Future Generation Computer Systems 64 (2016) 15–38

Contents lists available at ScienceDirect

Future Generation Computer Systems

journal homepage: www.elsevier.com/locate/fgcs

Robust partial-load experiments with Showstopper

Andrej Podzimek [a,c,*], Lubomír Bulej [a,c], Lydia Y. Chen [b], Walter Binder [a], Petr Tůma [c]

[a] Faculty of Informatics, Università della Svizzera italiana, Switzerland
[b] IBM Zurich Research Lab, Switzerland
[c] Department of Distributed and Dependable Systems, Charles University in Prague, Czech Republic

ARTICLE INFO

Article history:
Received 5 December 2015
Received in revised form
3 April 2016
Accepted 30 April 2016
Available online 19 May 2016

Keywords:
Partial load
Resource sharing
Performance evaluation
Resource isolation
Load control
Reproducible load

ABSTRACT

Concurrent execution of multiple applications leads to varying partial utilization of shared resources. Understanding system behavior in these conditions is essential for making concurrent execution efficient. Unfortunately, anticipating behavior of shared resources at partial utilization in complex systems is difficult, realistic experiments that reproduce and examine such behavior are therefore needed.

To facilitate experiments at partial utilization, we present a tool that accurately controls the processor utilization of arbitrary concurrent workloads, either establishing constant partial load or replaying a variable load trace. We validate the ability of the tool to enforce the configured partial utilization on multiple platforms, and use the tool to collect novel information on system behavior at partial utilization levels.

In detail, our experiments show how to examine the complex relationship between utilization and throughput, useful for tasks such as performance debugging or system dimensioning, and we show this relationship for the DaCapo benchmarks. Further, we show that CPU pinning (a technique used to improve workload isolation) can benefit from dynamic response to system utilization, improving system efficiency with partial utilization. Finally, we show that the overhead of virtualization also changes with partial utilization and CPU allocation.

© 2016 Elsevier B.V. All rights reserved.

1. Introduction

For reasons ranging from efficiency to maximizing achievable performance, concurrent execution of multiple workloads is the norm for most computer systems today. Workload colocation has been researched extensively, with a number of results spanning several decades—for example, Chase et al. [1] achieves demonstrated energy savings of 29% and projected energy savings of as much as 78% on a web workload.

To support concurrent execution efficiently, essential system features such as task scheduling and resource allocation have to deal with resource sharing, which influences multiple aspects of performance [2–4]. There are numerous possible interactions—workloads running on a contemporary server can influence each other through mechanisms such as simultaneous multithreading [5], same page merging [6], frequency boosting with thermal budgeting [7], and more.

The multitude of possible resource interactions is difficult to capture comprehensively—research advances therefore often tackle selected interaction aspects individually. In this context, abstracting away from certain interactions is essential to keep the research problems tractable. At the same time, evaluating the impact of such simplification on research results becomes equally necessary—a recent case study by Corradi et al. [8] in fact warns that the growing distance between the theoretical assumptions and the real system behavior may make some of the existing research results rather difficult to apply.

A straightforward method for assessing the impact of theoretical simplifications is experimental validation. Unfortunately, evaluating resource interactions in realistic conditions is an expensive undertaking. Resource utilization depends on workload type in complex ways and even if we can control the workload, it is not always clear what workload intensity to use to achieve a particular resource utilization. Our work contributes a tool that simplifies experiments with varying partial utilization of shared resources. Given an arbitrary executing workload, the tool observes a resource utilization metric of choice – processor utilization in our case – and throttles the workload to achieve the target utilization. This removes the need for implementing workload generation harnesses with configurable workload intensity, and makes it

* Corresponding author at: Department of Distributed and Dependable Systems, Charles University in Prague, Czech Republic.
E-mail address: podzimek@d3s.mff.cuni.cz (A. Podzimek).

http://dx.doi.org/10.1016/j.future.2016.04.020
0167-739X/© 2016 Elsevier B.V. All rights reserved.

possible to examine system behavior at a particular utilization directly. The modular tool design and the available experiment configuration options are described in Section 3, the ability to enforce configured partial utilization is validated in Section 4.

