BigHouse: A simulation infrastructure for data center systems

David Meisner
meisner@umich.edu

Junjie Wu
wujj@umich.edu

Thomas F. Wenisch
twenisch@umich.edu

Advanced Computer Architecture Lab
The University of Michigan

Abstract

Recently, there has been an explosive growth in Internet services, greatly increasing the importance of data center systems. Applications served from "the cloud" are driving data center growth and quickly overtaking traditional workstations. Although there are a many tools for evaluating components of desktop and server architectures in detail, scalable modeling tools are noticeably missing.

We describe BigHouse *a simulation infrastructure for data center systems. Instead of simulating servers using detailed microarchitectural models, BigHouse raises the level of abstraction. Using a combination of queuing theory and stochastic modeling, BigHouse can simulate server systems in minutes rather than hours. BigHouse leverages statistical simulation techniques to limit simulation turnaround time to the minimum runtime needed for a desired accuracy. In this paper, we introduce BigHouse, describe its design, and present case studies for how it has already been applied to build and validate models of data center workloads and systems. Furthermore, we describe statistical techniques incorporated into BigHouse to accelerate and parallelize its simulations, and demonstrate its scalability to model large cluster systems while maintaining reasonable simulation time.*

1. Introduction

Recently, there has been an explosive growth in Internet services, greatly increasing the importance of data center systems. Ever more applications are served from "the cloud" to mobile devices, quickly overtaking traditional workstations. Whereas there are a large number of tools for evaluating components of desktop and server architectures in detail [3, 6, 8, 22, 34, 37], these detailed modeling tools are not well suited to study data center systems. Conventional architecture simulators are often six orders of magnitude slower than the hardware they model, and simulation turnaround times grow linearly (or worse) with the number of modeled systems and cores. Furthermore, the memory footprint of the simulation is often as large or larger than the simulated workload. Simulating even modest clusters (10's of servers) with such tools is intractable.

This paper introduces *BigHouse*, a simulation infrastructure for data center systems. Instead of simulating servers using detailed microarchitectural models, BigHouse raises the level of abstraction. Using a combination of queuing theory and stochastic modeling, server clusters can be simulated in minutes rather than hours. BigHouse leverages statistical simulation techniques to limit simulation turnaround time to the minimum runtime needed for a desired accuracy. Our initial experiments find that BigHouse is surprisingly accurate and that it can be extended to scale up to large cluster systems while maintaining reasonable simulation time.

BigHouse is based on the *stochastic queuing simulation* (SQS) methodology [25, 27]. Rather than simulate workloads at the granularity of an instruction, memory, or disk access as in conventional simulation tools [6, 8, 22, 34, 37], SQS is built on the theoretical framework of queuing theory, where the fundamental unit of work is a *task* (a.k.a job). Tasks are characterized by a set of statistical properties—random variables that describe their length, resource requirements, arrival distribution, or other relevant properties—which are collected through observation of real systems. SQS abstracts the data center as an interrelated network of queues and power/performance models describing the relevant behaviors of software/hardware components. The discrete event simulation uses a variety of statistical sampling techniques to provide estimates of selected output variables (e.g., 95th-percentile response time) with quantifiable measures of confidence, while enabling parallel simulation to provide strong scaling to reduce turnaround time.

Under pen-and-paper analysis of queuing models, statistics like the moments of the arrival and service distributions are used to calculate performance measures in closed form. We, and others [18], have found that easily-analyzed queuing models (e.g., M/M/1) often poorly represent internet services. More generic models, such as the G/G/1 or G/G/k queue (generalized inter-arrival and service time distribution and either 1 or k servers), have no known closed-form solution. Approximations can be used; however, it has been shown their accuracy is often inadequate, especially when only using a few moments [18]. Instead, with BigHouse, we

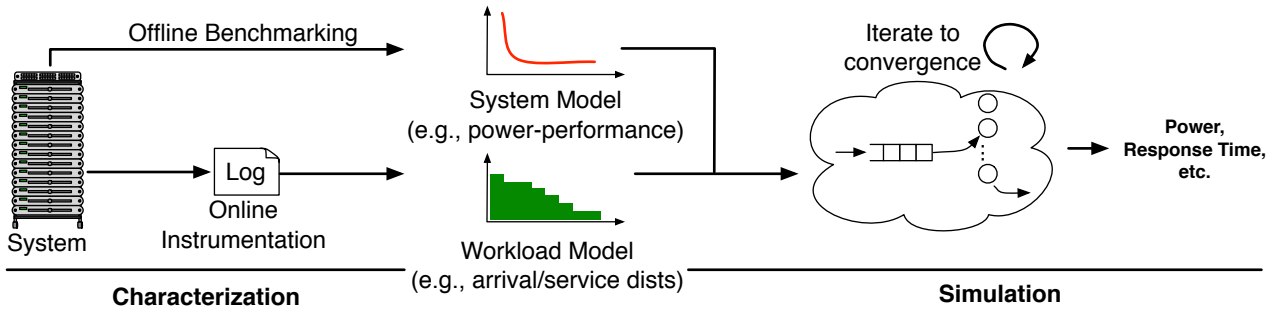

Figure 1: Overview of the BigHouse infrastructure flow. A system is (a) instrumented to derive workload inter-arrival and service time distributions and (b) characterized to create a model of system behavior (e.g., power-performance settings). From these inputs, simulations derive estimates for new system designs and/or configurations.

turn to simulation to exercise these analytically-intractable models. Although this does incur non-negligible simulation time, users of BigHouse need not be experts in queuing theory and models integrated into the framework can be reused.

In this paper, we first describe the software architecture and details of the BigHouse simulation infrastructure, and a general methodology for its use. We analyze factors affecting BigHouse simulation turnaround time, and discuss the mechanisms it uses to ensure desired statistical confidence in its output metrics and parallelize simulations over a cluster of simulation hosts.

BigHouse is best suited for studies investigating load balancing, power management, resource allocation, hardware provisioning, or cost optimization for clustered, distributed, or multi-tier data center applications. BigHouse is not appropriate for studies that require instruction-grain or microarchitectural detail; for example, it is not appropriate for the exploration of cache hierarchy designs (because it does not model individual memory accesses) or compiler optimizations (because it does not execute binaries).

