Rigorous Evaluation of Computer Processors with Statistical Model Checking

Filip Mazurek
filip.mazurek@duke.edu
Duke University
Department of Electrical and Computer Engineering
Durham, NC, USA

Arya Tschand
arya.tschand@duke.edu
Duke University
Department of Electrical and Computer Engineering
Durham, NC, USA

Yu Wang
yuwang1@ufl.edu
University of Florida
Department of Mechanical and Aerospace Engineering
Gainesville, FL, USA

Miroslav Pajic
miroslav.pajic@duke.edu
Duke University
Department of Electrical and Computer Engineering
Durham, NC, USA

Daniel J. Sorin
sorin@ee.duke.edu
Duke University
Department of Electrical and Computer Engineering
Durham, NC, USA

ABSTRACT

Experiments with computer processors must account for the inherent variability in executions. Prior work has shown that real systems exhibit variability, and random effects must be injected into simulators to account for it. Thus, we can run multiple executions of a given benchmark and generate a distribution of results. Prior work uses standard statistical techniques that are not suitable. While the result distributions may take any forms that are unknown *a priori*, many works naively assume they are Gaussian, which can be far from the truth. To allow rigorous evaluation for arbitrary result distributions, we introduce statistical model checking (SMC) to the world of computer architecture. SMC is a statistical technique that is used in research communities that depend heavily on statistical guarantees. SMC provides a rigorous mathematical methodology that employs experimental sampling for probabilistic evaluation of properties of interest, such that one can determine with a desired confidence whether a property (e.g., System X is 1.1x faster than System Y) is true or not. SMC alone is not enough for computer architects to draw conclusions based on their data. We create an end-to-end framework called SMC for Processor Analysis (SPA) which utilizes SMC techniques to provide insightful conclusions given experimental data.

CCS CONCEPTS

• **Computing methodologies** → **Simulation evaluation**.

KEYWORDS

evaluation, statistical model checking, confidence intervals

ACM Reference Format:
Filip Mazurek, Arya Tschand, Yu Wang, Miroslav Pajic, and Daniel J. Sorin. 2023. Rigorous Evaluation of Computer Processors with Statistical Model Checking. In *56th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '23), October 28–November 01, 2023, Toronto, ON, Canada.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3613424.3623785

1 INTRODUCTION

Computer architects, in academia and industry, develop new designs and features and seek to compare them to existing processors. The vast majority of evaluations are experimental, rather than purely analytical, and they involve running benchmarks on simulators and/or real hardware. Architects use the experimental results to determine which ideas and products are better than others.

Unfortunately, the standard practice for evaluating experimental data has shortcomings, and architects are liable to make incorrect conclusions regarding the relative merits of different processors. The underlying issue is *experimental variability*. When a real computer runs the same program multiple times, the runtimes (and other metrics) will all differ. Some of this variability depends on the hardware state, such as the contents of the branch predictor when the program begins (or resumes after being context switched back in). Other variability depends on other software currently running on the system, including full-fledged applications and kernel processes. Previous work has identified variability due to seemingly innocuous reasons, such as program linking order [31] or even bugs in the Linux kernel [29]. Regardless of its source, variability can have a significant impact on results.

Alameldeen and Wood [3] identified the importance of variability. They demonstrated that adding tiny latencies to DRAM access times can cause significant differences in runtime for multithreaded workloads, and further showed that ignoring this variability could lead to incorrect conclusions. Consider running N simulations each of System X and System Y, where System X is almost always faster than System Y. If we were to take only one simulation datapoint from each set of N and compare those two, there is some possibility of choosing one of the slowest System X simulations and one of the fastest System Y simulations. The probability of reaching the wrong conclusion can be reduced with more simulations; intuitively, the probability of continuing to choose only outlier datapoints decreases.

To overcome the issue of variability, Alameldeen and Wood proposed performing multiple simulations for each experiment (i.e., for each target system and benchmark tuple) and processing the results with typical statistical methods, like confidence intervals and hypothesis testing [2]. While straightforward, the proposed solution has significant shortcomings and has largely not been adopted.

One major problem with the proposed solution is that it requires the *a priori* assumption of a Gaussian distribution of the results, which is not always true. In Figure 1, we show the runtime result distribution for a real (non-simulated) processor running the PARSEC benchmark *ferret* [5]. The data is clearly not Gaussian and thus any statistical analysis based on that assumption is suspect. The failure of the Gaussian assumption is also observed in recent works on performance evaluation of computer systems [10, 30, 32]. While results could have a Gaussian distribution, one cannot know that without running many trials. Many prior works assumed Gaussian distributions without sufficient validation (e.g., [44, 56]).

To remove the possibly flawed Gaussian assumption, the main challenge is how to make statistical inferences from experimental results sampled from distributions that can take any form and are completely unknown. While there exist statistical techniques that do not make *a priori* assumptions about data distributions, they have fundamental drawbacks that we discuss in Section 2.4.

To overcome the drawbacks of prior solutions, we introduce the technique of *statistical model checking (SMC)*, which allows rigorous evaluation for arbitrary result distributions, to the world of computer architecture. SMC is a statistical approach for system evaluation that is used in communities that depend heavily on statistical guarantees. Specifically, in the world of cyber-physical systems (CPS), system designers often must provide statistically rigorous guarantees for systems like robotics, avionics, and closed-loop medical devices [4, 40, 53]. For example, one might want to show with 99% confidence that a system will never be in an error state for more than two cycles.

We created SMC for Processor Analysis (SPA) to provide an end-to-end framework for computer architecture research evaluation, providing intuitive results with push-button ease of use. SPA leverages the powerful hypothesis testing methodology of SMC to create confidence intervals for properties of interest (e.g., cache miss rate, speedup). SPA takes as input the experiment data of interest along with the researcher's hypothesis, and it outputs a confidence interval for the result.

We make the following contributions in this work:

- We demonstrate that prior evaluation methodologies are insufficient and can lead to incorrect conclusions.
- We develop a mathematical basis that extends SMC to provide confidence intervals for metrics of choice (e.g., L2 miss rate).
- We develop the SMC for Processor Analysis (SPA) framework to enable push-button analysis of results.

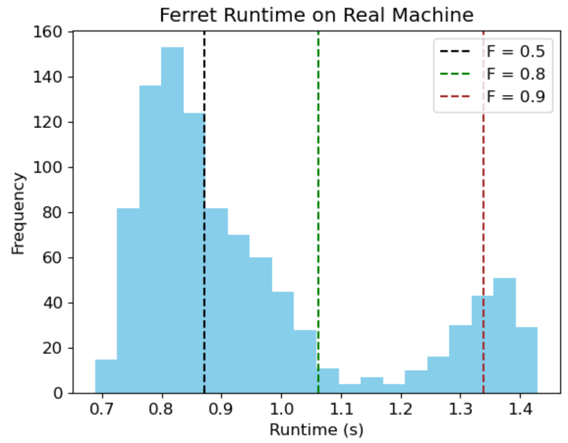

Figure 1: 1000 runtimes of *ferret* benchmark on real machine. Dashed lines are proportion values (discussed in Section 4).

- We show how to use SPA to achieve statistically rigorous results with a desired level of confidence on replicable simulation experiments with gem5.
- We publicly distribute SPA for use by academia and industry.[1]

2 BACKGROUND

Variability exists universally in computer systems and poses a major challenge to accurate evaluation of their performance [3]. In this section, we discuss the origins of variability, how to incorporate it into simulation experiments, the distinction between sample and population statistics, and prior approaches for evaluating the results of experiments that incorporate variability.

2.1 Origins of Variability

A process exhibits runtime variability due to three primary reasons: thread interleaving, scheduling decisions, and colocated processes. Other sources of variability exist—such as dynamic voltage and frequency scaling and address space layout randomization—but they tend to be less impactful.

Thread interleaving. Even a process with a deterministic functional outcome may have timing that varies from execution to execution. This phenomenon is most prevalent for multithreaded processes because of the different possible interleavings of thread executions. Consider a multithreaded application that synchronizes access to a shared data structure using a lock. The functional outcome of the application is independent of the order in which the threads obtain the lock, but runtime may differ.

Scheduling decisions. Because runtime scheduling decisions can vary, scheduling introduces variability. One extreme example is when a thread that is currently the bottleneck gets swapped out. Another example is scheduling a thread on a different core than it previously ran on [51]; the thread arrives to a "cold" core and cannot benefit from locality, as in a serverless system [55].

Impact of colocated processes. Any process will be impacted by the behavior of other processes currently running on the same machine. The processes might share the same multithreaded core

[1]https://github.com/filipmazurek/spa

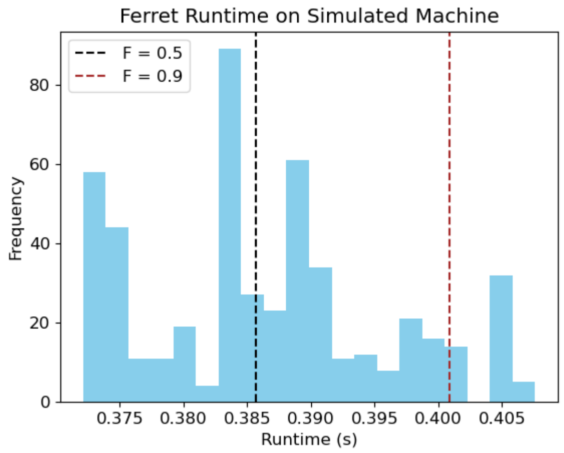

Figure 2: 500 simulated runtimes of the *ferret* benchmark on gem5 simulator with variability, using *simsmall* input.

or different cores of the same multicore processor. Regardless of how they share the machine, there will be shared resources (e.g., caches, ALUs, TLBs, etc.) for which they will contend. If a process with a large memory footprint is colocated with another process that also has a large footprint, its performance will be lower than if it were colocated with a process with a small footprint.