After introducing and validating the tool design, we use it to present novel results on the behavior of computing systems under partial utilization. In Section 5, we look at the complex relationship between workload intensity and processor utilization. Our results provide a more accurate empirical alternative to operational analysis, which is typically used to relate utilization and throughput [9]. In Section 6, we look at how partial utilization impacts the practice of pinning workloads to selected processor cores. Against common intuition, our results show that the most efficient and most performant pinning configurations change with utilization. Finally, in Section 7, we briefly show how partial utilization influences virtualization overhead. These results are particularly relevant to cloud computing environments, where partial utilization and virtualization is often the norm.

This article constitutes a comprehensive presentation of our work on experimental evaluation of partially utilized systems, and combines and extends previously published conference material. Our extensions focus both on adding more depth to existing results, as well as on entirely new experiments—ultimately providing a more complete picture that explains the need for partial-load experiments along with compelling experimental results.

Our contributions can be summarized as follows:

- We present a comprehensive description of the Showstopper tool used to conduct experiments at partial CPU utilization. The basic concepts and the high-level architecture of Showstopper were introduced in our prior work [10,11], based on a preliminary version of the tool. Thanks to its modular architecture, the tool has been significantly extended to provide a wider range of control and dithering mechanisms, and to provide support for measurement of system load inside Linux LXC containers.
  Here we present a detailed description of the architecture and the various building blocks of the control algorithm, along with guidelines for choosing the values of critical parameters. We validate the ability of Showstopper to enforce and maintain the desired partial utilization, and extend the previously published results with quantitative evaluation of accuracy of different configurations of the load-control algorithm.
- We introduce a novel methodology to examine the relationship between processor utilization and application throughput, and present experimental results that illustrate the complex nature of this relationship. We extend our prior work [10] by introducing fast-forwarding of load traces replayed by Showstopper, and evaluate the effect of fast-forwarding on the accuracy of throughput measurements. We also significantly extend the workload and platform coverage.
- We provide a novel study of the impact of CPU pinning on performance and energy efficiency for pairs of colocated workloads. Our experiments with different CPU pinning configurations at varying levels of background load expose a trade-off between workload isolation and overall system performance. We also show that the performance increase due to pinning configuration at certain background loads increases the overall energy efficiency of the system. Here we complement the results presented in our prior work [12] with a study of the overhead observed in the employed colocation solutions (KVM virtual machines and LXC containers) at partial loads.

2. Motivating challenges

In this section, we demonstrate several challenges that motivate our work on experiments with partial utilization. While some of our examples aim for brevity, we would like to stress that the presented challenges get even more complex in typical computing systems with a deep software stack and sharing of hardware among colocated virtual machines.

2.1. Controlling utilization

Resource utilization metrics serve as the basis for monitoring and managing the quality of service provided by a computer system. Services such as Amazon Cloudwatch [13] are commonly available to clients of cloud computing providers, and many resource management policies are defined in terms of resource utilization. Combined traces of virtual and physical resource utilization are commonly collected on contemporary production systems for both physical and virtual machines. These traces provide insight into allocation and sharing of resources and aid in developing resource management policies [14,15]. Fig. 1 shows an example of such a trace, capturing the average processor load over a four-hour period, sampled in 15-min intervals.

Given a trace that describes realistic resource utilization conditions, an experiment that seeks to validate theoretical results and assumptions in the same conditions will need to achieve the same utilization trace. Typically, the experiment would also rely on available benchmarks as examples of realistic workloads [16–20]. Many benchmarks, however, are designed to execute in isolation, with all system resources at their disposal. In contrast, reproducing realistic utilization conditions requires a partial and coordinated use of system resources. Even benchmarks that aim to limit the system load [21] do so by controlling application level metrics (typically problem size or request rate) rather than by observing and responding to system level resource utilization. The whole problem becomes even more complex if we consider sustaining a given partial load (see footnote 1) while executing multiple benchmarks in parallel.

