L. Gwennap, "Digital 21264 sets new standard," *Microprocessor Report*, vol. 10, Oct. 1996.

[25] R. E. Kessler, E. J. McLellan, and D. A. Webb, "The Alpha 21264 microprocessor architecture," in *Proceedings of the 1998 International Conference on Computer Design (ICCD)*, Oct. 1998.

[26] M. Lattuada and F. Ferrandi, "Fine grain analysis of simulators accuracy for calibrating performance models," in *Proceedings of the IEEE International Symposium on Rapid System Prototyping (RSP)*, Jun. 2010.

[27] J. Jalle, J. Abella, L. Fossati, M. Zulianello, and F. J. Cazorla, "Validating a timing simulator for the NGMP multicore processor," in *Proceedings of the Data Systems In Aerospace Conference (DASIA)*, May 2016.

[28] A. Rodchenko, C. Kotselidis, A. Nisbet, A. Pop, and M. Lujn, "MaxSim: A simulation platform for managed applications," in *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, April 2017.

[29] L. McVoy and C. Staelin, "lmbench: Portable tools for performance analysis," in *Proceedings of the USENIX Annual Technical Conference (ATC)*, Jan. 1996.

[30] Vertical Research Group, "microbench: Extremely simple microbenchmarks," https://github.com/VerticalResearchGroup/microbench, 2014.

[31] M. López-Ibáñez, J. Dubois-Lacoste, L. Pérez Cáceres, T. Stützle, and M. Birattari, "The irace package: Iterated racing for automatic algorithm configuration," *Operations Research Perspectives*, vol. 3, 2016.

[32] ARM, "ARM Cortex-A53 MPCore processor, technical reference manual," https://developer.arm.com/docs/ddi0500/latest, 2018.

[33] ARM, "ARM Cortex-A72 MPCore processor, technical reference manual," https://developer.arm.com/docs/100095/latest, 2018.

[34] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: Building customized program analysis tools with dynamic instrumentation," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Jun. 2005.

[35] D. L. Bruening, "Efficient, transparent, and comprehensive runtime code manipulation," Ph.D. dissertation, MIT, 2004.

[36] N. A. Quynh, "Capstone: next generation disassembly framework," Blackhat USA, 2014.

[37] M. Kharbutli, Y. Solihin, and J. Lee, "Eliminating conflict misses using prime number-based cache indexing," *IEEE Trans. Comput.*, vol. 54, May 2005.

[38] J. W. Fu, J. H. Patel, and B. L. Janssens, "Stride directed prefetching in scalar processors," in *Proceedings of the International Symposium on Microarchitecture (MICRO)*, Dec. 1992.

[39] K. J. Nesbit and J. E. Smith, "Data cache prefetching using a global history buffer," in *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA)*, Feb. 2004.

[40] P. Greenhalgh, "Big.LITTLE processing with ARM Cortex-A15 & Cortex-A7: Improving energy efficiency in high-performance mobile platforms," http://www.arm.com/files/downloads/big_LITTLE_Final_Final.pdf, Sep. 2011.

[41] A. Limaye and T. Adegbija, "A workload characterization of the SPEC CPU2017 benchmark suite," in *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, April 2018.

[42] Linux Programmer's Manual, "Linux Perf," http://man7.org/linux/man-pages/man2/perf_event_open.2.html, 2018.

[43] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder, "Automatically characterizing large scale program behavior," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Oct. 2002.