Fast Synthesis of Power and Temperature Profiles for the Development of Data-Driven Resource Managers

Ivan Ukhov, Diana Marculescu, Petru Eles, and Zebo Peng

***Abstract*—The goal of this work is to facilitate the development of proactive power- and temperature-aware resource managers that leverage machine learning in order to attain their objectives. In this context, the availability of sufficiently large amounts of relevant data, which are essential for learning and, therefore, exploration of research ideas, is elusive. In order to fulfill the need, we present a toolchain for fast generation of realistic power and temperature profiles of computer systems. The toolchain provides profuse representative data to learn from during development stages. The overreaching objective is to help research by making it tractable to experiment with the highly promising but data-demanding state-of-the-art techniques for prediction.**

***Index Terms*—Computer system, machine learning, network traffic, power, simulation, temperature, workload.**

I. Introduction and Motivation

Power consumption and heat dissipation are of great importance. Power translates to energy, and energy to hours of battery life and to electricity bills. Temperature, on the other hand, is one of the major causes of permanent damage [1], which necessitates adequate cooling equipment and, hence, escalates product expenses [2]. The situation is deteriorated further by the power-temperature interplay: higher power leads to higher temperature, and higher temperature to higher power [3]. Therefore, accounting for power and temperature is key to achieving effectiveness, efficiency, and robustness.

Our work is to assist researchers who are concerned with power and temperature in one specific but rather broad context: the development of data-driven power- and temperature-aware resource managers for computer systems where *data-driven* refers to the usage of algorithms that learn from data [4]. To this end, we develop a toolchain for fast synthesis of realistic power and temperature profiles, which is motivated as follows.

The power and temperature characteristics of a computer system are difficult to describe mathematically or algorithmically. Modern computer systems are complex, and many aspects are inherently uncertain to the designer due to such phenomena as process variation [5], aging [6], and varying workload. The concern is particularly difficult to address from a theoretical standpoint; it can be better dealt with taking a more pragmatic approach, namely, making use of runtime data. Assuming that the system provides mechanisms for profiling and monitoring, direct or indirect power and temperature measurements are an invaluable source of information for making proactive power- and temperature-aware resource-management decisions [2], [7].

Acting proactively requires forecasting the future by learning from the past, which is typically accomplished by virtue of machine learning [4]. Regardless of the learning technique utilized, the technique requires data to learn from. Real data are rarely available in large enough quantities. The target platform might not be at one's disposal or might not be an appropriate place for early experimentation, which is common in research. Therefore, a research environment is typically composed of a number of computer simulators as they constitute a feasible alternative source of the needed data. An illustrative example is given in Fig. 1, which depicts a potential environment for developing a resource manager for a computer system.

Computer simulators, however, fall short when it comes to complex systems: it might take days for a state-of-the-art simulator to simulate a short, in wall-clock time, program. This scheme is not affordable for designing data-driven solutions as they require many simulations with potentially large payloads.

To summarize, real data are rarely and sparingly available, and simulation data are prohibitively time consuming to obtain, which hampers the development of data-driven solutions. The need for alternative sources of high-quality data is prominent, which brings us to the goal of this work: our objective is to provide such a source for the case of power and temperature.

It is well understood that the solution being developed will eventually face real data, and, therefore, it should be tested and, if it is needed, calibrated in a stage environment prior to deploying the solution to a production environment. The synthetic data that we generate can substantially speed up this process due to the flexibility and fast feedback that they provided. In particular, one can filter out inadequate ideas at early stages and focus solely on those that are viable.

The remainder of the paper is organized as follows. In Sec. II, we elaborate on the related work and summarize our contribution. The addressed problem is formalized in Sec. III. The methodology that we follow and our toolchain are presented in Sec. IV and Sec. V, respectively. The experimental results are given in Sec. VI. Section VII concludes the paper.