To illustrate the difficulties, we explore two solutions applicable to achieving partial loads with existing benchmarks: (1) executing multiple benchmarks in parallel so that the number of benchmarks divided by the number of processors equals the desired load, and (2) executing a benchmark with a constant duty cycle[1] corresponding to the desired load. These solutions also serve as baselines for our approach.

We evaluate the two baseline solutions on six DaCapo [22] benchmarks with a default iteration time short enough to make throughput estimates from short time intervals (tens of seconds) feasible. The selected benchmarks capture various types of multi-threaded behavior—ranging from an almost single-threaded execution (`fop`, `luindex`), through a moderate multi-threadedness failing to saturate all available processors (`avrora`, `h2`), to heavily multi-threaded computations capable of consuming all CPU capacity (`sunflow`, `xalan`). Generally speaking, the key difference between moderately and heavily multi-threaded benchmarks lies in the overall frequency of synchronization among threads. While threads in moderately multi-threaded benchmarks synchronize and block each other frequently (forming e.g. producer–consumer relationships or participating in *rendezvous* events), threads in heavily multi-threaded benchmarks spend most of their time working on well isolated subproblems of a parallel computation, without the need to block and synchronize with each other frequently.

Our goal is to achieve partial loads of 33% and 75% using the above methods. For method (1), this means executing 4 and

[1] We use the terms *duty cycle* and *partial load* in an intuitive fashion throughout this section. A more formal definition of both quantities and their interpretation by our load control tool is given in Section 3.

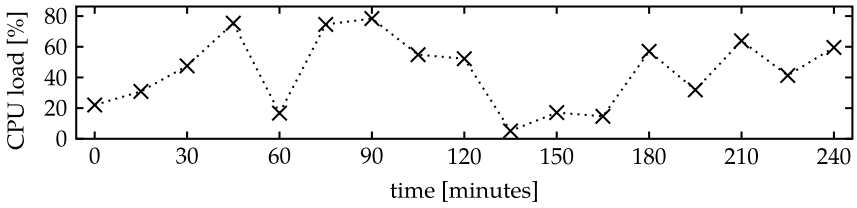
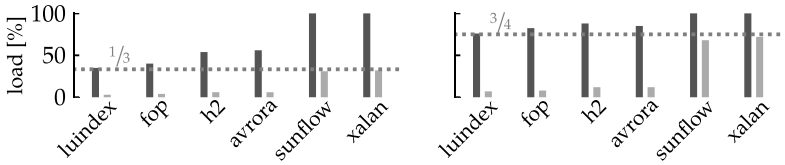
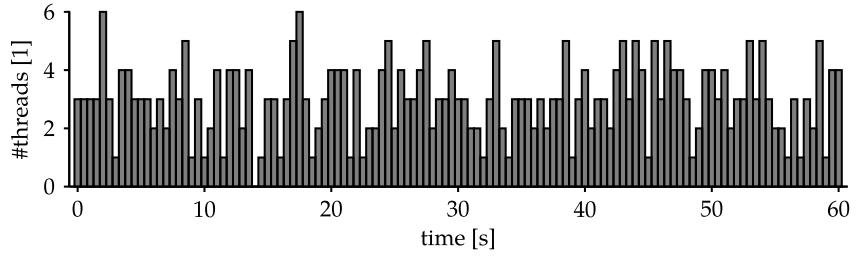


Fig. 1. An example of a trace from a production system hosting financial services. The trace shows samples of 15-min CPU utilization averages.