We describe three example studies where researchers have used BigHouse, a published study of the performance implications of power management for Google web search [24], a published study of scheduling mechanisms to coalesce idle periods in manycore systems [26], and a demonstration of how BigHouse could be used to analyze a data-center-wide power capping scheme. The former two case studies have been validated against hardware measurements, while the last demonstrates BigHouse's scalability. In each, we outline the steps taken to construct workload and system models and, where applicable, the validation performed to demonstrate that BigHouse makes reasonable predictions.

2. BigHouse

In BigHouse, the numerous systems comprising a compute cluster are represented as a generalized queuing network, described by the BigHouse user through configuration files and in concise Java code. A task in the queuing model corresponds to the most natural unit of work for the workload under study, such as a single request, transaction, query, and so on. The BigHouse queuing network captures the processing steps through which tasks must proceed at a level of detail appropriate to the question under study. Each server in the queuing network is coupled to power/performance models that modulate the service rate and generate output variables of interest (e.g., task execution time or energy consumption). This model is then exercised on the BigHouse simulation engine, a distributed discrete-event simulator, which samples output metrics and terminates the simulation upon *convergence*—that is, when each output variable has been measured to a desired level of statistical confidence. Figure 1 provides an overview of how BigHouse is used, which comprises two independent steps: (1) characterization of the workloads and system of interest and (2) simulation.

2.1 Software Architecture

The software architecture of BigHouse is divided into two main modules. The first module is used to describe the simulated data center, and comprises a collection of models and events similar to a classic discrete-event simulator. BigHouse uses an object-oriented hierarchy to represent various parts of the data center such as servers, racks, etc. A BigHouse user can either use these existing objects to describe a particular data center architecture, or can extend this hierarchy to model new functionality if more functionality is needed. For example, the server model might be subclassed or extended to include state variables for various ACPI power modes, which modulate task run time, control ACPI state transitions, and output power/energy estimates. Configuration files describe how BigHouse should instantiate and connect these objects and supply parameters such as number of cores, peak power, etc.

The second module of BigHouse orchestrates the simulation. This module includes the BigHouse statistics package, which manages warmup and statistical independence tests, tracks specified output metrics and terminates the simulation when output metric estimates have reached statistical

Table 1: Workload models included with BigHouse.

Workload	Interarrival			Service			Description
	Avg.	σ	C_v	Avg.	σ	C_v	
DNS	1.1s	1.2s	1.1	194ms	198ms	1.0	Departmental DNS and DHCP server under live traffic.
Mail	206ms	397ms	1.9	92ms	335ms	3.6	Departmental POP and SMTP server under live traffic.
Shell	186ms	796ms	4.2	46ms	725ms	15	Shell login server under live traffic, executing a variety of interactive tasks.
Google	319μs	376μs	1.2	4.2ms	4.8ms	1.1	Leaf node in a Google Web Search cluster. See [24] for details.
Web	186ms	380ms	2.0	75ms	263ms	3.4	Departmental HTTP server under live traffic.

convergence. It also provides a communication and control infrastructure to distribute a BigHouse simulation across a cluster of cores and/or machines. Users will typically not modify any of these support modules.

2.2 Workload and System Characterization

To apply BigHouse to implement a particular experiment, a user must either reuse or create system model components that implement models of the salient workload characteristics and output metrics. These models are typically derived from characterizations of actual hardware. Characterization involves both an online and offline component.

Workload Modeling. Instead of using application binaries or traces, as in a traditional simulator, BigHouse represents workloads as empirically measured distributions of arrival and service times for each kind of task in the system. The workload model may also include distributions for other critical task parameters (e.g., tasks' network traffic if modeling network links). Prior BigHouse users have constructed these empirical workload models online, by instrumenting a live system. Typically, this process involves instrumenting a binary such that the timing of task arrivals and their duration are recorded. Later, these traces can be processed to derive the desired distributions. It is necessary to capture these workload models online, under live traffic, because inter-arrival processes depend greatly on the users of an internet service.

BigHouse uses these distributions to generate a synthetic event trace to drive its discrete event simulation. Because it does not use traces or binaries, BigHouse workload models can be represented compactly—a typical distribution occupies less than 1 MB, whereas event traces often require multi-gigabyte files. Furthermore, in contrast to binaries, which industry is often loathe to disseminate, public dissemination of inter-arrival and service distributions is significantly easier, as they do not require releasing proprietary software.

The BigHouse distribution includes five example workload models that have been used in the studies described in Section 3. Table 1 briefly describes each workload. All the workloads BigHouse has been used to study to date follow a request-response model—a client issues a request, the server processes it and after some amount of time, responds. Each workload comprises a pair of distributions, represented via fine-grained histograms: the client request inter-arrival distribution and the response service time distribution. These distributions were all captured on real hardware, using various forms of instrumentation to log arrival and service times. Four workloads (Mail, DNS, Shell, Web) are derived from traces of live traffic to departmental servers. The Google workload was captured by monitoring a web search leaf node in a test cluster while replaying traces of actual search logs; the workload and measurement methodology are described in detail in [24].

Whereas these sample workloads make it easy for a user to get started with BigHouse, it is important to emphasize their limitations. First, they all model simple client-server roundtrip interactions. The BigHouse object model must be extended if a user wishes to model a workload with more complicated communication patterns (e.g., modeling all three tiers of a three-tier web service). Second, the service time distributions have been captured on particular hardware, and may have a different shape on other hardware (e.g., a web server on a Atom-class system may have a markedly different distribution than on a Xeon-class system). Hence, it may not always be valid to simply scale the service time distributions to represent faster/slower hardware.

BigHouse also suffers from the same limitation as other tools that generate synthetic traces [14]: only those correlations among events captured in the input models will be present in the synthetic trace. Though it is possible to exercise the BigHouse discrete-event simulator by replaying traces directly (which eliminates some sampling difficulties, such as sample auto-correlation), it remains unclear how to replay traces and obtain statistically rigorous performance estimates if the simulated system differs substantially from the one where traces are collected. BigHouse's sampling methods are built on the assumption that event sequences are generated synthetically by random draw from the empirical distributions.