II. Related Work and Our Contribution

The present work is related to computer simulation. For our purposes, it is sufficient to distinguish four types of simulation:

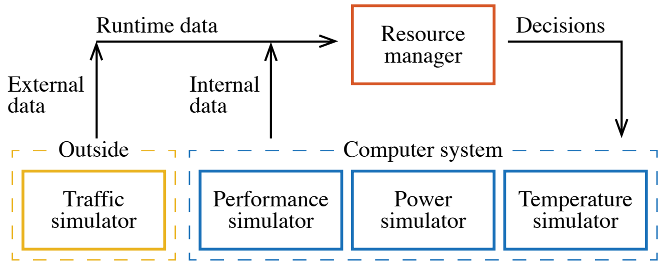

Figure 1. Research environment for developing a resource manager for a computer system. The right dashed box should be understood at a single module, and the corresponding arrows should be treated accordingly.

traffic, performance, power, and temperature, which are shown in Fig. 1. In this paper, *traffic* refers to a stream of jobs that lade the system with work, and *performance* to various performance metrics of the system such as the numbers of executed instructions and read/write memory accesses.

Let us discuss traffic first. The Poisson process [8] is arguably the most well-known model in this regard. However, the seminal work in [9] and subsequent studies have shown that network traffic exhibits fractal properties such as burstiness, self-similarity, and long-range dependence. The Poisson process is unable to express any of these properties. Another popular model is the fractional Gaussian noise [8], which is a self-similar stochastic process. However, this process is suited for modeling arrivals per unit of time but not for modeling arrival or inter-arrival times, which are what is typically needed for simulation. Moreover, the noise can take negative values and is a monofractal process, both of which are unrealistic. The work in [10] addresses the aforementioned concerns by introducing a multifractal wavelet model for characterizing positive-valued sequences with long-range-dependent correlations.

A performance simulator that we would like to highlight is Sniper [11]. Sniper is aimed at x86-based systems, and it has been validated against Intel Core 2 and Nehalem architectures. The simulator works at a higher level of abstraction than the one of cycle-accurate simulators, which makes its simulation times more affordable. Regarding power simulation, a common choice is the McPAT framework [12]. McPAT is also capable of estimating the areas occupied by processing elements, which is useful since this information is essential for temperature simulation. Architecture-level temperature simulation is arguably dominated by HotSpot [13]. A popular alternative is 3D-ICE [14], which is particularly focused on 3D structures.

The pipeline assembled from the aforementioned simulators is the one frequently used in today's research (see the blue boxes in Fig. 1). However, as we elaborate in Sec. I, it is extremely time consuming and, hence, is not suitable for learning purposes. In our experience, this is mainly due to performance simulation and, to a lesser extent, power simulation. Temperature simulation usually has a relatively small cost. These concerns are to be discussed and illustrated in Sec. VI.

Our work makes the following major contribution. We present an efficient toolchain for generating realistic power and temperature profiles of computer systems in order to facilitate the development of intelligent data-driven techniques for the analysis and management of such systems. The toolchain is open source and publicly available online at [15].

III. Problem Formulation

Consider an in-design or in-production computer system. The system is composed of a number of processing elements and is capable of performing a number of operations. The system receives a stream of user requests or jobs, which it is supposed to process. Each job implies a certain amount of work to be done and, thus, a certain amount of power to be consumed and a certain amount of heat to be dissipated.

Consider now a research project targeted at developing a resource manager for the system under consideration. Since the importance of power and temperature is well understood, the solution is required to take these two quantities into account. Suppose further that the resource manager is to be based on a technique that learns from the data available at runtime.

Our goal is to develop a toolchain for generating power and temperature profiles of the system in order to provide the project with plenty of data to experiment with. The generated profiles should preserve the particularities of job arrivals and program executions that are present in real life, which is what makes the subsequent learning meaningful. In addition, the generation process should be substantially faster than traditional simulations, which is what makes the work worth doing.

IV. Methodology

In this section, we describe the general methodology that our toolchain follows. An overview of the methodology is given in Fig. 2 and can be summarized as follows. There are two major stages: data acquisition and data synthesis. The leftmost boxes in Fig. 2 correspond to the data-acquisition stage, and the rightmost one to the data-synthesis stage. The data-acquisition stage collects and stores reference data while the data-synthesis stage fetches these data and produces the desired power and temperature traces of the system. There are two types of reference data: traffic and workload, which are referred to as patterns. The two pieces characterize job arrivals and job workloads, and they are to be discussed in Sec. IV-A and Sec. IV-B and to be combined in Sec. IV-C.