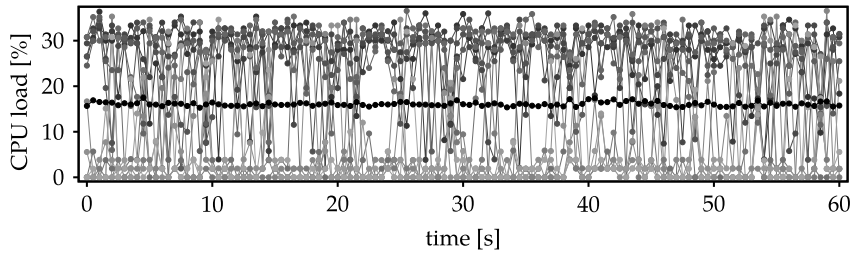


(a) Target load of 1/3. (b) Target load of 3/4.

Fig. 2. The desired target CPU load (dotted line), and the CPU load averages achieved by choosing the number of benchmarks (dark), and by enforcing the duty cycle of a single benchmark (light).



(a) Instantaneous number of runnable threads, sampled once per second.



(b) Average (•) and per-CPU (• • • • • • • • • • • •) utilization.

Fig. 3. Selected run-time characteristics of a single avrora benchmark on an Intel system with 12 processors.

9 benchmarks, respectively, on our 12-processor machine. For method (2), we keep a single benchmark runnable for 1.33 and 3 s, respectively, out of each 4-s period.

Fig. 2 shows the baseline solutions having trouble achieving the target loads. Controlling the number of benchmarks works better for single-threaded benchmarks, such as fop and luindex, while controlling the duty cycle of a single benchmark favors highly multi-threaded benchmarks, such as sunflow and xalan. This suggests that the difficulty in achieving a partial load condition lies in the parallelism in benchmarks, and that a proper solution may need to control the number of benchmarks and their execution states simultaneously.

To further substantiate the complexity of controlling modern benchmarks on a multicore system, we show the run-time thread dynamics and processor utilization for a single avrora benchmark in Fig. 3(a) and (b), respectively. We observe that the number of runnable threads changes over time, and although utilization also fluctuates over time, there is no obvious dependency between the number of runnable threads at a given point in time and CPU utilization around that time. This is typical for benchmarks whose threads block and synchronize extremely frequently (e.g. avrora). The necessity of controlling multiple benchmarks concurrently further exacerbates the difficulty of achieving partial loads. Section 4 shows how our solution succeeds at enforcing partial loads.

2.2. Utilization and workload intensity

Although useful in their own right, resource utilization traces are essentially disconnected from key application performance metrics, such as response time and throughput. For example, Martinec et al. [23] have shown that the throughput of the ActiveMQ JMS broker can more than double when CPU utilization increases from 80% to 95%, with critical overload still safely away at almost three times the throughput at 80% utilization.

Collecting application performance metrics may require workload profiling or intrusive platform-dependent instrumentation, which are generally shunned on production systems. Leveraging the abundance of system-level monitoring data to infer application-level performance metrics with reasonable overhead and accuracy is therefore a useful practical challenge.

Fig. 4 shows an example relationship between a system-level and an application-level metric, plotting the application throughput against the processor load for the avrora benchmark from the

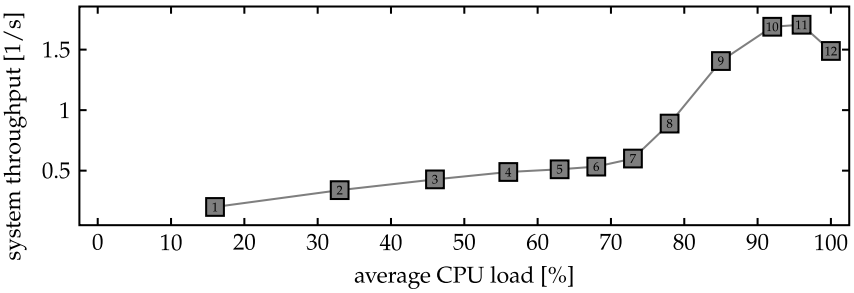


Fig. 4. The relationship between load and throughput: executing different numbers of `avrora` benchmarks on an Intel system.