System Modeling. To model a particular system, the BigHouse user provides concise Java code that tracks state variables associated with each task processing step and generates output metrics associated with each task. The user-supplied code accepts input tasks (synthesized by BigHouse) and calculates corresponding output metrics. For example, in

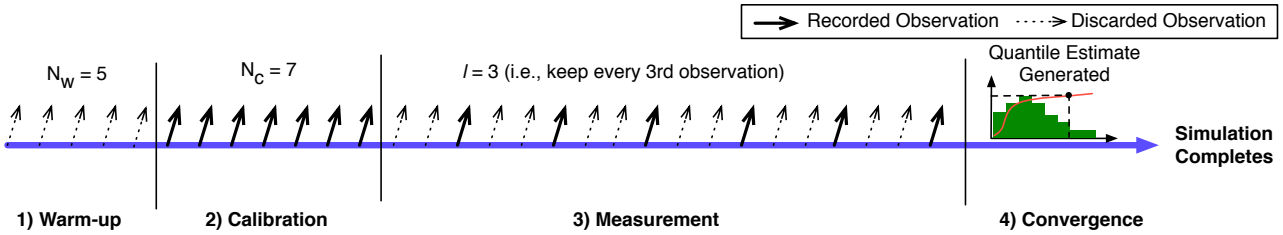

Figure 2: The Sequence of Phases in a BigHouse Simulation: At first, all observations are discarded during warm-up, avoiding cold-start bias. Next, during a brief calibration phase, a small sample is collected to determine the appropriate lag spacing and histogram configuration. The majority of the simulation is spent in the measurement phase, where observations are taken with sufficient spacing to ensure independence. Finally, when the desired statistical confidence is achieved, the simulation terminates, outputting quantile and mean estimates.

an experiment modeling power management in a multi-core server, the input task might be characterized by a size, state variables might track the ACPI power state of each core, while output metrics might include the time and energy consumed by the task. Typically, the relationship between input tasks and output metrics described in system models are derived from offline characterization of a real system. In Section 3, we present concrete examples of system characterization and how these models were incorporated into BigHouse.

2.3 Simulation

The BigHouse simulation engine synthesizes a task trace from the workload models and exercises the user-described queuing network and system models via a distributed discrete event simulation. The core functionality of the BigHouse discrete-event simulator does not differ substantially from other tools for simulating queuing networks. For a detailed survey of queuing models, we refer the reader to [19]. BigHouse augments conventional queuing networks with system models (e.g., the multicore power-performance model mentioned above) and sampling mechanisms that monitor and quantify the confidence of output metric estimates as the simulation proceeds.

For a given simulation, in addition to the data center configuration (e.g., the number of servers, workloads, etc.), the BigHouse user must specify a set of *output metrics*. The simulation's output metrics are aggregations of the per-task metrics generated by the user-supplied system model, which are recorded, analyzed, and reported with statistical confidence estimates. For example, when a task is completed, its response time can be recorded and then aggregated into a mean or quantile output metric. The user specifies each output metric along with a desired accuracy and confidence level for quantile and mean estimates.

Simulation Sequence. BigHouse simulations proceed by exercising the discrete-event queuing simulation, creating task arrival events through random draws according to the distributions captured in the workload model. We refer readers to the literature for details on implementing discrete-event queuing simulations [16]. We focus our discussion on the sampling methods at work in BigHouse, detailing the progression of a simulation from the perspective of an observed output metric (e.g., server response time or power consumption). The phases of a BigHouse instance, illustrated in Figure 2, are:

1. Warm-Up — A simulation begins in an initial *transient state*, where observations are biased by the initial simulation state (e.g., all queues are empty). To avoid this cold-start effect, the simulation must undergo a *warm-up* phase and is exercised for N_w observations, during which all observations are discarded. Unfortunately, a reliable method for determining N_w has been the subject of years of debate [29]. To date, no rigorous method for automatically detecting steady-state is available and N_w must be explicitly specified by the user.

2. Calibration — One of the key challenges that must be addressed when drawing a sample from a discrete event simulation is ensuring independence among the sampled observations. Using successive observations from a queuing-based simulation has been shown to introduce bias into estimates because observations tend to be autocorrelated (i.e., nearby observations are not independent) [11]. However, it has also been demonstrated that if observations are spaced sufficiently apart—by keeping only every lth sample—they can be treated as independent [10]. Determining this minimum spacing, l, is accomplished with the *runs-up* test detailed in [20]. The major consequence of this approach is that steady-state simulation length is inflated by a factor of l. Though a sample size of $N = n$ observations may be sufficient to achieve a given confidence in an i.i.d. draw, since $l - 1$ observations are discarded for every 1 taken, a total of $N = l \cdot n$ events must be simulated to achieve the target sample size. A small caveat is that this method often increases sample variance [12], further increasing n.

During the *calibration* phase, BigHouse performs the runs-up test to determine the *lag spacing*, l, between observations and the proper histogram binning parameters to enable quantile estimates.

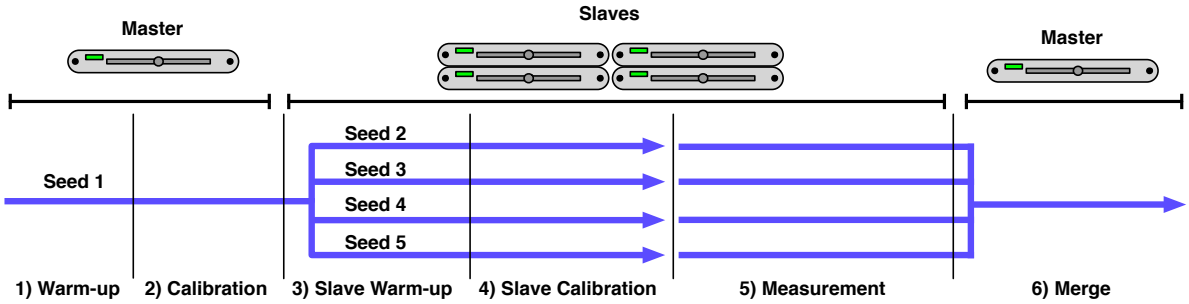

Figure 3: Parallel Execution on a Cluster: First, the simulation undergoes a warm-up and calibration phase on the master. A histogram is generated from the calibration sample and the bin scheme is sent to the slaves. Each slave then executes its own warmup and calibration phase using a unique random seed to achieve steady-state and determine its own lag-spacing. Samples are collected at each slave until their aggregate size is sufficient to achieve the desired accuracy. Finally, in the merge phase, each slave sends its histogram to the master, which aggregates the histograms and reports estimates.

3. Measurement — Once the simulation enters steady-state, observations are collected to populate the histogram representation of each output metric. The majority of simulation runtime is spent in this phase; the other three phases impose insignificant runtime overhead (in single-threaded simulations; however, their overheads can grow dominant in parallel simulations. See Section 2.4).