2.2 Incorporating Variability in Experiments

Variability exists in real-world experiments and thus should be incorporated into the experiments performed by computer architects. For simulation experiments, which are the focus of this paper, incorporating variability entails injecting it into executions. We next explain how we perform variability injection in our experiments, but a comprehensive exploration of how to perform variability injection is beyond the scope of this paper. Moreover, it is important to note that SMC's ability to analyze experimental results is independent of how variability is injected.

The primary challenges in variability injection are (1) injecting variability in a way that is controllable and compatible with repeatable, scientific experimentation; and (2) injecting variability that is representative of the expected variability.[2] We focus here on the first challenge; we leave the second challenge to future work.

If we inject variability into a simulation experiment, we must overcome the fact that the simulator is itself a deterministic program, so its execution is identical each time it is run from the same starting point. One of many possible solutions—and the one we use here—is Alameldeen and Wood's technique of adding small random latencies to DRAM accesses (a uniform distribution of $0 - 4ns$) [3]. These latencies are enough to perturb the execution and cause different thread interleavings. While this method is not necessarily capable of modeling all underlying sources of variability, it allowed us to observe considerable variability across various metrics using the gem5 simulator. As an example, the distribution of runtimes for the *ferret* benchmark is illustrated in Fig. 2.

[2]This problem is analogous to choosing benchmarks that are representative of the expected software workload.

2.3 Sample versus Population Statistics

Statistical evaluation of system performance with variability typically requires drawing samples from a population. While the variability in the samples can be captured by box plots (as in [18, 25]), box plots cannot accurately capture the uncertainties in the population due to random sampling error [9, 27]. Thus, rigorous statistical approaches are needed to draw correct conclusions about the population in terms of confidence intervals based on those samples. Admittedly, if there are many samples, then the sample distribution can well approximate the population distribution, and the boxes and confidence intervals may be almost the same. However, when there are few samples, the sample distribution may diverge significantly from the population distribution, and thus the sample variability indicated by the box plots can differ significantly from the population variability indicated by the confidence interval, leading to incorrect conclusions. In addition, when samples are scarce (e.g., fewer than 10), unlike confidence intervals, extrapolating for high quantiles (e.g., 95%) becomes challenging in box plots.

2.4 Prior Approaches for Non-Gaussian Data

There are two prior statistical approaches for addressing the issue of non-Gaussian distributions. One technique is based on non-parametric rank tests [10], in which every value in a distribution is ranked (i.e., placed in sorted order). Although computing the rank statistics does not need any assumptions, *comparing* the rank statistics requires the Gaussian assumption. Rank testing may be used to estimate the median and its confidence interval [26], but it faces problems with accuracy in cases where the sample population is small or contains many duplicates.

The other technique is statistical *bootstrapping* [30, 32], which resamples from a finite number of samples to extract statistical information of the unknown population distribution. Such a method is only asymptotically accurate when the sample size is large enough to cover the unknown distribution. It is likely to give incorrect answers when the sample size is small. The bias-corrected and accelerated (BCa) method of calculating the bootstrap confidence interval [21] is widely used due to its robustness, but may fail to produce a result if the sample contains duplicate data points.

While technically only used for Gaussian distributed data, we also consider the Z-score confidence interval [9] for comparison due to its frequent appearance in literature. A lot of data may be considered to be approximately Gaussian to make calculation easier, and these methods may be surprisingly effective in practice. We include this method in our comparison to check the importance of the underlying data distribution in computer architecture studies.

3 SMC TUTORIAL

The universal existence of variability and lack of rigorous evaluation of arbitrary result distributions motivates us to introduce Statistical Model Checking to the world of computer architecture. SMC provides a rigorous mathematical methodology to evaluate computer system performance from arbitrary result distributions with probabilistic guarantees. It can be applied to results from either hardware or simulator experiments without suffering from the weaknesses of prior approaches discussed in Section 2.4. We start with a description of SMC's capabilities and usage, before delving into its theoretical underpinnings. For more details on SMC, please refer to [1, 41]. Because existing SMC algorithms may not directly provide computer architects with their desired conclusions, we create an end-to-end framework called SMC for Processor Analysis (SPA) to fill the gap in Section 4.

3.1 Big Picture

From a statistical point of view, a system's evaluation metric $X(\omega)$ (e.g., runtime) is a random variable whose value varies across repeated experiments ω due to system variability. Due to variability, the probability distribution P_X of X can differ significantly across computer systems and software workloads. Ideally, to fully capture X, we need to present the full probability distribution P_X with all possible values of X in different cases. However, this process requires a huge number of experiments, which is infeasible in most cases. Thus, in practice, we seek to answer questions about certain statistical characteristics of the probability distribution.

Previous work has typically focused on using the expected value $\mathbb{E}X$ of an evaluation metric to characterize the metric's randomness (e.g., [54]). Experimentally, the expected value is estimated by taking N sample system executions and computing the sample average $\bar{X} = \sum_{i=1}^{N} X_i/N$. This approach seems natural as $\bar{X}$ converges to $\mathbb{E}X$ as N increases to infinity by the Law of Large Numbers. However, deriving the estimation error (i.e., the difference between $\mathbb{E}X$ and $\bar{X}$) in a statistically rigorous way for small N remains a challenging statistical problem when the probability distribution P_X is unknown. For simulation studies, the largest value of N we have found in the literature is only 10 [51, 58], with typical values 3-5 [8]. In some previous work [3], this issue is heuristically addressed by introducing the convenient, but possibly flawed, assumption that $\bar{X}$ obeys a Gaussian distribution.

Unlike prior work which estimates $\mathbb{E}X$, SMC statistically asserts *properties* of a computer system, which are binary functions that involve one or more evaluation metrics. A property is either true or false for a given system execution, and it may vary among executions due to variability in the system. A property can be simple, such as testing whether a metric (e.g., runtime, power) is greater than a threshold. Properties can also describe more sophisticated situations, such as whether the occurrence of an event will, with a probability greater than a threshold, keep the computer in a given state until the occurrence of another event. (For example, if we enter a sprinting state [52], with probability greater than X, we will stay in the sprinting state until a thermal alert.) A non-exhaustive list of property templates and concrete examples is in Table 1.

SMC can answer questions about the probability of a computer satisfying the above properties, i.e., whether a property φ (which is related to the performance of a computer system) will be true for at least $F \in [0, 1]$ fraction of executions. SMC is preferable to estimating the expected value for two reasons.

First, SMC allows rigorous statistical inference even if the randomness is unknown. Since the truth value of properties is binary, statistical inference techniques for binomial distributions can be applied without any assumptions. Given N sample executions that are derived from independent experiments, we can not only answer whether the probability of satisfying a property φ is greater than the threshold F, but also compute the confidence level $C \in (0, 1)$ of the answer. The confidence level guarantees that *out of all possible sets of N random executions, our answer agrees with the ground truth on at least a fraction C of them.*

Second, SMC provides greater insight than just an expected value. Consider the *ferret* benchmark run on a computer system that has the bi-modal distribution shown in Fig. 1. Due to variability, about 80% of executions have a runtime of less than 1.1 seconds, and about 20% executions have a runtime slower than 1.1 seconds. Using the expected value to characterize runtime cannot capture that in the majority of executions, runtime is faster than 1.1 s. SMC, however, can check if runtime is consistently faster than 1.1 s in at least 80% of cases and with a specified confidence of, say, 90%.

SMC offers two additional advantages that we note but leave for future work. First, SMC can handle a richer set of properties compared to existing methods. Until now, the computer architecture community has performed experiments that map only to the simple properties listed in Table 1. For example, we examined every paper published in ISCA 2022, and we found that every experiment could be mapped to rows 1-4 if using SMC. Second, SMC offers the ability to evaluate *hyperproperties* [12]. Whereas a property enables us to evaluate a collection of *individual* experimental executions, a hyperproperty enables us to evaluate multiple executions *taken together.* For example, SMC with hyperproperties enables us to study whether the performance of multiple executions will differ by less than a given threshold.

3.2 Theoretical Background

There is a long history of checking general properties on complex systems with *model checking*. Traditional model checkers, such as Murphi [19], PRISM [38], UPPAAL [16], and NuSMV [11], exhaustively explore the reachable state space of a system (i.e., finite state machine) and check whether specified invariants hold related to the properties of interest. Model checkers are widely used in many applications, such as verifying randomized algorithms [34, 39], wireless networks [20, 22], software/hardware security [17, 24, 48], performance/reliability [36, 46], robotic planning [23, 62], power management [35, 47], and biological processes [15, 37].

Model checking can handle general classes of time-related properties by expressing them in temporal logic, a formal symbolic language that can be parsed and understood by computer algorithms. Common logics include linear temporal logic (LTL) [50] and probabilistic computation tree logic (PCTL) [28]. Based on temporal logic, model checkers can exploit the relations between various specifications and systematically verify them.

Because traditional model checking suffers from the state space explosion problem, it is not tractable for many systems, including most systems of interest to computer architects. To overcome this limitation, SMC was developed to provide probabilistic guarantees—unlike the strict guarantees provided by model checking—based on sampling from the system. Given a property specified in a logic, a statistical model checker can automatically parse it into several sub-specifications based on the logic's syntax and semantics. Then, the model checker can verify whether these sub-specifications hold for a given system model by formulating them into a set of hypothesis testing problems, each of which is evaluated with sufficiently high probabilistic guarantees by cleverly deciding the number of samples

	Property Template	Examples

1	metric ≷ threshold	performance > A;
		power < B;
		MTTF > C;

2	threshold1 > metric > threshold2	A > performance > B;
		C > power > D;
		E > MTTF > F;

3	%time in state ≷ threshold	%time handling mispredictions < A
4	avg #cycles/event ≷ threshold	avg #cycles between TLB misses > A
5	metric1 ≷ threshold → metric2 ≷ threshold	power > A → performance > B
6	event1 occurs → Prob [event2 occurs within C cycles] ≷ threshold	if error occurs, probability of second error occurring within C cycles < P_B; if we power down component, probability of using it within C cycles < P_B;
7	event1's latency ≷ threshold1 → event2's latency ≷ threshold2	service time for request R > A → service time for request S > B
8	event1 occurs → Prob [stay in state until event2] ≷ threshold	if we enter sprinting state, probability of staying there until thermal alert < P_A
9	Prob [event when Prob[state] ≷ threshold1] ≷ threshold2	Prob [new TLB miss when Prob[handling old TLB miss] > P_A] < P_B

Table 1: Non-exhaustive list of properties that one can evaluate with SMC

needed on-the-fly. Finally, these results are combined to yield an assertion for the full specification with probabilistic guarantees. Such a statistical approach reduces the computational cost compared to exhausting all possible system executions—the approach used for symbolic model checking.