DaCapo suite. The relationship was measured by launching a varying number of benchmark instances in parallel on an Intel system with 12 logical processors.[2] Obviously, the dependency of throughput on system load is non-linear, even non-monotonic, explaining why a utilization trace alone does not tell much about application performance, and why estimating application throughput by a simple method, such as interpolation based on a single throughput observation at peak load, is not accurate.

We observe that even more complex methods for describing the relationship between response time, throughput and resource utilization – such as operational analysis of queueing networks [9] – are not sufficient for inferring detailed application-level performance metrics from system-level utilization traces collected with little information about the applications. More complex methods for determining performance metrics for consolidated applications with complex and dynamic workloads [24–26] typically require a workload to be characterized using parameters obtained by application profiling—such as the number of requests (transactional applications), or the size of inputs (computational applications).

As described in Section 5, our method of enforcing partial loads for arbitrary applications provides means for efficient construction of empirical models that relate utilization and throughput for a given application executing on a chosen platform. Besides providing an important insight into application behavior, knowing the relationship is important for practical tasks such as resource provisioning. Our solution works on real-world systems, including both virtual and physical machines—without the need to adjust the system or the benchmarks. This is especially useful when examining systems with modern power and thermal management functions activated; most of them are typically disabled in common benchmarking exercises.

2.3. Utilization and resource allocation

Resource utilization is also an important indicator of efficiency—in terms of work done per energy unit consumed, a server tends to be more efficient at higher utilization [27]. This observation stands among major reasons for workload colocation, a technique that increases energy efficiency by trading it against certain other aspects of performance that can suffer due to resource sharing. In server clusters, colocation plays a significant (but neither sole nor entirely sufficient [28]) role in improving energy efficiency.

Colocating workloads with coinciding resource demand peaks increases the risk of performance anomalies due to workload interference [29]. Although this risk can be reduced by colocating workloads with complementary workload intensities, techniques to increase workload isolation are also required. On multi-core, multi-processor, and especially NUMA platforms, one approach to increasing isolation is processor pinning, where workloads are simply confined to a subset of available processors [27,30]. Unfortunately, the effect of pinning depends on workload type and system architecture, adding further dimensions to an already difficult resource allocation problem.

The effect of pinning on performance interference has not been extensively studied—perhaps because pinning is generally considered useful only for heavily-loaded systems, and because common-sense pinning guidelines may appear sufficient. After all, if we consider cache sharing to be the primary source of performance interference for many workloads, then simply pinning such workloads to different cores, CPU packages, or NUMA nodes should obviously reduce cache sharing and improve per-CPU memory locality, potentially improving performance. However, to the best of our knowledge, the effect of CPU pinning on energy efficiency in general and on performance at other than peak CPU utilization has not been studied yet.

In Section 6, we use our method of enforcing partial loads to analyze the effect of CPU pinning on performance interference and energy efficiency for pairs of colocated workloads, simulating a scenario with a single workload executing at peak utilization in presence of a background workload executing at varying levels of partial utilization. We show that while the effect of CPU pinning meets the common sense expectations concerning performance isolation, the effects on performance (in terms of aggregate throughput) and energy efficiency can be surprising. In particular, we find that unexpected CPU pinning configurations improve energy efficiency at partial background loads, indicating that systems hosting colocated workloads could benefit from dynamic CPU pinning based on CPU load and workload type.

3. Showstopper experimental tool

To support experiments with varying partial utilization of shared resources, we develop Showstopper—a tool that controls the execution of multiple applications in a distributed fashion so as to follow a prescribed CPU load trace. Showstopper controls the duty cycle of the executing processes by alternating their runnable and stopped states, employing feedback control mechanisms to translate CPU load requirements into duty cycles to sustain the desired partial load. Showstopper is implemented in user space, and is completely hardware-agnostic, which makes it suitable for both physical and virtual machines.