4. Convergence — An output metric is considered *converged* once the observed sample size is sufficient to achieve the desired confidence interval. If the sample has been generated using distributed simulation (Section 2.4), it is coalesced at this point. Finally, estimates of quantiles and averages can be reported.

Accuracy and Confidence. An estimate of an output metric has an associated *accuracy*, ϵ, and *confidence level*, $1-\alpha$, that together form a *confidence interval*. The value ϵ defines the half-width of a confidence interval in the same units as the output metric (e.g., response time with $\pm$50ms). We normalize this value by the mean estimate, $\bar{X}$, to enable meaningful comparison across multiple output metrics:

$$E = \epsilon / \bar{X} \tag{1}$$

With this definition, a given E describes the desired accuracy as a percentage (e.g., response time with $\pm 5\%$). The confidence level of an estimate describes the expected percentage of estimates that would fall within the confidence interval if the simulation were repeated a large number of times. A confidence level of 95% is common, and we use this value for the remainder of this paper.

To determine the confidence interval for mean estimates (e.g., mean response time), we leverage standard techniques for large-sample analysis. According to the central limit theorem, the sampling distribution of a mean value estimate tends towards the normal distribution as sample size increases. Hence, we can determine the sample size needed for a given confidence by:

$$N_m = \frac{Z^2_{1-\alpha/2} \cdot \sigma^2}{\epsilon^2} \tag{2}$$

Where $Z_{1-\alpha}$ comes from the standard normal: it is the value of the standard normal distribution at the $(1-\alpha/2)^{\text{th}}$ quantile and is 1.96 for 95% confidence. σ is the sample standard deviation and ϵ is the half-width of the desired confidence interval.

Confidence intervals for quantiles (e.g., the 95th-percentile latency) can also be derived using the central limit theorem [10].

$$N_q = \frac{Z^2_{1-\alpha/2} \cdot q(1-q)}{\epsilon^2} \tag{3}$$

The notation is the same as for mean estimates with the addition of q as the desired quantile. To find an exact quantile, one would need to record and sort all observations in the sample. The sample size required for even a single output metric can be quite large. Accordingly, recording and sorting the entire sample sequence to determine quantiles imposes a large burden. However, space-efficient approximations using online algorithms are described in [9, 10]. We use the method presented in [10] to maintain a histogram representation of an observed variable, drastically reducing memory overhead. This method requires the histogram binning parameters to be determined in advance; we do so during the calibration phase of the simulation sequence (see below).

Typically, it is useful to know both the mean and at least one quantile of a given output metric. In this case, the required sample size for the desired confidence will be N=max(N_m, N_q).

Observing Multiple Output Metrics. Typically, multiple metrics are observed in a single simulation. For simplicity, we have explained a sequential procedure and illustrated the sequence in Figure 2 in terms of a single output metric. However, it is important to understand that there is an

associated sequence for each output metric in the simulation. There are two important constraints on the simulation progression when targeting multiple outputs. First, the simulation may not progress out of the warm-up phase until N_w observations have been collected for all output metrics. This constraint ensures that measurement does not take place until the entirety of the model is warm. Second, the simulation may not terminate until *all* outputs have a sufficient sample size to reach convergence. Again, the slowest convergence will determine simulation runtime.

2.4 Parallel Simulation

The procedure of a distributed BigHouse simulation is outlined in Figure 3. A simulation cluster comprises a single *master* and many *slave* machines. First, the master executes just the warmup and calibration phase of a serial BigHouse simulation. After calibration, the master constructs the appropriate histogram bin structure, which is forwarded to the slaves.

Next, the master broadcasts the histogram setup and simulation configuration and each slave begins their own BigHouse instance. Each slave must use a unique seed for their random number generator. The BigHouse process at the slave is nearly identical to a single-machine BigHouse simulation, requiring warmup, calibration and steady-state measurement, except that the slave's calibration phase does not determine the histogram setup. Also, slaves do not determine when the simulation converges; the master monitors the slaves' progress and signals convergence when aggregate sample size is sufficient across the entire cluster.

Once the aggregate sample is large enough, the master collects all the histograms and combines them to form a single estimate. In a number of ways, the master-slave relationship resembles the MapReduce framework [13]—a single program is executed with high fan-out across a number of slave machine (map) with different inputs (the random seed). After completion, their results are then merged (reduce) to form a result.

3. Validated Case Studies

In this section we provide examples of experiments using BigHouse that have been validated against real hardware. These case studies demonstrate that BigHouse can provide accurate server performance estimates.

3.1 Power Management in Google Web Search

Our first case study is taken from [24] in which we used BigHouse to understand the performance effect of processor and memory low-power modes for Google Web search. The goal of the study was to understand how to achieve energy-proportional operation in Web search servers, while maintaining reasonable latency. Using BigHouse, we were able to predict the effects of intrusive experiments such as changing server performance states.

As described in Section 2.2, empirical arrival and service distributions must first be collected for the workload. We obtained the inter-arrival distribution by instrumenting a production binary to track the arrival sequence of live traffic. To measure the service time distribution, queries were injected one-at-a-time into an isolated Web search node, thereby ensuring no queuing within the search node. Accordingly, query service times could be measured simply as the the difference between the finish and arrival time of the query. The distributions were both measured over large samples of queries.

We next measured the power-performance behavior of the isolated search node to construct a system model. Through offline experimentation, we varied processor frequency and memory latency and measured the resulting average service time for an individual query at each point in the space of processor and memory performance settings. From this data, we built a BigHouse system model that modulated service times and reported power estimates for each query.

Finally, with these characterization steps complete, BigHouse can be used to estimate the effects of power management policies under various loads. Load can be varied by scaling the inter-arrival distribution. Figure 4 provides a partial view of the validation of the BigHouse simulation results. Lines represent 95th-percentile latency as predicted by BigHouse and points represent measured data from real hardware. Because the experiment varies both processor and memory settings, the performance setting space is 2-dimensional; the figure shows a subset with fixed memory performance and variable CPU performance. The horizontal axis shows utilization (in terms of percentage of maximum queries per second or QPS) varying over the typical operating range. The average error across all validated points is 9.2%.