3.3 Technical Description

We now explain the mathematical underpinning of SMC. Mathematically, our goal is to use SMC to check if a property φ holds on a computer system S with probability greater than F:

$$p_\varphi := \mathbb{P}_{\sigma\sim S}(\varphi \text{ holds on } \sigma) \geq F, \tag{1}$$

where σ is a random execution from the computer system with variability, $\sigma \sim S$ indicates that σ is "drawn" from the system S, and φ is the given property of interest. Effectively, we want to know if the probability p_φ that an execution σ of the system S satisfies the property φ is greater than F. Although SMC can handle more complex statements, we focus on (1) because it can cover most performance evaluation problems for computer systems.

For computer systems, common properties (e.g., the ones in Table 1) are expressible in signal temporal logic (STL) [45]. All STL formulas have well-defined semantics — they can always be parsed and yield a meaning without ambiguity [45]. This feature guarantees that SMC will never "misunderstand" a property specified in STL. For simplicity, we can view properties as binary random variables taking values $true$ or $false$ for each sample execution.

Checking (1) requires two steps. The first step is to parse the STL property φ and derive its true value for a sample execution σ. This process is standard, so we refer the readers to literature (e.g., [45] and subsequent work). The second step is to calculate the probability p_φ from (1). In model checking, the probability is computed from model knowledge; in SMC, the probability is estimated from sample executions. Here, we introduce the SMC method developed from [60, 63] based on the Clopper-Pearson Exact Method. Compared to alternative methods based on Sequential Probability Ratio Tests [1, 41], this method only requires a minimal assumption on the probability $p_\varphi \neq F$ in (1), which is rarely violated.

Our method works as follows. Consider a set of N sample executions $\sigma_1, \dots, \sigma_N$ that are taken independently from repeated experiments on the system S. For $i = 1, \dots, N$, with a slight abuse

Algorithm 1 SMC for $\mathbb{P}_{\sigma\sim S}(\sigma \models \varphi) \geq F$.

```
1: Input: Desired confidence level $C \in (0, 1)$.
2: Initialization: $N \leftarrow 1$, $M \leftarrow 0$, $C_{CP} \leftarrow 0$.
3: while $C_{CP} < C$ do
4:     Draw sample execution $\sigma_N$ and compute $\varphi(\sigma_N)$ by (2).
5:     $M \leftarrow M + \varphi(\sigma_N)$, $N \leftarrow N + 1$.
6:     Update $\mathcal{A}$ by (3) and $C_{CP}$ by (4) and (5).
7: return $\mathcal{A}$.
```

of notation, let

$$\varphi(\sigma_i) = \begin{cases} 1, & \text{if } \varphi \text{ is true on } \sigma_i, \\ 0, & \text{otherwise.} \end{cases} \tag{2}$$

Then, the total number of executions satisfying φ, which can be captured by $M = \sum_{i\in[N]} \varphi(\sigma_i)$, should obey the binomial distribution $\text{Binom}(n, p_\varphi)$, and the average statistics M/N is an unbiased estimator for p_φ. Intuitively, when $M/N < F$, we should assert *negative* to the condition (1); otherwise, we assert *positive*. Therefore, we define the statistical assertion for (1) based on the N sample executions as

$$\mathcal{A}(\sigma_1, \dots, \sigma_N) = \begin{cases} negative, & \text{if } M/N < F \\ positive, & \text{if } M/N \geq F \end{cases} \tag{3}$$

Due to the randomness of the sample executions, the statistical assertion $\mathcal{A}$ from (3) does not always agree with the ground truth. The statistical accuracy of $\mathcal{A}$ is captured by its confidence level C, which requires that

$$\mathbb{P}_{\sigma_1,\dots\sigma_N\sim S}\big(\mathcal{A} \text{ is } negative \mid \text{condition (1) is } true\big) \leq 1 - C,$$
$$\mathbb{P}_{\sigma_1,\dots\sigma_N\sim S}\big(\mathcal{A} \text{ is } positive \mid \text{condition (1) is } false\big) \leq 1 - C.$$

Intuitively, this means that out of all the possible sets of sample executions, the value of $\mathcal{A}$ agrees with the ground truth on at least a C fraction of them. If $C = 0.99$, we roughly expect at most 1 disagreement after using the statistical assertion 100 times (i.e., on the results of 100 executions). We also call $1 - C$ the *significance level*; this can be viewed as the maximum of Type I/II Errors or False Positive/Negative Rates from other statistics literature.

Specifically, we compute the confidence level of $\mathcal{A}$ with the widely used Clopper-Pearson Exact Method, i.e.:

$$C_{\mathrm{CP}}(a, b \mid M, N) = \tag{4}$$
$$\begin{cases} (1-a)^N - (1-b)^N, & \text{if } M = 0 \\ b^N - a^N, & \text{if } M = N \\ \mathcal{B}(b \mid M+1, N-M) - \mathcal{B}(a \mid M, N-M+1), & \text{else;} \end{cases}$$

here, $\mathcal{B}(\cdot \mid x_1, x_2)$ is the cumulative probability function of the beta distribution with the shape parameters (x_1, x_2), and the values of a and b are given by

$$\begin{cases} a = 0,\ b = F, & \text{if } M/N < F, \\ a = F,\ b = 1, & \text{if } M/N \geq F. \end{cases} \tag{5}$$

This method is statistically accurate for any sample sizes, while previous works on statistical evaluation of computer systems [10, 30, 32]) using Gaussian assumption or bootstrapping are only asymptotically accurate for large samples sizes. Moreover, this method is customized for the binomial random variable M and has the best sample efficiency among all statistically accurate methods [13].

SMC is designed to achieve any desired confidence level $C \in (0, 1)$ by running in a loop. It draws new sample executions and updates the confidence level C_{CP} using (4) and (5) until $C_{\mathrm{CP}} \geq C$. We can prove the following facts to justify this process (see [63]). First, with probability 1, the confidence level converges to 1 in this process; thus, this process always terminates. Second, whenever this process stops, the statistical assertion has a confidence level of *at least* C. Our SMC approach for checking (1) is in Algorithm 1.

4 SPA FRAMEWORK

SMC is a methodology for determining, with some specified confidence C and a proportion F, the truth value of a given binary (true/false) property. Because F refers to the probability a sample will satisfy the property, it can also be referred to as the *proportion* of the population that satisfies the property. For example, using SMC, an architect could specify a hypothesis using the property "the cache miss rate is less than 10%" with a proportion of "at least 80% of all possible executions", at a confidence of 90%.

This process is unlike what architects are accustomed to, though. Typically, an architect would run a (single) execution and observe that the cache miss rate is some specific number (say 8%), rather than presupposing the possible cache miss rate of 10%. Furthermore, unlike what architects do today, SMC logically operates in a loop, in which each iteration the SMC engine processes a new execution (i.e., simulation). Given the latency to perform simulation, such a loop would be prohibitively slow.

Consequently, to create a methodology that is compatible with how architects approach evaluations, we have developed a framework called *SMC for Processor Analysis (SPA)*.

We illustrate SPA in Figure 3. The user provides to SPA a metric of interest, a desired confidence level C, and a desired proportion F. The user may also optionally provide a batch size, b, which will be used to control the number of simultaneous simulator executions to improve total SPA execution time. The SPA wrapper then controls the SMC engine and the simulator, so the user does not have to.

SPA's three primary benefits over textbook SMC use, discussed in the next subsections, are (1) providing confidence intervals for

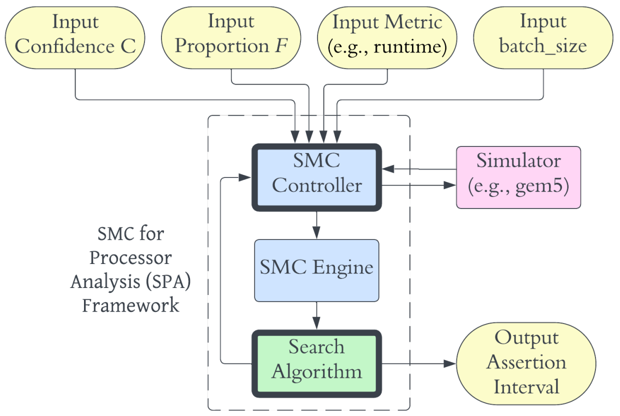

Figure 3: SPA Structural Diagram. The yellow ovals are inputs/output to SPA. The bold boxes are our contribution to facilitate use of SMC.

metrics, (2) performing the logic to control the SMC engine, and (3) improving performance with parallelism.

4.1 Confidence Intervals Using SMC

Existing SMC tests binary properties, but it does not provide confidence intervals. Confidence intervals are a range of values within which a population parameter, such as cache miss rate, is likely to fall with a specified level of confidence. We have extended SMC to do so, incorporating the process into SPA. This is a contribution of our work and, secondarily, enables apples-to-apples comparisons with prior statistical techniques such as bootstrapping.

First, we modify SMC to use a constant number of data points when performing hypothesis tests. That is, SMC will terminate only after testing every data point using (2), which removes the previous condition of $C_{\mathrm{CP}} > C$. After testing every data point, if $C_{\mathrm{CP}} > C$, SMC returns $\mathcal{A}$, but if the C_{CP} fails to reach a statistically significant level, the algorithm will return "None" instead. The changes to SMC are summarized in Algorithm 2.