3.1. CPU loads and duty cycle

To describe the Showstopper design, we first introduce the necessary terminology. In the following, we make a clear distinction between CPU loads, L, and duty cycles, D, which are control variables and outputs of Showstopper, respectively.[3]

[2] Hardware configuration details are given in Section 4.1.

[3] Showstopper can use other control variables than CPU load, but this option is not investigated here.

The term *CPU load* denotes the fraction of time processors spend working (on average) over a sampling interval, the length of which determines the accuracy of CPU load observations. At any given moment, a CPU is either doing useful work (100% utilization), or idling (0% utilization); other values of CPU load stem from averaging observations of the two extremes over a sampling interval and over multiple CPUs. For example, assume we observe a 4-CPU system for 1 min, finding two CPUs idling all the time and the other two CPUs working in the first 30 s and idling afterwards. The CPU load average is 25% for the entire minute, but 50% and 0% for each half-minute. In summary, CPU load is not a continuous function and the choice of the sampling interval is critical.

The term *duty cycle* denotes the fraction of time a benchmark is *runnable* (permitted to run by the load control mechanism), over a control epoch. With multiple benchmarks executing in parallel, the time for which a benchmark actually runs can be shorter than what the duty cycle allows—the OS kernel might schedule benchmarks in runnable state in many different ways and not necessarily on multiple processors. Also note that because the number of runnable threads in a benchmark changes during execution, the execution time of a benchmark is not equivalent to the total time the processors spend working. Similar to the sampling interval above, the choice of control epoch duration is critical for both accuracy and stability of Showstopper's load control.

In summary, Showstopper controls the duty cycles D_i, $i \in [1, N]$, of N benchmarks so that the induced CPU load, L, follows the target load value, L^*.

3.2. Showstopper design

At the process level, the architecture of Showstopper consists of a master process and a set of executor processes, one for each benchmark. The master process initializes shared memory areas for data exchange and keeps track of the executors it spawns. An executor starts the corresponding benchmark, repeatedly invokes the control algorithm to determine the target duty cycle for its benchmark, and enforces the duty cycle by controlling the duration of the benchmark's runnable and stopped states using standard POSIX signals, SIGSTOP and SIGCONT.

The architecture of Showstopper controlling two benchmarks is shown in Fig. 5, along with the modular structure of an executor. The solid arrows indicate API invocations (and thus dependencies) among modules. The modules themselves – trace, feedback, space, control and dithering– are responsible for different aspects of the control algorithm.

During operation, the difference between the current target load dictated by the trace module and the current load estimate obtained from the feedback module is used by the control module to compute the global average duty cycle to be enforced by Showstopper. To determine the contribution of individual benchmarks to the global duty cycle, the space module calculates a local duty cycle for each benchmark so that the average of local duty cycles matches the global duty cycle. The dithering module then transforms the local duty cycle into durations of runnable and stopped states to be enforced by an executor. We now elaborate on each module in turn.

3.2.1. Trace: replaying load traces

A trace module determines the load trace for Showstopper to follow. It can be implemented in many different ways, and is only required to provide an up-to-date target load value whenever requested. Showstopper provides modules that can generate synthetic traces on the fly, such as a sine-shaped load trace for testing, as well as a module that replays a file-based trace in real time.

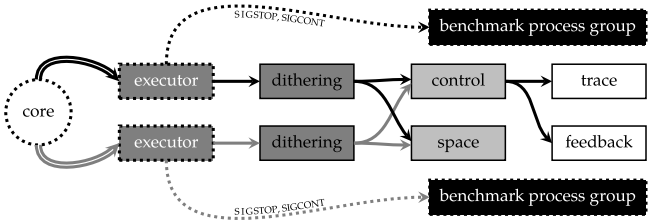


Fig. 5. Showstopper controlling two benchmarks. Rectangles with dotted and solid borders represent processes and Showstopper modules, respectively. Double lines represent the POSIX process hierarchy. Dotted lines indicate communication via POSIX signals. Solid lines represent library calls within a process.