To underscore the advantage of BigHouse's empirically measured workloads and simulation over pen-and-paper analysis, it is useful to contrast alternative models for the system's inter-arrival distribution. Figure 5 reports the normalized 95th-percentile latency measured on real hardware as a function of QPS load for three different inter-arrival distribution scenarios. For "Low C_v", queries arrive at a near-uniform rate with little variance. The "Exponential" series represents an exponentially distributed inter-arrival process; a common assumption when analyzing queuing models by hand. Results using the actual inter-arrival distribution measured in production are shown by "Empirical"; the actual distribution has greater variance than either synthetic distribution. The important trend in this example is that poor assumptions about inter-arrival times (made in the interest of expedient pen-and-paper analysis) can lead to large estimation errors.

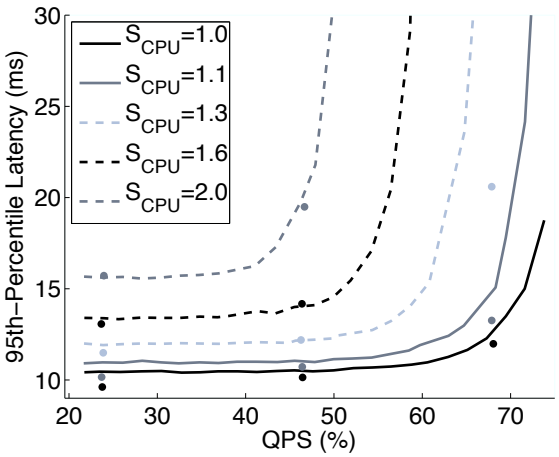

Figure 4: Validation of Google Web search performance scaling. Points depict measured 95th-percentile latency of Web search on production hardware at various performance settings. S_{CPU} is the relative processor slowdown. Lines are latency as predicted by BigHouse. Overall error is 9.2%. Data from [24].

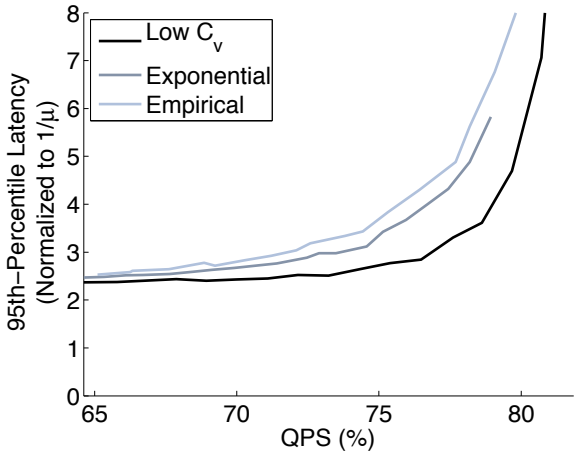

Figure 5: Inter-arrival distribution has a large effect on latency. While an exponential distribution is typically assumed in analytic modeling, it clearly differs form the latency observed with empirically measured distributions. Similarly low C_v distributions (used by many loadtesters), do not reflect real traffic accurately. Data from [24].

3.2 Scheduling for Idleness

In our second case study, BigHouse was used to study DreamWeaver [26], a scheduling mechanism that seeks to coalesce idle periods to enable the use of idle low-power modes (e.g., PowerNap [23]) in many-core servers. The scheduling mechanism is designed to align idle and active times across all cores as much as possible, to maximize the intervals where all cores are idle and the entire system can be put into a deep sleep mode. The essence of the scheduling mechanism is to preempt execution and enter deep sleep if there are fewer outstanding tasks than cores. However, if any task is delayed by more than a pre-specified threshold, the system wakes up and execution resumes even if some remain idle. In essence, the technique trades per-request latency to create opportunities for deep sleep.

We used BigHouse to predict the effectiveness of this scheduling mechanism and validated the BigHouse predictions against a software implementation of the scheduling mechanism for the Solr open source Web search system [4]. Solr is a full-featured web indexing and search system used in production by many enterprises to add local search capability to their web sites. The validation experiment exercised Solr with the AOL query set [1] and an index of Wikipedia [2]. The scheduling mechanism and sleep/wake transition delays were implemented in a BigHouse system model, and arrival and service distributions for Solr were captured using an approach similar to that described for the Google web search case study (see Section 3.1). The search latency distribution from the software prototype and the BigHouse model were then compared.

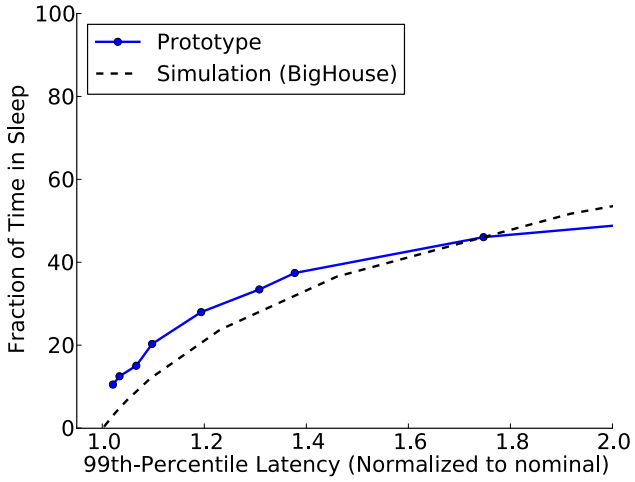

Figure 6: Validation of scheduling for idleness. The scheduling mechanism trades latency for idleness by intelligently delaying and preempting requests. This figure illustrates the fraction of time an entire server spends idle as a function of 99th-percentile latency. The "Prototype" series depicts measurements of the schedule mechanism implemented on real hardware and the "Simulation" series shows BigHouse-derived results. Data from [26].

Figure 6 illustrates the validation results. The Figure shows the fraction of time the system is idle (either naturally or as forced by the scheduler) plotted against 99th-percentile query latency, which both vary as a function of the pre-specified maximum per-task delay threshold (i.e., the tuning knob of the scheduling mechanism). The "Prototype" series shows the measured results of the software implementation

running on real hardware, whereas "Simulation (BigHouse)" shows the BigHouse-derived estimates. There is a close correspondence between the simulated and measured results.

4. Simulation Performance

We next report on the performance of BigHouse in terms of simulation turnaround time and scalability. Through discussion of a hypothetical experiment built on top of BigHouse, we demonstrate the idiosyncrasies of the framework that affect runtime. We explore how BigHouse performance changes as we scale up the size of the modeled cluster.