This modification ensures that the same set of samples is used when testing different property thresholds, e.g., for the two properties "cache miss rate < 8%" and "cache miss rate < 10%". As a result, it is possible to directly compare the outputs at different property threshold values when using the same F and C.

By repeating SMC for different property threshold values on the same samples, we can construct a confidence interval for the true satisfaction probability in (1). Specifically, we can use SMC to find the property threshold values for which SMC returns $\mathcal{A} = \textit{positive}$ and $\mathcal{A} = \textit{negative}$. Then, we can compose a confidence interval with confidence C between any two hypothesis tests yielding opposing results with confidence greater than C [9]. We choose the narrowest possible construct to ensure the confidence interval is narrow enough to be informative.

Figure 4 illustrates this process. Each point in the plot is the resulting C_{CP} in the *positive* result, meaning that when $C_{\mathrm{CP}} < 1 - C$, the test is statistically significant for the *negative* result. We plot dashed lines on the plot at C and $1 - C$ where points above and below are statistically significant for the specified confidence C. Points between the dashed lines are where the SMC hypothesis test did not converge to a statistically significant result for the specified C (a result of "None"). In this case, the SPA hypothesis was $F = 0.9$ and $C = 0.9$, with different threshold values tested for Property 1 from Table 1. All hypothesis tests to the left of threshold 1.41 converge to a *positive* result, and converge to *negative* to the right of threshold 1.48. As a result, we find that the confidence interval is between 1.41 and 1.48 with confidence $C = 0.9$ for the speedup.

Algorithm 2 SMC for $\mathbb{P}_{\sigma \sim S}(\sigma \models \varphi) \geq F$.
Modified for a constant sample size.

```
1:  Input: Desired confidence level $C \in (0, 1)$.
2:  Initialization: $N \leftarrow 1$, $M \leftarrow 0$, $C_{\mathrm{CP}} \leftarrow 0$.
3:  while samples remain in $S$ do
4:      Draw sample execution $\sigma_N$ and compute $\varphi(\sigma_N)$ by (2).
5:      $M \leftarrow M + \varphi(\sigma_N)$, $N \leftarrow N + 1$.
6:      Update $\mathcal{A}$ by (3) and $C_{\mathrm{CP}}$ by (4) and (5).
7:  if $C_{\mathrm{CP}} > C$ then
8:      return $\mathcal{A}$
9:  else
10:     return None
```

4.2 SMC Engine Management

With SPA, the architect provides a metric and the SMC parameters C and F. SPA automatically generates the required property thresholds and controls the SMC process.

If the architect wants to directly provide a property, such as whether a metric is greater than a given threshold (e.g., "the cache miss rate is less than 10%"), she may specify that to SPA as well. In this case, SPA's task is trivial. It simply passes the property and the SMC parameters to the SMC engine and it runs a single SMC hypothesis test based on the unmodified SMC algorithm.

Most of the time, however, the architect's goal is not a simple property. Instead of wanting to know whether a metric is greater/less than a threshold, she wants to know the metric's confidence interval (e.g., the interval for the cache miss rate).

To determine a confidence interval for a metric, we use SPA to control the SMC engine. SPA makes an initial estimate of the metric, V_0, and creates the property "metric is at least V_0". It runs the SMC engine with this property and the desired confidence C and proportion F, and either validates or invalidates the property (i.e., asserts it is true or false, respectively), or does not have enough samples to converge. If the property was not invalidated, SPA reruns the SMC engine as before but with a metric estimate V_1 that is greater than V_0 by a user-defined granularity. SPA continues this search process until it identifies the smallest value of V for which the property is invalidated, V_{upper}. SPA similarly searches for the largest value of V for which the property is validated V_{lower}. In cases where the property used is differently stated, e.g., "metric is *no more than* V_0", we can still follow this procedure. However, V_{upper} will be the smallest value for which the property is *validated* and V_{lower} will be the largest value for which the property is *invalidated*. The terms "largest" and "smallest" in this context are not precise, but are rather a function of the granularity at which SPA searches.

Figure 4 illustrates the above process, where the user wants to evaluate how doubling the L2 cache size from 512kB to 1MB

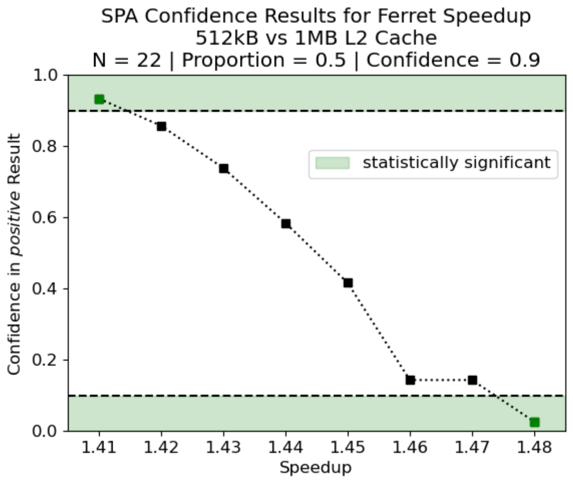

Figure 4: Each point represents the resulting confidence of an SMC hypothesis test for a *positive* result at the corresponding speedup. The points in the unshaded region did not converge and are thus included in the confidence interval. This procedure generates the SPA confidence interval in Fig. 5.

affects *ferret* benchmark runtime (i.e., speedup) with $F = 0.9$ and $C = 0.9$. Assume that $V_0 = 1.46$, and SPA has executed the simulator 22 times (calculated by equation 8, discussed later). SPA will first test $V_0 = 1.46$ to find that it does not converge with the data it was given. It will then test the next point, $V_1 = 1.47$ to reach the same conclusion. Upon testing $V_3 = 1.48$, SMC will converge to a $\mathcal{A}$ = *negative* result, setting $V_{upper} = 1.48$. SPA repeats the same process in the opposite direction to find that $V_{lower} = 1.41$. Therefore, SPA determines the confidence interval to be between 1.41 and 1.48 for $F = 0.9$ and $C = 0.9$. This final confidence interval is the SPA interval shown later in Fig. 5.

It is important to note that each iteration of the search does not require new simulations; rather, SPA will re-use the same set of simulation results as input to the SMC engine, as discussed in Section 4.1. Furthermore, if the architect decides that the interval V_{lower} and V_{upper} is wider than desired, she can decide to run more simulator executions, which may result in a narrower interval.

4.3 Improving Performance with Parallelism

Logically, SMC operates in a loop in which it (1) runs the simulator for one execution, (2) uses SMC to statistically analyze the results, and (3) uses SMC's confidence result to decide whether another execution is needed.

With SPA, we improve performance by running batches of simulations in parallel until at least the minimum number of samples are collected for SPA to calculate the confidence interval. The minimum number of executions for convergence is based only on the desired confidence C and proportion F of the property. As discussed in Section 3, the SMC engine's confidence level changes after each input according to (4). The fastest path to convergence for a $\mathcal{A}$ = *positive* output requires that each trial result is *true* and therefore follows the case that $M = N$. We denote N as $N+$ in this case, i.e.,

$$C \leq 1^{N+} - F^{N+}. \tag{6}$$

Table 2: Simulated system parameters

OS	Ubuntu 18.04
cores	4 out-of-order x86 cores
L1 I/D	I:32KB/2-way, D:32KB/8-way, 2-cycle
shared L2	inclusive 3MB/16-way, 16-cycle
cache block size	64B
Memory	3GB, 90-cycle
coherence protocol	MESI directory
on-chip network	crossbar with 16B links (same as flit size)

Similarly, the fastest path to convergence for a $\mathcal{A} = negative$ output requires all comparison results to be $false$, i.e., the $M = 0$ case in (4). Similarly, we denote N as $N-$ in this case. For the same confidence C and proportion F, the number of executions required to reach the $false$ conclusion satisfies

$$C \leq (1-0)^{N-} - (1-F)^{N-}; \tag{7}$$

Therefore, the minimum number of samples required by SPA is summarized by the following equation:

$$\max\{N+, N-\}, \tag{8}$$

where $N+$ and $N-$ satisfy (6), (7).

Thus, if given $C = 90\%$ and $F = 90\%$ as inputs, SPA finds that it requires 22 samples to converge to a $\mathcal{A} = positive$ result and 1 samples to converge to a $\mathcal{A} = negative$ result. Therefore to find the confidence interval, it requires at least 22 samples.

4.4 Choosing F and C

F is the likelihood (equivalent to the population proportion) that a property is true, and C is the confidence in the boolean SMC hypothesis result. Users should make a choice of F that is meaningful to their analysis. e.g., $F = 0.5$ considers the median performance value, while $F = 1.0$ provides a likelihood bottom bound.

5 EXPERIMENTAL METHODOLOGY

We choose to experiment with SPA on simulation data (instead of hardware data) to ensure replicability.

5.1 Simulator and Benchmarks

While SPA can analyze results regardless of their origin, we use gem5 v22.1, with the Ruby memory system simulator, due to its wide use in the computer architecture community [42]. We simulate the x86/Ubuntu multicore processor described in Table 2.

We run the PARSEC benchmarks with *simsmall* inputs [5], excluding *raytrace* due to its long simulation time and *vips* and *x264* because of compilation issues. We consider only the regions of interest, i.e., results exclude initialization and wrap-up code. We focus now primarily on *ferret*. This benchmark exhibits some of the greatest variability, due to frequent synchronization and data sharing.

5.2 Variability Injection and SPA Usage

As discussed in Section 2, there are several ways to inject variability. In the simulator, we add a uniform (pseudo-)random 0-4 cycles of latency to each L2 cache miss, similar to [3]. Each execution itself is deterministic, with the sequence of random numbers determined by a seed that we input. When evaluating speedups, we randomly take one execution from both the base and improved population runtimes and divide them to give a single speedup input to the SMC engine. For other metrics, we directly use the results from gem5.

5.3 Comparing to Ground Truth

For each benchmark, we run 500 simulations to determine the ground truth, where ground truth is defined as F of the entire population – i.e., the proportion of executions for which a property is true. For example, in Figure 1, the $F = 0.9$ value of 1.33 seconds is treated as the "correct" population value.