A. *Traffic*

The foremost step is to construct a traffic model describing *when* jobs arrive. The model should satisfy a number of requirements in order to be practically useful. First, it should be straightforward to configure given a data set of traffic data (a sequence of arrival times of jobs) serving as a reference. Second, it should be able to capture the idiosyncrasies present in real traffic as it is the foundation for the subsequent generation of power and temperature profiles (discussed in Sec. IV-C).

The required ability to absorb reference data acknowledges the utility of the data that are readily at one's disposal due to the ubiquitously deployed monitoring and logging mechanisms. These mechanisms are outside the scope of this work. In Fig. 2, they are represented by a set of boxes labeled "Logger." The collected traffic patterns are assumed to be stored in a repository; see the top cloud in Fig. 2. For a good example of such data, the interested reader is referred to the cluster-usage traces published by Google in 2011 [16]; the data set will be discussed further in Sec. VI. Let us now move on to the analysis and synthesis of a particular traffic pattern.

In order to be able to generate realistic traffic, we employ the multifractal wavelet model proposed in [10], which is motivated in Sec. II. In accordance with the model, we take a reference time series of arrival times, analyze it by means of the discrete wavelet transform based on Haar wavelets, and construct a certain representation of the data, which can then be used for generating random time series matching the fractal properties of the original one. As a result, the model becomes tailored to the traffic pattern of the particular problem at hand.

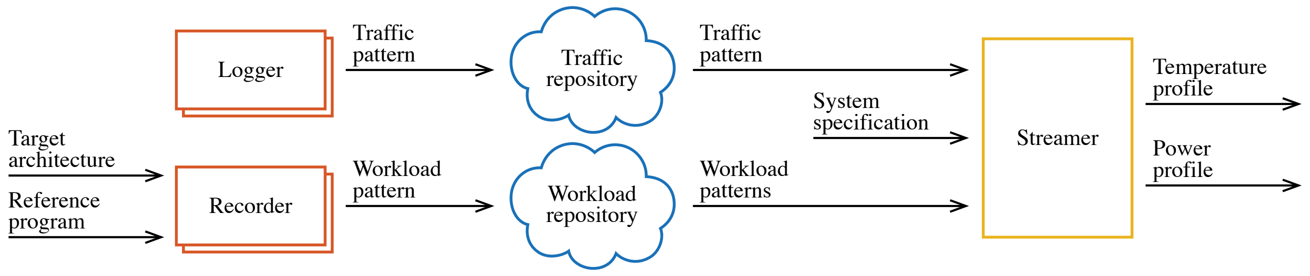

Figure 2. Workflow of our methodology. The data-acquisition stage is to the left of the clouds, and the data-synthesis is to the right.

To summarize, we have obtained a data set of reference arrival times and a technique for analyzing them and generating similar arrival streams. The technique is capable of capturing the properties that are commonly present in real traffic.

B. *Workload*

An arrival is only a time stamp without any information about the actual workload. In this subsection, we describe how workload candidates are obtained and complement job arrivals, which is depicted at the bottom of Fig. 2. To begin with, workload candidates should conform to a number of criteria. First, since we aim to produce realistic trace, workloads should represent well the applications or services that the system is supposed to provide to the user. Second, a workload should be fast to evaluate, which, in our context, refers to obtaining the power consumption over time of that workload.

Our workload modeling is based on full-system simulations of reference programs. However, if we had incorporated such simulations into our workflow *directly*, we would have ended up with a configuration similar to the one displayed in Fig. 1. This would have defeated the purpose of our work since, as motivated in Sec. I, detailed simulations are considerably time consuming. Instead, we utilize high-level recordings; this functionality corresponds to the boxes labeled "Recorder" in Fig. 2. To elaborate, using a simulator capable of modeling the target architecture, we execute each reference program in isolation and record certain information about this execution. At a later stage of our pipeline (see Streamer in Fig. 2), the collected information is utilized in order to substantiate jobs upon their arrival, and this stage requires no simulation.