4.1 Extending BigHouse — An example

Power capping is technique that allows a data center to deploy more servers than its provisioned power infrastructure can support at peak. It has been observed that—especially in large installations—servers rarely draw peak power concurrently [15, 21, 30, 35]. Because a cluster's aggregate power draw is typically significantly less than the potential sum of all its servers' peak power, provisioning the number of servers based on peak power is wasteful. Although we do not have access to thousand-server clusters, we demonstrate that simulating such systems is feasible with BigHouse. We hope this demonstration motivates future research leveraging this framework.

To amortize the high cost of power infrastructure, it is desirable to provision servers based on their average power consumption. While such a scheme might work in the common case, rare power spikes across many machines do occur, which can exceed the provisioned capacity of the power infrastructure, tripping a circuit breaker and taking the cluster offline. Power capping solves this problem by assigning hard limits, or "caps", to each server's power consumption. These limits are enforced by throttling a server's performance thereby reducing its power consumption.

To evaluate the ability of BigHouse to simulate large-scale systems, we use it to model a dynamic power capping scheme for a large cluster populated with quad-core servers. Our case study uses a relatively simple power capping scheme; we wish to demonstrate the scalability of BigHouse system rather than explore sophisticated power capping strategies. Servers are assigned a *power budget*, the maximum power they may draw over a given interval. We use a fair, proportional budgeting mechanism such that every server gets a budget in proportion to its utilization in the previous budgeting interval. Budgets are calculated every second. At each budgeting epoch, the *capping* level can be observed and is defined as how much more power a server would draw, beyond its budget, without a cap. We assume idealized DVFS as the power-performance throttling mechanism. The salient feature of this power capping scheme (from the point of view of measuring BigHouse performance) is that it is global—the system models within BigHouse must interact each simulated second to determine each server's power cap each epoch. Hence, the behavior of all system models are coupled.

Power-Performance Model. To simulate power capping, we must implement a system model for power savings and performance loss under DVFS. We use the linear model validated by [15] and [31]:

$$P_{\text{Total}} = P_{\text{Dynamic}} \cdot U + P_{\text{Idle}} \tag{4}$$

Where U is the average server utilization, P_{Dynamic} represents the dynamic range of the server's power, and P_{Idle} the idle power. Our power model is based on typical server specification from industry [5]. For simplicity, we assume that the CPU is the only component with a dynamic power range:

$$P_{\text{CPU}} \propto \left(\frac{f}{f_{\text{Max}}}\right)^3 \tag{5}$$

Where f is the operating clock frequency of the CPU. We assume that this frequency can be continuously scaled from $f = 1.0$ to $f = 0.5$, even though in practice these setting are discrete. The exact scaling of DVFS with respect to frequency has been receiving increasing scrutiny [7]; however, since our focus is on simulator performance rather than power capping efficacy, we assume the classic cubic scaling.

Next, we require a performance model to understand the slowdown imposed by various DVFS settings. The slowdown in service rate due to DVFS can be modeled as:

$$\mu' = \mu \cdot \alpha \cdot \left(\frac{f}{f_{\text{Max}}}\right) + \mu \cdot (1 - \alpha) \tag{6}$$

For some α, which represents how "CPU-bound" an application is. We assume an α of 0.9, which would be typical of a CPU-intense application (e.g., LINPACK).

The power-performance model described here is a simple example of the kind of model that can be integrated into BigHouse; the particular details of this model are not critical to the simulation approach.

Unless otherwise specified, all simulations are run to achieve 95% confidence of E=.05 for both the average value and a 95th-percentile quantile.

In Figure 7 we demonstrate how simulation time scales with the size of the simulated cluster. Simulation of a ten-server system is trivial, taking no longer than a minute to converge at the desired confidence. As we increase the number of servers, simulation time increases roughly linearly. While simulation time across our workloads varies, the scaling relationship is the same. Even at three orders of magnitude greater cluster size (10,000 servers), simulations take hours rather than days.

It is important to note that the primary cause of increased simulation time is the overhead of maintaining and updating

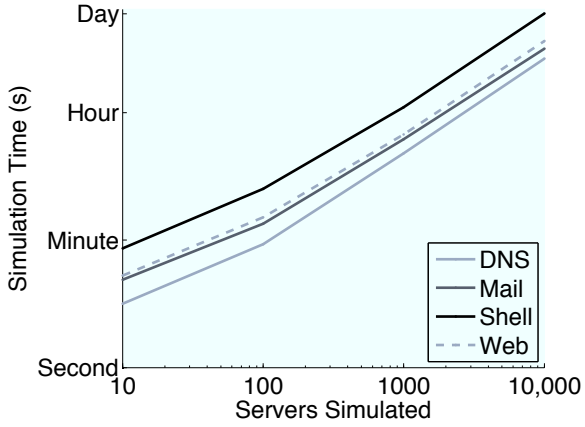

Figure 7: Simulation Time Scaling: Simulation time required for convergence scales roughly linearly with the number of servers simulated. Scaling simulation size typically does not increase the variance of the output variables, so the required sample size does not increase significantly. Instead, the overhead of maintaining the discrete-event-simulation state is the main cause of increased runtime.

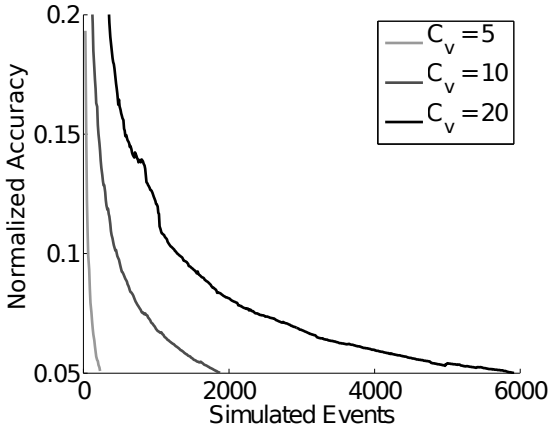

Figure 8: Sensitivity to Workload Distribution Variance: Increasing service distribution coeff. of variation (C_v) leads to increased variance in the target variables, requiring a disproportionate increase in simulation time.