5.4 Comparing to Prior Statistical Approaches

We compare our novel SPA confidence interval (CI) construction against bootstrapping, rank testing, and Z-score CI construction.[3] Bootstrapping is the most relevant and popular statistical technique used in previous computer architecture studies [30, 32]. Rank testing has been used in previous studies and can be used to create confidence intervals [10, 26]. We include the Z-score method of creating the confidence interval [9] even though it requires the Gaussian assumption. In practice, these methods can still perform effectively if the population distribution is approximately Gaussian. Additionally, it is a demonstrative example of what may happen when the Gaussian assumption is incorrectly used.

To provide a fair comparison among the four techniques, we initially evaluate them for $F = 0.5$ (i.e., median) with a confidence level of $C = 0.9$. This is because rank testing and Z-score methods are best suited for estimating the median CI. For SPA, we specifically use property 1 in Table 1 because we are finding the threshold value for which the metric is true. For all other proportions $F \neq 0.5$, Z-score and rank testing methods are not suitable, so we later compare SPA against bootstrapping for $F = 0.9$.

Our objective is to evaluate the error probability of the constructed CI across 1000 trials, ensuring its compliance with the desired confidence. To achieve this, we check if the CI covers the ground truth value in every trial. Furthermore, we analyze the average CI width for all methods, as reduced error probability accompanied by a much broader CI may not be a worthwhile tradeoff.

In each trial, 22 samples are randomly drawn from the benchmark population, and the metric of interest is extracted. Using $F = 0.5$ and $C = 0.9$, each method constructs a CI which is compared against the calculated ground truth. If the CI covers the ground truth, that technique is counted to be accurate for that trial. This process allows us to determine the error probability and mean CI width for each technique. First, we calculate the mean width for each method by averaging the widths of the 1000 CIs it generated. Then, to allow comparison between different CI widths across metrics, we normalize these values by dividing the mean width by its corresponding ground truth value.

A sample of a single trial can be seen in Figure 5. In this case, the CIs constructed by SPA and the Z-score method cover the ground truth value, while the bootstrapping and rank testing CIs do not.

[3]We do not compare against error bars, which are based on the uncertainty of statistics within a sample. They are less statistically rigorous and no easier to use than the chosen CI methods.

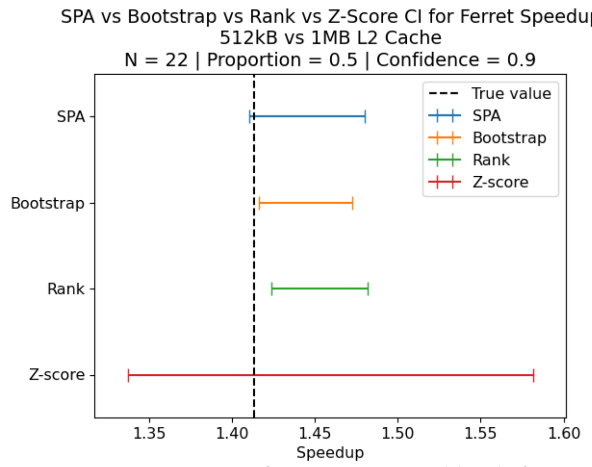

Figure 5: Comparison of CIs constructed by different techniques. Only SPA and Z-score CI cover population ground truth value. SPA gives tightest CI that covers true value.

Note that this is a case study and no conclusions are drawn from it. An accuracy evaluation is performed in the following section.

Regardless of the statistical technique used, the cost of running experiments dominates the cost of statistical analysis. Because the analysis runtimes are negligible compared to the overhead of running experiments, we do not compare their runtimes here.

6 EXPERIMENTAL EVALUATION

We show how to use SPA for statistically rigorous evaluations and compare its performance against other CI construction methods. We emphasize the importance of creating CIs that are not only narrow enough to be informative but also achieve the requested confidence level C. Using SPA for CI construction performs well by maintaining an error below the requested threshold and having a width comparable to the other investigated techniques.

SPA analyzes data regardless of its variability. In what follows, the coefficient of variation (standard deviation divided by the mean) ranges from 0.022 to 0.117 across metrics in the *ferret* benchmark and from 0.0002 to 0.127 across benchmarks for the metric *L1 Cache Misses per 1k Instructions.*

6.1 Evaluation at the Median

For a fair comparison among bootstrapping, rank-based, and Z-score CI construction methods, we test their error probability in estimating the median value ($F = 0.5$) with desired confidence $C = 0.9$ for selected metrics. Figure 6 presents the error probabilities, where probabilities exceeding the dashed line do not comply with the confidence threshold. In this case, only SPA and Z-score CIs are within the requested confidence C. In fact, the Z-score method is never incorrect, which is why it does not have any visible error bars in the figure. Bootstrapping sometimes fails to generate a CI, represented by the "Bootstrapping Null" stacked bar, discussed in Section 6.4.

The geomean error probabilities for each method are as follows: SPA has an error probability of 0.065, Z-score 0.000, bootstrapping 0.127, and rank testing 0.119. Despite the differences in error probabilities, because the CIs were constructed with a confidence level of $C = 0.9$, we cannot claim that Z-score is more accurate than SPA. As

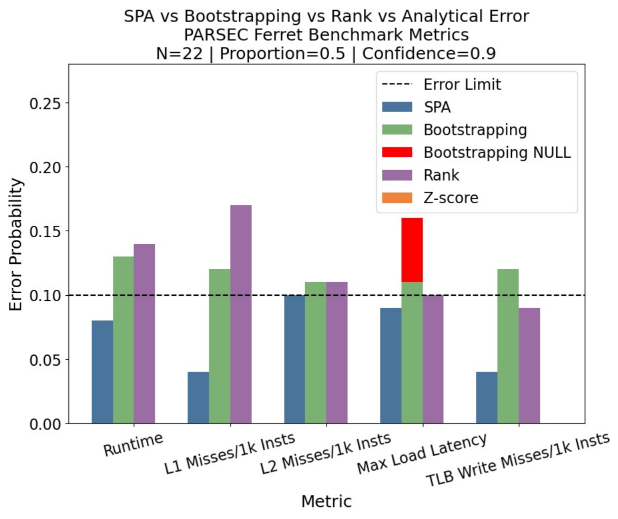

Figure 6: CI error probability for *ferret* over different metrics at $F = 0.5$. Average of 1000 trials. There is no Z-score error bar because it is accurate on every trial.

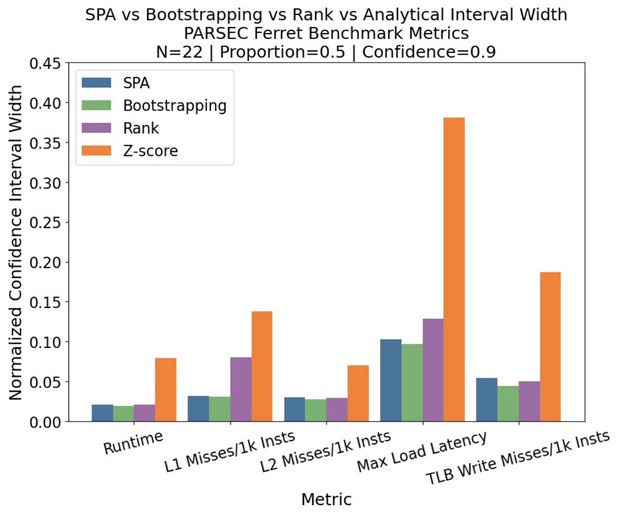

Figure 7: CI width for *ferret* over different metrics at $F = 0.5$. Average of 1000 trials.

both methods comply with the requested confidence threshold, they are considered equally correct in this case. Bootstrapping fails to comply with the confidence in every tested case, while rank testing is accurate for some metrics but highly inaccurate for others.

Despite Z-score CI construction seemingly performing as well as SPA in terms of error probability relative to the requested confidence, we check the CI widths in Figure 7. The Z-score CIs are 2.3-4.3 times broader than those of SPA, reducing their practical effectiveness. SPA's CI width is comparable to the other techniques while maintaining a much lower error probability.

6.2 Comparison at Other Proportions

For estimating proportions other than the median (any $F \neq 0.5$), only SPA and bootstrapping are suitable methods. We compare the two methods across various metrics and benchmarks while examining CI widths to ensure error probability differences are not

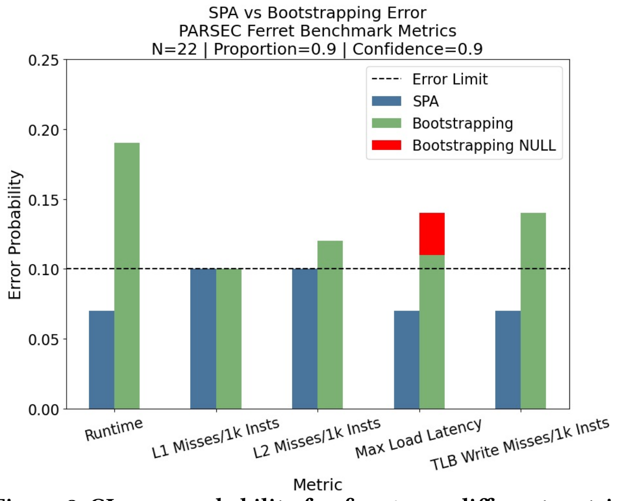

Figure 8: CI error probability for *ferret* over different metrics at $F = 0.9$. Average of 1000 trials.