The file-based trace contains records mandating a particular CPU load at a particular time, and allows replaying load traces from other systems—obtained by sampling the CPU load periodically. The target load for moments falling between two trace file records is approximated by linear interpolation.

The module can be also instructed to scale all time intervals in a trace file by a given factor, which allows reproducing a trace in a fast-forwarding fashion. We note that for this technique to provide meaningful results, the amount of "work" (the total number of benchmark iterations) accomplished while fast-forwarding a trace should be proportional to the time scaling factor. If the proportionality requirement is not met, it may be impossible to draw any conclusions about the original full-length run from its fast-forwarded reproduction. Situations that fail to meet the proportionality requirement can arise, for example, from load traces with steep and abrupt changes or short periodic patterns or from benchmarks with load characteristics changing based on wall clock time or in a strictly monotonic fashion.

3.2.2. Feedback: estimating current CPU load

The choice of a load estimation method is critical for the stability of the load control algorithm. After evaluating several variants, we choose the method below, which leads to the most stable load control in our experiments.

The feedback module calculates CPU load estimates on demand —as a weighted average of the preceding estimate and a new CPU load sample. This results in a series, where the nth load estimate, L_n, is calculated as

$$L_n = (1 - \omega_n)L_{n-1} + \omega_n \frac{T_n^w}{T_n^i + T_n^w}, \tag{1}$$

with T_n^w and T_n^i representing the times all processors spent working or idling, respectively, since the preceding request. The new CPU load sample is a ratio of average time spent working to total time elapsed between the $(n-1)$th and nth estimate.

To account for the variable sampling interval duration, the weight ω_n of the new sample depends on the ratio of a constant base period, δ, to the duration of the sampling interval (time elapsed since the preceding request):

$$\omega_n = 1 - \frac{\delta}{\delta + T_n^w + T_n^i}. \tag{2}$$

Consequently, load samples reflecting shorter intervals are assigned a lower weight and vice versa. In particular, a load sample from an interval of length δ has the same weight as the accumulated estimate L_{n-1}. The value of δ allows balancing between responsiveness and stability of the load estimate. A longer base period δ results in a slower response to quick changes in load, but provides a more stable load estimate.

Because a feedback module operates in each executor process, it is beneficial to share the latest CPU load estimates among executors. This avoids frequent fetching and processing of CPU load

data from the operating system when multiple executors query the load estimate within a very short time interval. (We set this interval to $\frac{160}{P}$ ms on a system with P CPUs.) Additionally, the data sharing also makes the series of load estimates produced by different executors' feedback modules less prone to random noise and transient mutual differences, both of which could worsen Showstopper's load control stability. The shared latest load estimate is accompanied by a time stamp, so that the ω_n value can be calculated when re-using and updating the estimate.

3.2.3. Control: global duty cycle control

The control module implements a feedback controller responsible for achieving and sustaining the target load. The Showstopper architecture allows many controller types to be implemented—a control module only needs to calculate a global duty cycle for all controlled benchmarks in response to the target and current CPU load. We evaluate two controller types.

The first controller, denoted geom, is a simple geometric growth/decay controller, which uses the previous global duty cycle, D_{n-1}, as the base value, and the ratio of the target load, L_n^*, to the estimate of the actual load, L_n, as the growth factor. The next global duty cycle, D_n, is calculated based on D_{n-1} as

$$D_n = D_{n-1}\frac{L_n^*}{L_n}. \tag{3}$$

The fact that the duty cycle cannot grow past 100% prevents severe stability problems, and while the controller remains somewhat unstable, it is surprisingly effective. This controller type is used to throttle a process to a particular fraction of CPU time by tools such as cpulimit [31], or to control the time sharing among groups of processes at the OS level in the Illumos kernel [32].