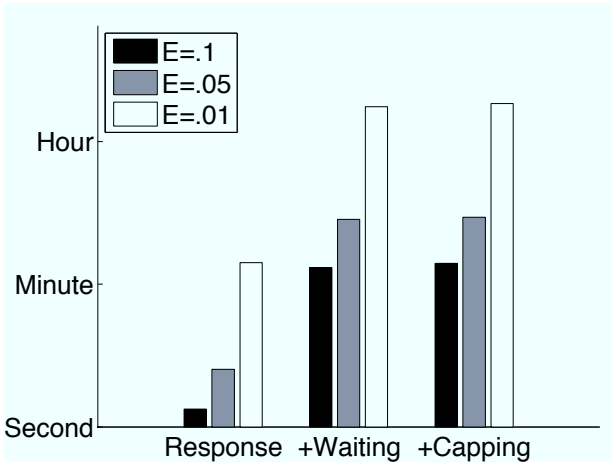

Figure 9: Sensitivity to Accuracy and Target Metrics: Runtime is affected by the selected output metrics and desired confidence intervals. Monitoring response and waiting time (+Waiting) increase simulation time over monitoring response time alone, because most requests do not experience queuing, which makes "waiting" observations more rare. Additionally including power capping as an output metric (+Capping) further increases runtime because capping epochs occur less frequently than request completions.

the enlarged state of the discrete-event simulation. The sample size required for convergence, however, depends only on the variance of the output metrics and may be reduced slightly due to averaging effects in larger clusters (as in the case of power capping).

For a given system size, simulation time is strongly dependent on the workload model. To understand this effect, we simulate a system where the workload's service distribution is adjusted to a desired coefficient of variation, C_v (the standard deviation normalized by the mean). We use the response time as the sole output metric because it is most dependent on the C_v parameter. Figure 8 shows how the accuracy of an output metric, E, reaches a target value of .05 with the number of simulated events for three values of C_v. For larger values of E, the difference in the number of simulated events across values of C_v is small; however, at .05, the required number of simulated events becomes pronounced. This phenomenon is a direct implication of Equations 2 and 3—simulation time increases quadratically with increased accuracy and the standard deviation of the worst case across output metrics. In our example, the C_v of the service time strongly affects response time variance; however, in more complex systems the relationship may not be as clear.

Finally, we evaluate how the selected output metrics impact runtime. We use the same power capping system as before, but vary the set of output metrics and their desired accuracy. First, we monitor only response time ("Response"). Increasing the desired accuracy drastically increases runtime, but simulations require at most a few minutes. Adding a wait time ("+Waiting") output metric greatly increases runtime. This increase occurs because wait events are much less frequent than request completion events (i.e., queuing is relatively infrequent). Finally, additionally monitoring power capping ("Capping") results in a further, slight increase in runtime (note that results are on a log scale).

4.2 Parallel Simulation Performance

We demonstrate the ability for BigHouse to parallelize across multiple slaves using our power capping example.

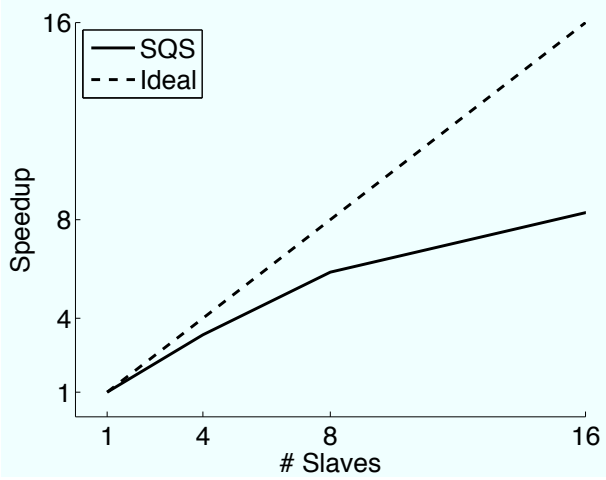

Figure 10: Parallel Simulation: BigHouse achieves speedup by parallelizing measurement across multiple slaves. The primary limiting factor to parallel scalability is the calibration phase, which requires 5000 observations for the runs-up test on each slave. Since this simulation requires a sample size just under 40,000, calibration imposes an Amdahl bottleneck.

We run the simulation with $E = .01$ so that it is sufficiently long to gain benefit from parallel execution. Figure 10 demonstrates the speedup gained by using an increasing number of slaves. We distribute the slaves across 4 hosts such that each host has an equal number of slaves (e.g., with 8 total slaves, each host has 2 slaves).

A system with perfect parallel scaling would achieve a speedup equal to the number of slaves ("Ideal"). BigHouse demonstrates good scaling up to 8 slaves ("SQS"), but Amdahl effects limit scalability beyond 16 slaves. Each slave must execute a 5000-observation calibration phase to complete the runs-up test. As this particular simulation problem requires a sample size around 40,000, calibration overhead becomes dominant beyond 16 slaves.

5. Related Work

Previous studies have attempted to parallelize discrete-event simulations by executing different sections of the modeled system at the same time [17, 28]. Generally, such parallelization is difficult because the system must have a consistent state and requires explicit communication and/or locking of data structures. In contrast, our parallelization strategy, which distributes generation of independent observations for sampled output metrics, does not require synchronization, greatly reducing design complexity and communication overhead.

Our work bears similarity to architectural simulators that use statistical simulation [14, 32] and/or sampling techniques [33, 36, 37, 38]. These methods also provide significant reduction in simulation time by either simulating with a statistical abstraction and/or by simulating only those events necessary for the desired level of statistical confidence.

6. Conclusion

We have introduced BigHouse, a simulation infrastructure for data center systems. By raising the level of simulation abstraction, BigHouse can model servers or clusters significantly faster than traditional microarchitectural simulators. We describe two studies that have validated BigHouse-derived results against hardware and find that the accuracy of the system is quite good. Finally, we analyze the scalability of BigHouse to demonstrate that it can be used to study large cluster systems.

Acknowledgments

The authors would like to thank the anonymous reviewers for their feedback. This work was supported in part by grants from Google, ARM and NSF grants CNS-0834403, CCF-0811320, and CCF-0815457.

References

[1] AOL Query Log, 2006. `http://www.gregsadetsky.com/aol-data/`.

[2] A Solr index of Wikipedia on EC2/EBS. 2010. `http://hublog.hubmed.org/archives/001917.html`.

[3] The Network Simulator - ns-2, 2010. `http://www.isi.edu/nsnam/ns/`.

[4] Welcome to Solr, 2011. `http://lucene.apache.org/solr/`.