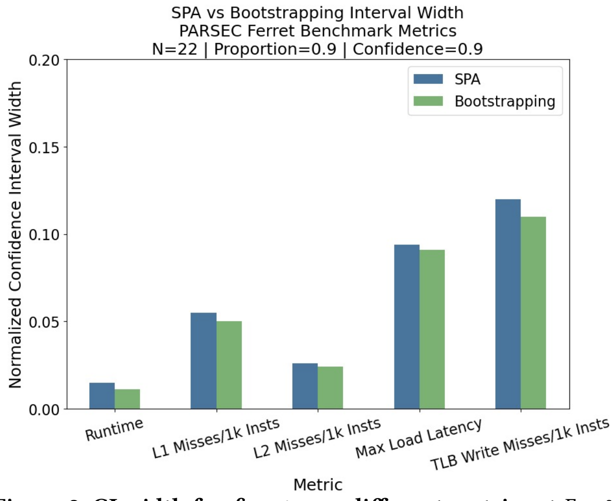

Figure 9: CI width for *ferret* over different metrics at $F = 0.9$. Average of 1000 trials.

a result of overly wide CIs. The following evaluation is conducted at proportion $F = 0.9$ and by constructing 90% confidence CIs.

6.2.1 Across Metrics. We evaluate the error probabilities of CIs constructed for several selected metrics from $ferret$ simulation runs, with the results illustrated in Figure 8. For each metric, the SPA CI achieves the desired error probability, while bootstrapping frequently yields error probabilities that exceed specification. The geomean error for the SPA CI is 0.081, while the bootstrapping CI has geomean error 0.135. Figure 9 shows that SPA achieves this error probability improvement with CIs that are only slightly wider than the bootstrapping CIs.

Even when duplicate data points cause bootstrapping to fail in generating a CI, the error probability—excluding these "Null" results—is above the confidence threshold. Note that the runs resulting in "Null" were neither re-run nor extended to find whether the resulting bootstrap CI would be correct or incorrect. It is likely that a portion of the "Null" results would be correct and a portion would be incorrect, implying that the error probability in the graph would likely be slightly higher. This suggests that SPA is more suitable for estimating values at percentiles other than the median.

6.2.2 Across Benchmarks. We observe the same trend when evaluating the metric *L1 Cache Misses / 1k Instructions* across eight PARSEC benchmarks. Figure 10 displays the error probabilities, with SPA's geomean error at 0.081 and bootstrap's at 0.135. With the exception of the *ferret* benchmark, the bootstrap CI error probability exceeds the specified error limit for all other benchmarks, with the error probability approaching 18% in the case of the *canneal* benchmark. In contrast, the SPA CI error probability has a more stringent limit, as it remains within the specified error bounds for all benchmarks.

The CI width comparison presented in Figure 11 tells the same story. SPA achieves compliance with the requested confidence with only a marginal increase in CI width over bootstrapping.

We also evaluate the error rate for a different metric, *L2 Cache Misses / 1k Instructions*, across eight PARSEC benchmarks. Figure 12 shows that the error probability again remains below the threshold

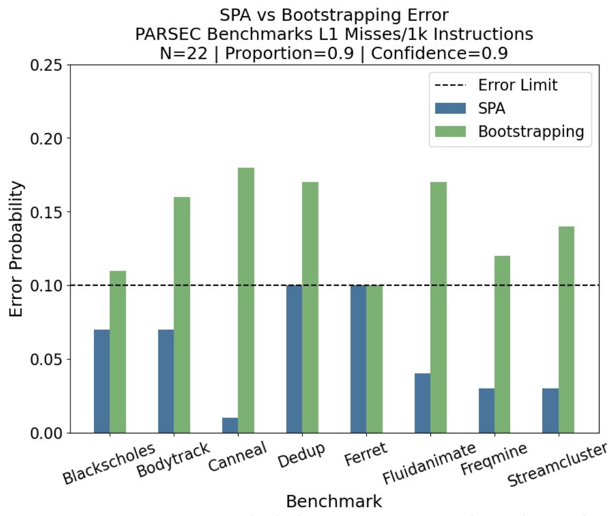

Figure 10: CI error probability over many benchmarks at $F = 0.9$. Average of 1000 trials.

only for SPA. The trend for CI widths is the same as before, with bootstrapping constructing a slightly narrower CI than SPA, as shown in Figure 13. This suggests that SPA may construct a better CI than bootstrapping in these cases, providing a preferable option for computer architects for statistical analysis. The robustness of SPA across various benchmarks and metrics further supports its potential as an effective method in computer architecture research.

6.3 CI Width Sensitivity to Confidence

To study the sensitivity of the CI width to the desired confidence level, we fix the proportion at $F = 0.5$ and vary the confidence from 90% to 99.9%, covering confidence values commonly used in research. For consistency, we use the four CI construction methods to estimate the $F = 0.5$ proportion of the *L1 Cache Misses / 1k Instructions* metric for the $ferret$ benchmark. For each confidence value, we conduct 100 trials; in each trial we randomly draw 22 samples to form a sample population and then construct a CI using each method. Afterwards, we take the mean of the 100 CIs for each

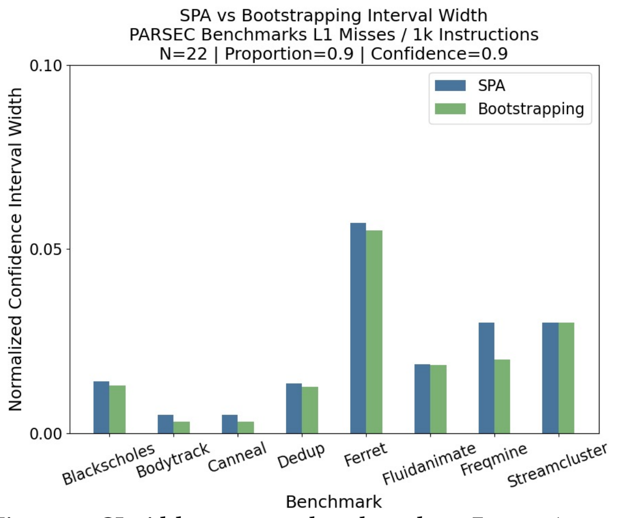

Figure 11: CI width over many benchmarks at $F = 0.9$. Average of 1000 trials.

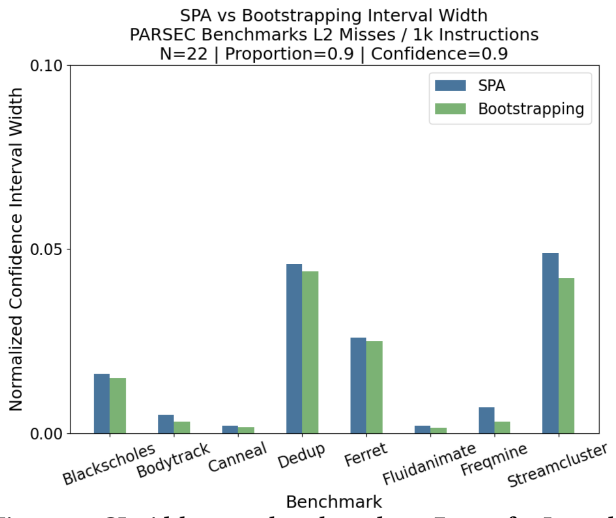

Figure 13: CI width across benchmarks at $F = 0.9$ for L2 cache miss probability. Average of 1000 trials.

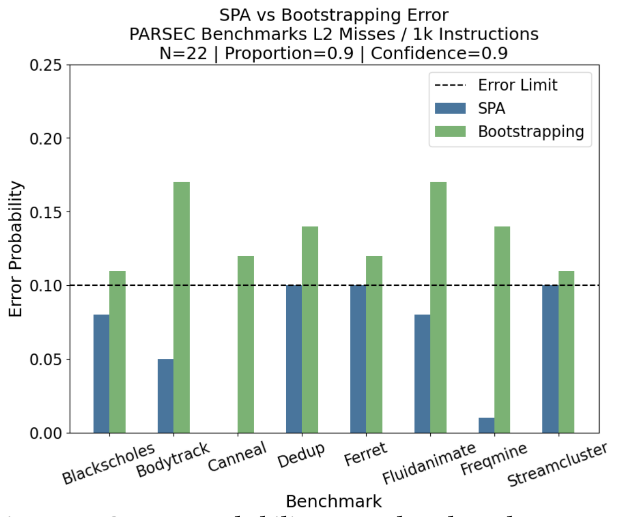

Figure 12: CI error probability across benchmarks at $F = 0.9$ for L2 cache miss probability. Average of 1000 trials. SPA has no error for *canneal*.

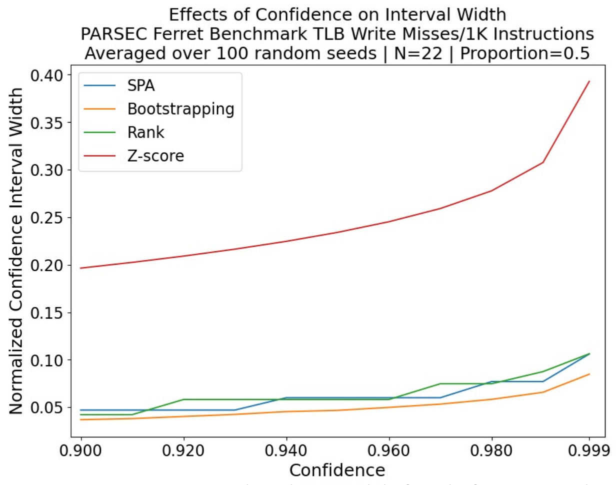

Figure 14: Mean normalized CI width for different methods over varying confidence values. Tested at the median ($F = 0.5$). From 90% to 99.9% confidence.

method, and then normalize by dividing the mean width by the ground truth value. We plot the results in Figure 14.

The CI widths of SPA, bootstrapping, and rank testing remain approximately the same size, though the bootstrapping CI remains the narrowest in all cases. The SPA CI remains wider than the one constructed by bootstrapping, though this is a worthwhile tradeoff for the lower error probability it offers. The Z-score CI remains considerably wider compared to the other methods.

6.4 Bootstrapping Failures

We create bootstrapping CIs using the "bias-corrected and accelerated" (BCa) [21] method, which offers better accuracy for non-Gaussian data. However, BCa struggles when there is an excessive amount of duplicate data in the sample population—leading to failure to generate any CI. This issue was observed when evaluating the CI for *Max Load Latency* at every proportion F at 90% confidence. At $F = 0.5$, bootstrapping failed to produce a CI in 5% of cases, while at $F = 0.9$, it failed in 3% of cases (indicated by red bars in Figures 8 and 10).