Performance and power simulations are by far the largest expense in terms of time. Therefore, the information about a reference program's execution that we record is the power consumption of that execution. This approach pushes the aforementioned expense to the data-acquisition stage and eliminates it from the data-synthesis stage. Since the actual data generation is deliver from the expensive simulations, it has a very low computational demand. This demand is negligible compared to the one of the scenario depicted in Fig. 1, in which one undertakes performance and power simulations nonstop.

The power consumption of a program can be recorded in different ways. The first aspect to note is that we record power as a function of time (assuming a certain sampling interval). Second, the dynamic and static components of the power consumption are recorded separately for a better control over the subsequent composition (Sec. IV-C). Third, the power consumption is recorded for all the processing elements of interest (for instance, cores and shared caches).

The result of the recording is a repository of power traces corresponding to real programs, which we refer to as workload patterns; see the bottom cloud in Fig. 2. Full-system simulations take time; however, they have to be done only once.

Let us turn to the assignment of a workload pattern to an arrival. We shall refer to this functionality as the workload model. The input to this model is a stream of arrival times, and its output is a stream of fully characterized jobs (time and work). The workload model relies on domain-specific knowledge. For instance, the output stream can be made dependent on the input stream in order to introduce autocorrelations and, thereby, model periodic and coupled workloads. Therefore, we let the model be user defined. The default option that is used in our toolchain is choosing workload patterns at random.

To recapitulate, we have obtained a database of reference workloads and discussed the formation of job streams from arrival streams. The utilized patterns correspond to executions of real programs and, therefore, exhibit realistic traits.

C. *Composition*

The stream of jobs needs to be now processed by progressively building a schedule, constructing a power profile, and computing the corresponding temperature profile. In Fig. 2, this functionality resides in the box labeled "Streamer."

Since the major use case for our toolchain is the development of data-driven resource managers, the resource manager of the system at hand is assumed to be given, which also includes the scheduling policy. For each job, the policy specifies what computational resources are granted to the job and *when*.

Once the job has been scheduled, we proceed to updating the power profile of the system. The power profile is a matrix that specifies the power consumption of the processing elements over discrete time moments; the rows and columns traverse the processing elements and time moments, respectively. The workload pattern of the job can also be seen as such a matrix but smaller. Then, in accordance with the scheduling decision, each row of the workload pattern is added to an appropriate row of the power profile starting from an appropriate column.

There are several aspects that are worth noting. First, as mentioned in Sec. IV-B, we record dynamic and static power separately. We then make sure that the static component is present even when no job is being "executed." Second, the

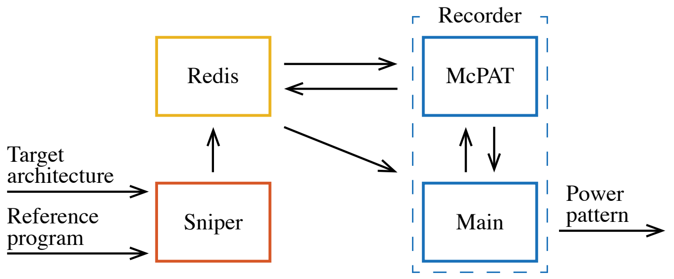

Figure 3. Recording infrastructure. The Recorder tool corresponds to the two blue boxes on the right-hand side of the figure.

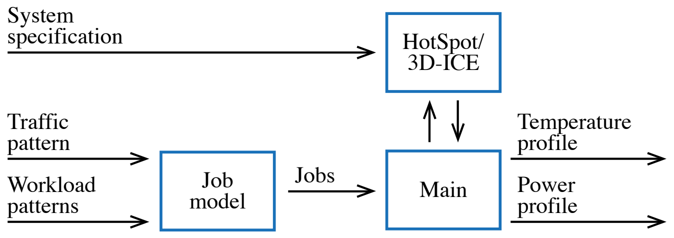

Figure 4. Streamer tool. *System specification* refers primarily to the information about the thermal package and floorplans of the platform, which are needed for temperature simulation. *Job model* refers jointly to the traffic and workload models (Sec. IV-A and Sec. IV-B).

types of the processing elements that the workload pattern was recorded on should be respected when scheduling and distributing the pattern. Third, certain processing elements might be shared across several jobs, which we discuss now.