[5] L. A. Barroso and U. Hölzle. The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines. *Synthesis Lectures on Computer Architecture*, Morgan-Claypool, 2009.

[6] N. Binkert, R. Dreslinski, L. Hsu, K. Lim, A. G. Saidi, and S. K. Reinhardt. The M5 simulator: Modeling networked systems. *IEEE Micro*, v.26 n.4, Jul. 2006.

[7] D. Blaauw, S. Das, and Y. Lee. Managing variations through adaptive design techniques, ISSCC Tutorial, 2009.

[8] J. S. Bucy and G. R. Ganger. The DiskSim Simulation Environment Version 3.0 Reference Manual, 2003.

[9] E. J. Chen and W. D. Kelton. Simulation-Based Estimation of Quantiles. In *Proc. of the Winter Simulation Conf.*, 1999.

[10] E. J. Chen and W. D. Kelton. Quantile and histogram estimation. In *Proc. of the Winter Simulation Conf.*, 2001.

[11] E. J. Chen and W. D. Kelton. Determining simulation run length with the runs test. *Simulation Modelling Practice and Theory*, 11(3-4), 2003.

[12] R. Conway. Some Tactical Problems in Digital Simulation. *Management Science*, 10(1), 1963.

[13] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *Proc. of the 6th USENIX Symp. on Operating System Design and Implementation,* 2004.

[14] L. Eeckhout, S. Nussbaum, J. Smith, and K. De. Statistical simulation: Adding efficiency to the computer designer's toolbox. *IEEE Micro*, v.23 n.5, Sep. 2003.

[15] X. Fan, W.-D. Weber, and L. A. Barroso. Power provisioning for a warehouse-sized computer. In *Proc. of the 34th International Symp. on Computer Architecture*, 2007.

[16] G. S. Fishman. *Discrete-event simulation*. Springer-Verlag, 2001.

[17] R. Fujimoto. Parallel Discrete Event Simulation. In *Proceedings of the Winter Simulation Conf.*, 1989.

[18] V. Gupta, M. Harchol-Balter, J. G. Dai, and B. Zwart. On the inapproximability of M/G/K: why two moments of job size distribution are not enough. *Queueing Systems: Theory and Applications*, 64(1):5–48, Aug. 2009.

[19] M. Harchol-balter. Theory of Performance Modeling, CMU Class Notes, 2005.

[20] D. Knuth. *The Art of Computer Programming - Voll II.* Addison-Wesley, 1981.

[21] C. Lefurgy, X. Wang, and M. Ware. Power capping: a prelude to power shifting. *Cluster Computing*, 11(2): 183–195, Nov. 2008. ISSN 1386-7857.

[22] M. M. K. Martin, D. J. Sorin, B. M. Beckmann, M. R. Marty, M. Xu, A. R. Alameldeen, K. E. Moore, M. D. Hill, and D. A. Wood. Multifacet's general execution-driven multiprocessor simulator (GEMS) toolset. *ACM SIGARCH Computer Architecture News*, 33(4):92, Nov. 2005.

[23] D. Meisner, B. T. Gold, and T. F. Wenisch. PowerNap: Eliminating Server Idle Power. In *Proc. of the 14th International Conf. on Architectural Support for Programming Languages and Operating Systems*, Feb. 2009.

[24] D. Meisner, C. M. Sadler, L. A. Barroso, W.-D. Weber, and T. F. Wenisch. Power Management of Online Data-Intensive Services. In *Proc. of the 38th International Symp. on Computer Architecture*, 2011.

[25] D. Meisner and T. F. Wenisch. Stochastic Queuing Simulation for Data Center Workloads. In *Exascale Evaluation and Research Techniques Workshop*, 2010.

[26] D. Meisner and T. F. Wenisch. DreamWeaver: Architectural Support for Deep Sleep. In *Proc. of the 17th International Conf. on Architectural Support for Programming Languages and Operating Systems*, 2012.

[27] D. Meisner, J. Wu, and T. F. Wenisch. Towards a Scalable Data Center-level Evaluation Methodology. In *Proc. of the International Symp. on Performance Analysis of Systems and Software*, 2011.

[28] D. M. Nicol. Parallel Simulation Of FCFS Stochastic Queueing Networks. In *Principles and Practice of Parallel Programming*, 1988.

[29] K. Pawlikowski. Steady-state simulation of queueing processes: survey of problems and solutions. *ACM Computing Surveys (CSUR)*, 22(2), 1990.

[30] P. Ranganathan, P. Leech, D. Irwin, and J. Chase. Ensemble-level Power Management for Dense Blade Servers. In *Proc. of the 33rd International Symp. on Computer Architecture*, May 2006.

[31] S. Rivoire, P. Ranganathan, and C. Kozyrakis. A comparison of high-level full-system power models. In *HotPower*, 2008.

[32] B. Schroeder, E. Pinheiro, and W. Weber. DRAM errors in the wild: a large-scale field study. In *SIGMETRICS*, 2009.

[33] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically Characterizing Large Scale Program Behavior. In *Proc. of the 10th International Conf. on Architectural Support for Programming Languages and Operating Systems*, October 2002.

[34] D. Wang, B. Ganesh, N. Tuaycharoen, K. Baynes, A. Jaleel, and B. Jacob. DRAMsim: a memory system simulator. *ACM SIGARCH Computer Architecture News*, 33(4):107, 2005.

[35] X. Wang, M. Chen, C. Lefurgy, and T. Keller. SHIP: Scalable hierarchical power control for large-scale data centers. In *Proc. of the 18th International Conf. on Parallel Architectures and Compilation Techniques*, 2009.

[36] T. F. Wenisch, R. E. Wunderlich, B. Falsafi, and J. C. Hoe. Simulation sampling with live-points. In *Proc. of the International Symp. on Performance Analysis of Systems and Software*, 2006.

[37] T. F. Wenisch, R. E. Wunderlich, M. Ferdman, A. Ailamaki, B. Falsafi, and J. C. Hoe. SimFlex: Statistical Sampling of Computer System Simulation. *IEEE Micro*, 26(4), 2006.

[38] R. E. Wunderlich, T. F. Wenisch, B. Falsafi, and J. C. Hoe. SMARTS: accelerating microarchitecture simulation via rigorous statistical sampling. In *Proc. of the 30th International Symp. on Computer Architecture*, 2003.