We further explore the issue of bootstrapping failing to produce a CI in cases of duplicate data points in the sample population. Because simulators produce results with very high precision, we only ran into the problem of the BCa method failing in one case—where the results were reported as integers. However, if we first round the simulator metrics to 3 digits past the decimal to eliminate "unreasonable" precision, we find that the large amounts of duplicate data in the population lead to frequent duplicates in the sample population, and therefore to bootstrapping's frequent failure to create a CI in most cases. We show these results in Fig. 15.

This limitation may pose a problem for researchers who seek to estimate values from metrics that may have repetitive data. In

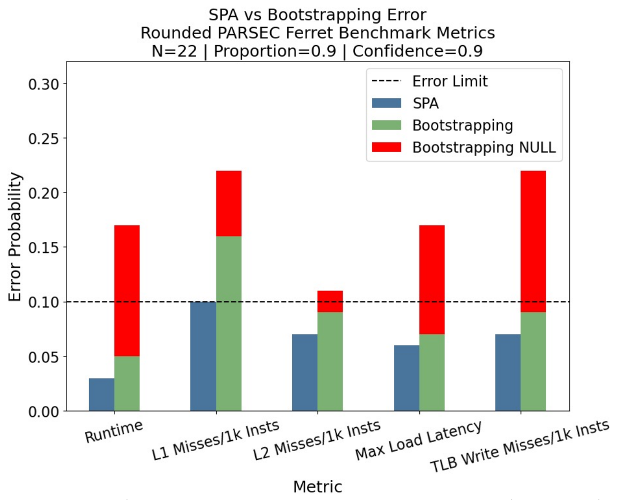

Figure 15: The same experiment as in Fig. 8, redone and resampled. All data was rounded to 3 places past the decimal. Duplicate data leads to bootstrapping failures.

contrast, SPA does not suffer from such limitations, as SMC treats each sample independently.

6.5 Number of Samples Used

Bootstrapping and rank testing rely on asymptotic statistics, thus risking significant inaccuracy if given few samples. We demonstrated in Figure 6 that they can be inaccurate even with over 20 samples. For SPA, the confidence C is guaranteed as long as the minimum number of samples is provided, as discussed in Section 4.3.

7 RELATED WORK

There is a long history of using model checking (but *not* SMC) to verify computer systems, or at least portions of them. Architects have used model checkers—including Murphi [19], PRISM [38], UPPAAL [16], and NuSMV [11]—often to verify cache coherence protocols (e.g., [49, 57], but also other subsystems that are amenable to model checking. Less often, architects have also used probabilistic model checking to overcome the state space explosion problem [43].

SMC is already used in cyber-physical systems (CPS) [53, 60, 63]. Due to the criticality of many CPS, that community has been quicker to embrace SMC and its ability to facilitate more rigorous evaluations. We are unaware, though, of any prior use in computer architecture, and we seek to ease its adoption by wrapping it with our SPA framework.

Researchers have identified many challenges in experimentally evaluating systems like computer processors [59]. While statistical techniques have previously been used for data clustering [61] and risk analysis [14], our work focuses on the performance evaluation of computer systems based on *rigorous statistical inference*. Statistical inference can draw conclusions about a population based on samples, regardless of the population distribution.

Our work is related to but different from other works that use *statistical regression* [7, 33], such as ANOVA [25] and quantile regression [18], to study how the population (for example, its mean and quantiles) changes with external factors. Therefore, these are two complementary statistical techniques that are for different analyses. We note that these regression analysis techniques may not be applicable to general cases due to their underlying assumptions. ANOVA requires the Gaussian assumption, which, as noted in Section 1 is not always valid. Additionally, both ANOVA and quantile regression require an assumption about the relationship between the independent and dependent variables. Prior work assumes it to be linear without sufficient justification [18].

The broad idea of improving the automated management of a simulator and its results has been explored before, although not in the context of SMC. The gem5art framework [6] provides a nice infrastructure for this purpose, and we could imagine incorporating it into SPA in the future.

8 CONCLUSIONS

Variability is an important, real-world phenomenon that must be accounted for when evaluating computer systems. Doing so requires multiple executions with variability injection and a rigorous approach for analyzing results. We have shown that SMC offers great advantages for result analysis, particularly when compared to prior work that requires untenable assumptions. We have developed the SPA framework to facilitate the adoption of SMC among computer architects. The SPA tool integrated with gem5 is available on GitHub, and standalone SPA for result analysis is available on PyPI. Avenues for future work include the exploration of richer properties and the use of hyperproperties.

ACKNOWLEDGMENTS

This work is supported in part by the National Science Foundation under grant CCF-213-3160.

REFERENCES

[1] Gul Agha and Karl Palmskog. 2018. A survey of statistical model checking. *ACM Transactions on Modeling and Computer Simulation* 28, 1 (2018), 6:1–6:39.
[2] A. R. Alameldeen, C. J. Mauer, M. Xu, P. J. Harper, M. M. K. Martin, and D. J. Sorin. 2002. Evaluating Non-deterministic Multi-threaded Commercial Workloads. In *Proc. of Computer Architecture Evaluation Using Commercial Workloads.*
[3] Alaa R. Alameldeen and David A. Wood. 2003. Variability in Architectural Simulations of Multi-Threaded Workloads. In *Proceedings of the 9th International Symposium on High-Performance Computer Architecture.*
[4] David Arney, Miroslav Pajic, Julian M. Goldman, Insup Lee, Rahul Mangharam, and Oleg Sokolsky. 2010. Toward Patient Safety in Closed-Loop Medical Device Systems. In *ACM/IEEE International Conference on Cyber-Physical Systems.*
[5] C. Bienia and K. Li. 2009. PARSEC 2.0: A New Benchmark Suite for Chip-Multiprocessors. In *Proc. of the 5th Annual Workshop on Modeling, Benchmarking and Simulation.*
[6] B. R. Bruce, A. Akram, H. Nguyen, K. Roarty, M. Samani, M. Friborz, T. Reddy, M. D. Sinclair, and J. Lowe-Power. 2021. Enabling Reproducible and Agile Full-System Simulation. In *IEEE Int'l Symp. on Performance Analysis of Systems and Software.*
[7] B. S. Cade and B. R. Noon. 2003. A gentle introduction to quantile regression for ecologists. *Frontiers in Ecology and the Environment* 1, 8 (2003), 412–420.
[8] T. E. Carlson, W. Heirman, and L. Eeckhout. 2011. Sniper: Exploring the Level of Abstraction for Scalable and Accurate Parallel Multi-Core Simulation. In *SC.*
[9] G. Casella and R. L Berger. 2021. *Statistical inference.* Cengage Learning.
[10] Tianshi Chen, Yunji Chen, Qi Guo, Olivier Temam, Yue Wu, and Weiwu Hu. 2012. Statistical performance comparisons of computers. In *IEEE International Symposium on High-Performance Comp Architecture.*
[11] A. Cimatti, E. Clarke, F. Giunchiglia, and M. Roveri. 2000. NUSMV: A New Symbolic Model Checker. *International Journal of Software Tools for Technology Transfer* (2000).
[12] M. R. Clarkson, B. Finkbeiner, M. Koleini, K. K. Micinski, M. N. Rabe, and C. Sanchez. 2014. Temporal Logics for Hyperproperties. In *International Conference on Principles of Security and Trust.*

[13] Charles J Clopper and Egon S Pearson. 1934. The use of confidence or fiducial limits illustrated in the case of the binomial. *Biometrika* 26, 4 (1934), 404–413.

[14] Weilong Cui and Timothy Sherwood. 2017. Estimating and Understanding Architectural Risk. In *International Symposium on Microarchitecture.*

[15] F. Dannenberg, M. Kwiatkowska, C. Thachuk, and A. Turberfield. 2013. DNA Walker Circuits: Computational Potential, Design, and Verification. In *Proc. 19th International Conference on DNA Computing and Molecular Programming.*

[16] A. David, K. G. Larsen, A. Legay, M. Mikučionis, and D. B. Poulsen. 2015. Uppaal SMC Tutorial. *Int'l Journal of Software Tools for Technology Transfer* (2015).

[17] C. Daws, M. Kwiatkowska, and G. Norman. 2004. Automatic Verification of the IEEE 1394 Root Contention Protocol with KRONOS and PRISM. *International Journal of Network Security and Its Applications* 5 (2004).

[18] Augusto Born De Oliveira, Sebastian Fischmeister, Amer Diwan, Matthias Hauswirth, and Peter F Sweeney. 2013. Why you should care about quantile regression. In *Architectural Support for Programming Languages and Operating Systems, ASPLOS '13.* 207–218. http://dl.acm.org/citation.cfm?doid=2451116.2451140

[19] David L Dill. 1996. The Murphi Verification System. In *CAV*, Vol. 1102.

[20] M. Duflot, M. Kwiatkowska, G. Norman, and D. Parker. 2006. A Formal Analysis of Bluetooth Device Discovery. *International Journal of Software Tools for Technology Transfer* (2006).

[21] B. Efron and Tibshirani R.J. 1993. An introduction to the bootstrap. Chapman and Hall, New York, NY. *Farrell, J., Johnston, M. and Twynam, D.(1998),"Volunteer motivation, satisfaction, and management at an elite sporting competition", Journal of Sport Management* 12 (1993), 288–300.

[22] M. Elboukhari, A. Azizi, and M. Azizi. 2010. Analysis of the Security of BB84 by Model Checking. *Int'l Journal of Network Security and Its Applications* 2 (2010).

[23] L. Feng, C. Wiltsche, L. Humphrey, and U. Topcu. 2015. Controller Synthesis for Autonomous Systems Interacting with Human Operators. In *Proceedings of the ACM/IEEE Sixth International Conference on Cyber-Physical Systems.*

[24] M. Fruth. 2011. *Formal Methods for the Analysis of Wireless Network Protocols.* Ph. D. Dissertation. University of Oxford.