The jobs that are mapped by the scheduling policy to disjoint sets of processing elements require no special treatment. The interleaving of several jobs on a core can be modeled by an appropriate interleaving of the corresponding workload patterns: workload patterns have the notion of time, and they can be cut into pieces and stitched back as needed. The joint effect of several jobs on a cache can be reproduced by an additive model: the power values pertaining to the cache, as captured by the workload patterns of the jobs in question, are summed up. This approximation is sufficient as the power consumption of caches is usually relatively low. Other effects originating from resource sharing are not currently addressed in our methodology.

Finally, the temperature profile of the system is obtained by gradually feeding the power profile into a temperature simulator. These two profiles are our final output. They are fed into the resource manager as if they were the actual sensor readings.

To recapitulate, the methodology that we follow is composed of two stages. In the data-acquisition stage, reference traffic and workload data are harvested. In the data-synthesis stage, the data are used to generate a stream of jobs and subsequently compute a power profile and a temperature profile. The produced profiles preserve the particularities of the reference data. At the same time, the synthesis is fast since it has no expensive performance or power simulations involved; the time consumed by the procedure delineated above is practically negligible.

V. Toolchain

The toolchain follows the methodology described in Sec. IV. It consists of a number of command-line tools, and the tools are composed of a number of stand-alone packages. We also make use of third-party software, including the simulators introduced in Sec. II. The toolchain is publicly available online at [15].

The main programs of the toolchain are called *Recorder* and *Streamer*, which we discuss in the following subsections.

A. Recorder

The Recorder tool is used at the data-acquisition stage of our methodology in order to record reference workload patterns, which are discussed in Sec. IV-B and needed for Streamer. In Fig. 2, Recorder corresponds to the bottom-left boxes labeled "Recorder." The tool has a curtain infrastructure around it. This infrastructure is depicted in Fig. 3, in which Recorder itself is represented by the two blue boxes on the right-hand side.

Given a reference application and a target platform, the first step is performance simulation, which we undertake by virtue of the Sniper simulator [11]. The output of the simulator is a series of files containing performance-related information over time. These files are further processed by our Recorder.

The communications between Sniper and Recorder are handled by Redis [17] (see Fig. 3), which is an in-memory key-value storage commonly used for caching and message passing. Whenever Sniper produces a new file with performance information, it sends a message with the file's location to a Redis queue. Recorder fetches this message from the queue (observe the diagonal arrow in Fig. 3) and processes the corresponding file. There are several aspects to note here. First, the indirection allows Sniper and Recorder to work asynchronously, each at its own pace. Second, there can be multiple Recorder processes serving the same message queue. Third, there can be many Sniper processes, each of which simulates a different program and uses a different Redis queue. The above aspects substantially seed up the recording.

The next step is power simulation, which is delegated to McPAT [12]. McPAT is originally a command-line program; we have made a shared library out of it and embedded it into Recorder. In addition, we have introduced a caching mechanism into McPAT, which is inspired by Sniper. The caching mechanism eliminates the repetition of certain computations inside McPAT, which leads to considerable time savings. For caching purposes, we again use Redis; see the direct arrows between Redis and McPAT in Fig. 3. All Recorder processes that are connected to the same Redis server can leverage the same cache, making the caching mechanism shine.

Finally, the computed power pattern is stored in a file, which can then be uploaded to a repository as described in Sec. IV-B and illustrated in Fig. 2. Each such output file is an SQLite database [18], which implies that the versatile SQL language is automatically at one's disposal for working with the data.