[25] Andy Georges, Dries Buytaert, and Lieven Eeckhout. 2007. Statistically rigorous java performance evaluation. *ACM SIGPLAN Notices* 42, 10 (2007), 57–76.

[26] Jean Dickinson Gibbons and Subhabrata Chakraborti. 2011. *Nonparametric Statistical Inference.* Springer Berlin Heidelberg, Berlin, Heidelberg, 157–166. https://doi.org/10.1007/978-3-642-04898-2_420

[27] Louis Guttman. 1977. What is not what in statistics. *Journal of the Royal Statistical Society. Series D (The Statistician)* 26, 2 (1977), 81–107.

[28] H. Hansson and B. Jonsson. 1994. A Logic for Reasoning About Time and Reliability. *Formal Aspects of Computing* 6, 5 (1994).

[29] A. S. Harji, P. A. Buhr, and T. Brecht. 2011. Our troubles with Linux and why you should care. In *Proc. of the Second Asia-Pacific Workshop on Systems.* 1–5.

[30] Samuel Irving, Bin Li, Shaoming Chen, Lu Peng, Weihua Zhang, and Lide Duan. 2020. Computer Comparisons in the Presence of Performance Variation. *Frontiers of Computer Science* 14, 1 (2020).

[31] Tomas Kalibera, Lubomir Bulej, and Petr Tuma. 2005. Benchmark precision and random initial state. In *Proc. of the 2005 Int'l Symposium on Performance Evaluation of Computer and Telecommunication Systems.*

[32] Tomas Kalibera and Richard Jones. 2020. Quantifying Performance Changes with Effect Size Confidence Intervals. arXiv:2007.10899 [stat.ME]

[33] Roger Koenker and Kevin F Hallock. 2001. Quantile regression. *Journal of economic perspectives* 15, 4 (2001), 143–156.

[34] M. Kwiatkowska and G. Norman. 2002. Verifying Randomized Byzantine Agreement. In *Formal Techniques for Networked and Distributed Systems.*

[35] M. Kwiatkowska, G. Norman, and D. Parker. 2005. Probabilistic Model Checking and Power-Aware Computing. In *7th International Workshop on Performability Modeling of Computer and Communication Systems.*

[36] M. Kwiatkowska, G. Norman, and D. Parker. 2006. Controller Dependability Analysis by Probabilistic Model Checking. *Control Engineering Practice* 15 (2006).

[37] M. Kwiatkowska, G. Norman, and D. Parker. 2008. Using Probabilistic Model Checking in Systems Biology. *ACM SIGMETRICS Performance Evaluation Review* 35 (2008).

[38] M. Kwiatkowska, G. Norman, and D. Parker. 2011. *Computer Aided Verification.* Springer Berlin Heidelberg, Chapter PRISM 4.0: Verification of Probabilistic Real-time Systems.

[39] M. Kwiatkowska, G. Norman, and D. Parker. 2012. Probabilistic Verification of Herman's Self-Stabilisation Algorithm. *Formal Aspects of Computing* 24 (2012).

[40] Marta Kwiatkowska, Gethin Norman, and David Parker. 2018. Probabilistic Model Checking: Advances and Applications. In *Formal System Verification: State-of the-Art and Future Trends.* 73–121.

[41] A. Legay, B. Delahaye, and S. Bensalem. 2010. Statistical model checking: An overview. In *Runtime Verification*, Howard Barringer, Ylies Falcone, Bernd Finkbeiner, Klaus Havelund, Insup Lee, Gordon Pace, Grigore Roşu, Oleg Sokolsky, and Nikolai Tillmann (Eds.). Vol. 6418. Springer Berlin Heidelberg.

[42] J. Lowe-Power, A. M. Ahmad, A. Akram, M. Alian, R. Amslinger, M. Andreozzi, A. Armejach, N. Asmussen, B. Beckmann, S. Bharadwaj, G. Black, G. Bloom, B. R. Bruce, D. Rodrigues Carvalho, J. Castrillon, L. Chen, N. Derumigny, S. Diestelhorst, W. Elsasser, C. Escuin, M. Fariborz, A. Farmahini-Farahani, P. Fotouhi, R. Gambord, J. Gandhi, D. Gope, T. Grass, A. Gutierrez, B. Hanindhito, A. Hansson, S. Haria, A. Harris, T. Hayes, A. Herrera, M. Horsnell, S. Ali Raza Jafri, R. Jagtap, H. Jang, R. Jeyapaul, T. M. Jones, M. Jung, S. Kannoth, H. Khaleghzadeh, Y. Kodama, T. Krishna, T. Marinelli, C. Menard, A. Mondelli, M. Moreto, T. Mück, O. Naji, K. Nathella, H. Nguyen, N. Nikoleris, L. E. Olson, M. Orr, B. Pham, P. Prieto, T. Reddy, A. Roelke, M. Samani, A. Sandberg, J. Setoain, B. Shingarov, M. D. Sinclair, T. Ta, R. Thakur, G. Travaglini, M. Upton, N. Vaish, I. Vougioukas, W. Wang, Z. Wang, N. Wehn, C. Weis, D. A. Wood, H. Yoon, and É. F. Zulian. 2020. The gem5 Simulator: Version 20.0+. arXiv:2007.03152 [cs.AR]

[43] A. Lungu, P. Bose, D. Sorin, S. German, and G. Janssen. 2009. Multicore Power Management: Ensuring Robustness via Early-Stage Formal Verification. In *Seventh ACM-IEEE Int'l Conference on Formal Methods and Models for Codesign.*

[44] Yue Luo and Lizy K. John. 2008. *Using Statistical Theory to Study Issues in Microprocessor Simulation.* Technical Report TR-040225-01. University of Texas, Department of ECE.

[45] Oded Maler and Dejan Nickovic. 2004. Monitoring temporal properties of continuous signals. In *Formal Techniques, Modelling and Analysis of Timed and Fault-Tolerant Systems.* Springer, 152–166.

[46] G. Norman, D. Parker, M. Kwiatkowska, and S. Shukla. 2005. Evaluating the Reliability of NAND Multiplexing with PRISM. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 24 (2005).

[47] G. Norman, D. Parker, M. Kwiatkowska, S. Shukla, and R. Gupta. 2003. Using Probabilistic Model Checking for Dynamic Power Management. In *3rd Workshop on Automated Verification of Critical Systems.*

[48] G. Norman and V. Shmatikov. 2006. Analysis of Probabilistic Contract Signing. *Journal of Computer Security* 14 (2006).

[49] N. Oswald, V. Nagarajan, D. J. Sorin, V. Gavrielatos, T. Olausson, and R. Carr. 2022. HeteroGen: Automatic Synthesis of Heterogeneous Cache Coherence Protocols. In *IEEE Int'l Symposium on High-Performance Computer Architecture.*

[50] A. Pnueli. 1977. The Temporal Logic of Programs. In *18th Annual Symposium on Foundations of Computer Science.*

[51] Kishore Kumar Pusukuri, Rajiv Gupta, and Laxmi N. Bhuyan. 2012. Thread Tranquilizer: Dynamically Reducing Performance Variation. *ACM Transactions on Architecture and Code Optimization* 8, 4 (2012).

[52] A. Raghavan, Y. Luo, A. Chandawalla, M. Papaefthymiou, K. P. Pipe, T. F. Wenisch, and M. M. K. Martin. 2012. Computational Sprinting. In *Proc. of the 18th Symp. on High Performance Computer Architecture.*

[53] N. Roohi, Y. Wang, M. West, G. E. Dullerud, and M. Viswanathan. 2017. Statistical Verification of the Toyota Powertrain Control Verification Benchmark. In *ACM Int'l Conference on Hybrid Systems: Computation and Control.*

[54] Somayeh Sardashti, Andre Seznec, and David A. Wood. 2016. Yet Another Compressed Cache: A Low-Cost Yet Effective Compressed Cache. 13, 3 (2016).

[55] M. Shahrad, J. Balkind, and D. Wentzlaff. 2019. Architectural Implications of Function-as-a-Service Computing. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Computer Architecture.*

[56] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. 2002. Automatically Characterizing Large Scale Program Behavior. In *Proc. of the Tenth Int'l Conference on Architectural Support for Programming Languages and Operating Systems.*

[57] S. Srinivasan, P.S. Chhabra, P.K. Jaini, A. Aziz, and L. John. 1999. Formal verification of a snoop-based cache coherence protocol using symbolic model checking. In *Proceedings of the Twelfth International Conference on VLSI Design.*

[58] T. Tajimi, M. Hayashi, Y. Futamase, R. Shioya, M. Goshima, and T. Tsumura. 2018. Isolation-Safe Speculative Access Control for Hardware Transactional Memory. In *25th IEEE Int'l Conference on Electronics, Circuits and Systems.*

[59] J. Vitek and T. Kalibera. 2011. Repeatability, Reproducibility, and Rigor in Systems Research. In *Proc. of the Ninth ACM Int'l Conf. on Embedded Software.*

[60] Y. Wang, M. Zarei, B. Bonakdarpour, and M. Pajic. 2019. Statistical Verification of Hyperproperties for Cyber-Physical Systems. *ACM Transactions on Embedded Computing Systems* 18 (2019).

[61] R. E. Wunderlich, T. F. Wenisch, B. Falsafi, and J. C. Hoe. 2003. SMARTS: Accelerating Microarchitecture Simulation via Rigorous Statistical Sampling. In *Proc. of the 30th Annual Int'l Symp. on Computer Architecture.*

[62] H. Younes, G. Norman M. Kwiatkowska, and D. Parker. 2006. Numerical vs. Statistical Probabilistic Model Checking. *International Journal on Software Tools for Technology Transfer* 8 (2006).

[63] M. Zarei, Y. Wang, and M. Pajic. 2020. Statistical Verification of Learning-Based Cyber-Physical Systems. In *ACM International Conference on Hybrid Systems: Computation and Control.*