B. Streamer

The Streamer tool corresponds to the data-synthesis stage. It is responsible for synthesizing power and temperature data using reference data as a material for the synthesis. In Fig. 2,

Table I
RECORDING (THE PARSEC BENCHMARK SUITE)

	Recording time (hours)			Simulated time (seconds)		
Program	Small	Medium	Large	Small	Medium	Large
blackscholes	0.18	0.74	3.00	0.07	0.28	1.13
bodytrack	0.51	2.00	7.36	0.17	0.70	2.71
canneal	0.61	1.74	4.04	0.18	0.67	1.72
dedup	1.35	3.48	17.20	1.97	2.85	10.75
facesim	12.68	13.18	15.48	7.84	7.93	7.87
ferret	0.83	2.65	12.15	0.40	1.06	4.59
fluidanimate	0.78	2.18	6.90	0.53	1.32	4.11
freqmine	1.24	4.87	18.04	0.67	2.63	9.21
raytrace	4.22	5.69	10.08	0.24	0.63	1.51
streamcluster	0.74	2.88	15.71	0.34	1.40	10.68
swaptions	0.59	2.33	9.05	0.23	0.91	3.64
vips	1.57	4.89	14.47	0.53	1.59	4.39
x264	0.49	2.59	8.61	0.17	0.99	3.09

Small, *medium*, and *large* signify the input size of the programs.

it corresponds to the rightmost box labeled "Streamer." The structure of the tool is illustrated in Fig. 4.

Given a traffic pattern and a set of workload patterns, Streamer proceeds as follows. The traffic pattern is processed as it is described in Sec. IV-A, which results in an adequately configured multifractal wavelet model [10]. The model is then used to generate a stream of arrival times. The arrival times are substantiated using the workload patterns, which is explained in Sec. IV-B. The result is a stream of jobs. In Fig. 4, the whole above operation takes place in the box labeled "Job model." The rest follows Sec. IV-C. Namely, the job stream is first handled by the user's resource manager, which resides in the box labeled "Main." As jobs are being scheduled, the power profile of the system is being progressively constructed. The power profile is piped into a temperature simulator (to be discussed), which delivers a temperature profile. Finally, the synthesized power and temperature profiles are made available to the manager. They can also be saved on disk, in which case, similar to reference data, the output is an SQLite database.

Let us now elaborate on the temperature simulation that Streamer undertakes. It is based on the well-known thermal RC model. We construct a thermal RC circuit for the platform at hand and then use it for analyzing the thermal behavior of the system. The analysis boils down to solving a system of differential equations. In our case, the solution is based on a solver leveraging exponential integrators [19]. The construction of thermal circuits is delegated to either HotSpot [13] or 3D-ICE [14] (see Fig. 4) depending on the user's preferences.

To summarize, Recorder captures workload patterns, and Streamer produces streams of power and temperature data. Both closely follow the methodology described in Sec. IV.

VI. EXPERIMENTAL RESULTS

This section illustrates the performance of the toolchain presented in Sec. V. The experiments that are reported below were conducted on a GNU/Linux machine equipped with 16 CPUs Intel Xeon E5520 2.27 GHz and 24 GB of RAM.

In order to obtain real-life traffic patterns for our experiments, we used a data set published by Google in 2011 [16]. The data set contains the usage data of a large computer cluster over a month period. We downloaded the table tracking the life cycle of the jobs submitted to the cluster and extracted the time stamp of the first event related to each job. As a result, there were around 670 000 arrival times available, which we used for fitting the traffic model as it is described in Sec. IV-A.

A set of workload patters was obtained by simulating and recording (via our infrastructure shown in Fig. 3) the programs from the popular PARSEC [20] and SPEC CPU2006 [21] benchmark suites; the former contains 13 programs, and the latter 29 programs. The architecture used in these simulations is Intel's Nehalem-based Gainestown series; Sniper is shipped with a configuration for this architecture (`nehalem.cfg` and `gainestown.cfg`), and we used it without any changes.

All the reference data that we collected and processed to make them suitable for our toolchain are available at [15].

A. Recording

In the above, we outlined how the reference workload data were harvested using Recorder and the infrastructure around it (recall Sec. V-A and Fig. 3). Let us now elaborate on the performance characteristics of that recording process.

The benchmark suite that we shall look at is PARSEC. Our findings are summarized in Table I. PARSEC provides several choices of inputs to the programs, and each program was recorded with three different inputs, namely, with the ones classified as small, medium, and large. There are two types of information shown in Table I: recording time (in hours), which is the time that was taken to simulate and record the programs, and simulated time (in seconds), which is the time that the programs would have taken in real life. The sampling interval used in all the experiments was one millisecond.

Each input class was recorded in a single batch: all 13 programs were simulated at the same time using 13 Sniper processes, which is explained and motivated in Sec. V-B. Consequently, the total recording time with respect to each batch is dictated by the program that took the most time to finish. For small and medium inputs, this program was `facesim`, which took approximately 13 hours in both cases. The simulated times of `facesim` indicate that PARSEC actually has only one input size for this particular program. Regarding large inputs, `freqmine` finished last; more concretely, the program took 18 hours. As an aside for the interested reader, the simulated and recording times of SPEC CPU2006 (not shown) were an order of magnitude larger than the ones of PARSEC.

It can be seen in Table I that the throughput in terms of simulated time is (expectedly) low: roughly speaking, two–three hours of recording time amounts to one second of simulated time. However, it is important to realize that these are one-time expenses in our methodology; the situation would be much worse if one had to perform such simulations all the time (see Sec. IV-B). Another important aspect to note is that the observed recording times have been substantially reduced by the choice of performance simulator—Sniper is based on novel simulation ideas [11]—and the parallelization strategy and caching mechanism described in Sec. V-B.

Table II
STREAMING (SYNTHESIS OF POWER AND TEMPERATURE PROFILES)

Synthesized time (seconds)	Synthesis time (seconds) $4+1$	$8+2$	$16+4$	$32+8$
10	0.24	0.40	0.67	1.13
100	2.11	3.66	6.18	10.19
1000	20.59	37.47	64.58	104.00
10000	214.98	394.84	598.89	984.59

"$M+N$" stands for M cores and N L3 caches. Every four cores have a single shared L3 cache; therefore, it holds that $N = M/4$.

B. Streaming

Let us turn to Streamer, which corresponds to the data-synthesis stage (recall Sec. V-B and Fig. 4). Our objective in this subsection is to study the scalability of the tool as measured by synthesis time, which is the time that is needed for the tool to synthesize power and temperature profiles under certain conditions or requirements. In these experiments, workload patterns are assigned to job arrivals randomly, and a simple first-in-first-served scheduling policy is assumed; both are the default but easily replaceable options used by the toolchain.

The preformed experiments are consolidated in Table II. We report our synthesis time along two axes: synthesized time (rows) and platform size (columns). The former is analogous to simulated time, and the latter represents different platforms as follows. Each considered platform is composed of a number of cores, and there is a single L3 cache for every four cores; both cores and caches are referred to as processing elements. Platform size is defined as the number of processing elements, and it is denoted by "$M + N$" in Table II where M and N are the numbers of cores and L3 caches, respectively.

Unlike the throughput of simulation (discussed in the previous subsection), the throughput of synthesis in terms of synthesized time is very high, which is well supported by Table II. To give an example, it takes Streamer around a minute to produce power and temperature data that are worth around 17 minutes of runtime of a computer system with 16 cores, which would be practically infeasible to achieve with full-fledged simulations (refer to the results given in Table I).

Another observation made from Table II is that synthesis time scales linearly with respect to the length of the time span being synthesized (synthesized time), and the same can be concluded regarding the other dimension, platform size. The growth with respect to platform size is due to the increasing complexity of the underlying thermal RC circuit used for temperature simulation; thermal circuits are elaborated on in Sec. V-B.

To summarize, the results reported in Table I motivate our work and communicate well the message of this paper: the speed of the state-of-the-art simulators is severely onerous for the purpose of experimenting with machine-learning techniques. The core problem is that such techniques typically require lots of data (long execution traces); moreover, these data might need to be recalculated each time a parameter changes such as the parameterization of the scheduling policy used. The results in Table II show that the proposed approach can efficiently tackle this problem by taking the data burden away and, hence, making it easier to experiment with data-driven techniques.

VII. CONCLUSION

In this paper, we have emphasized the need for developing tools for the design of computer systems with data-driven components. We have argued that the techniques which capitalize on learning from runtime data have special requirements, and that the state-of-the-art simulation tools are unable to fulfill them due to their prohibitively large simulation times.

Acknowledging the importance of power and temperature for resource management of computer systems, we have developed a methodology for fast generation of power and temperature profiles of such systems, which preserve the idiosyncrasies of their real-life counterparts. Following the methodology, we have implemented and open sourced a toolchain, which has been assessed and shown to have a high computational throughput.

REFERENCES

[1] (2017, September) Failure mechanisms and models for semiconductor devices. JEDEC. [Online]. Available: https://www.jedec.org

[2] M. Chaudhry *et al.*, "Thermal-aware scheduling in green data centers," *ACM Computing Surveys*, vol. 47, pp. 39:1–39:48, February 2015.

[3] Y. Liu *et al.*, "Accurate temperature-dependent integrated circuit leakage power estimation is easy," in *DATE*, 2007, pp. 1526–1531.

[4] C. Bishop, *Pattern recognition and machine learning*, ser. Information Science and Statistics. Springer, 2006.

[5] A. Chandrakasan, W. Bowhill, and F. Fox, *Design of high-performance microprocessor circuits*. Wiley-IEEE Press, 2000.

[6] A. Coskun *et al.*, "Analysis and optimization of MPSoC reliability," *J. Low Power Electronics*, vol. 2, pp. 56–69, 2006.

[7] A. Coskun, T. Rosing, and K. Gross, "Proactive temperature management in MPSoCs," in *International Symposium on Low Power Electronics & Design*, 2008, pp. 165–170.

[8] M. Lifshits, *Random processes by example*. World Scientific, 2014.

[9] W. Leland *et al.*, "On the self-similar nature of Ethernet traffic," *IEEE/ACM Trans. Networking*, vol. 2, pp. 1–15, February 1994.

[10] R. Riedi *et al.*, "A multifractal wavelet model with application to network traffic," *IEEE Trans. Inf. Theory*, vol. 45, pp. 992–1018, April 1999.

[11] T. Carlson, W. Heirman, and L. Eeckhout, "Sniper: Exploring the level of abstraction for scalable and accurate parallel multi-core simulations," in *Int. Conf. High Performance Computing, Networking, Storage and Analysis*, November 2011, pp. 52:1–52:12.

[12] S. Li *et al.*, "McPAT: An integrated power, area, and timing modeling framework for multicore and manycore architectures," in *IEEE/ACM Int. Symp. Microarchitecture*, December 2009, pp. 469–480.

[13] K. Skadron *et al.*, "Temperature-aware microarchitecture: Modeling and implementation," *ACM Trans. Architecture and Code Optimization*, vol. 1, pp. 94–125, March 2004.

[14] A. Sridhar *et al.*, "3D-ICE: Fast compact transient thermal modeling for 3D ICs with inter-tier liquid cooling," in *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, November 2010, pp. 463–470.

[15] (2017, September) Source code, configuration files, and input data. Embedded Systems Laboratory at Linköping University. [Online]. Available: https://www.ida.liu.se/~ivauk83/research/FSPT

[16] C. Reiss, J. Wilkes, and J. Hellerstein, "Google cluster-usage traces: Format + schema," Google, Tech. Rep., November 2011. [Online]. Available: https://github.com/google/cluster-data

[17] (2017, September) Redis. [Online]. Available: https://redis.io

[18] (2017, September) SQLite. [Online]. Available: https://sqlite.org

[19] I. Ukhov, P. Eles, and Z. Peng, "Temperature-centric reliability analysis and optimization of electronic systems under process variation," *IEEE Trans. VLSI Syst.*, December 2014.

[20] C. Bienia, "Benchmarking modern multiprocessors," Ph.D. dissertation, Princeton University, January 2011.

[21] (2017, September) SPEC CPU2006. Standard Performance Evaluation Corporation. [Online]. Available: https://www.spec.org/cpu2006