The second controller, denoted PI, implements a discretized proportional–integral controller. The difference between the target load, L_n^*, and the estimated actual load, L_n, represents the tracking error, $e_n = L_n^* - L_n$. The sum of tracking errors weighted by the time elapsed between the $(n-1)$th and nth updates represents the error accumulated over time, $I_n = I_{n-1} + e_n(T_n^w + T_n^i)$. The next global duty cycle, D_n, is then calculated as

$$D_n = k_e e_n + k_I I_n, \tag{4}$$

where the coefficients k_e and k_I determine the contribution of the respective term to the result. After extensive evaluation, we set the coefficients to $k_e = k_I = 1$, which outperformed our attempts at automatic on-line tuning using a discrete adaptation of the Ziegler–Nichols method.

The output of the controller is susceptible to fluctuations in the actual load estimates, which influence the tracking error. To improve resilience to such fluctuations, each executor's control module stores its most recent output into shared memory for other executors' control modules to consult. The final outcome is an average of the local result and the most recent results from κ control modules in other executors, which prevents the independent and mutually unsynchronized controllers from diverging too far apart in their duty cycle estimates.

3.2.4. Space: distributing work spatially

A space module determines the contribution of individual benchmarks to the global duty cycle produced by a control module. Given the number of benchmarks, N, and a global duty cycle, D_n, the module calculates the local duty cycles, D_n^i, so that

$$\frac{\sum_{i=1}^{N} D_n^i}{N} = D_n.$$

Showstopper provides even and uneven variants of the module, which correspond to different workload types. The even variant keeps the contribution of all benchmarks even, assigning the same duty cycle to all benchmarks, which corresponds to a server machine handling many requests in parallel. The uneven variant uses as few benchmarks as possible to meet the average, which resembles a long-running computation able to fully utilize a group of processors for extensive periods of time, leaving other processors idle.

For example, given a global duty cycle of 50% over three benchmarks, the even module variant will assign a duty cycle of 50% to all benchmarks, while the uneven variant will use the duty cycles of 0%, 50%, and 100%.

3.2.5. Dithering: distributing work temporally

A dithering module determines how to distribute the work of a benchmark over time according to its assigned duty cycle. Specifically, the module calculates the durations of the benchmark's runnable and stopped states for an executor to enforce.

To convert a unit-less duty cycle to time, we use a base quantum, τ, which represents the average duration of an epoch for enforcing the local duty cycles. Given the duty cycles of the ith benchmark, D_n^i, the module calculates the durations of subsequent runnable and stopped states as $D_n^i\tau$ and $(1 - D_{n+1}^i)\tau$, respectively.

We use this particular implementation because it results in an almost constant number of context switches per unit of time across a wide range of duty cycles. Other implementations may focus on capturing different workload characteristics, such as burstiness (when servicing asynchronous client requests), regularity (when processing a steady data stream from an input device) or combinations thereof.

3.2.6. Feedback base periods (δ), dithering quanta (τ) and control peers (κ)

The choice of Showstopper's parameters (such as the feedback base period, δ, the dithering base quantum, τ, or the number of peer control modules for duty cycle averaging) is mostly an empirical process based on trial and error. We summarize the basic guidelines we follow while searching for a set of parameters that yields an acceptable stability of load control across the entire set of experiments presented in this article.

Dithering quanta are chosen to be on the same order of magnitude as scheduling quanta commonly used by contemporary kernels, i.e., 10 s of milliseconds. (Historically, these quanta have always been set slightly below the smallest time interval distinguishable by most humans.) Our settings lean slightly towards longer quanta in order to generate as few context-switches as possible, minimizing the overhead induced by Showstopper in comparison to a "naturally" occurring partial CPU load. Additionally, we avoid setting a round interval duration (such as 100 ms) in order to prevent undesired "harmonic" interference with regular bookkeeping that may take place in the controlled benchmarks or on the system at large. We repeat our experiments with at least two dithering base quanta (107 ms and 53 ms), obtaining practically identical results each time.

Feedback base periods set the tradeoff between stability of CPU load observations and their responsiveness to load changes. In general, the more CPUs a system has, the faster it accumulates CPU load data (such as profiling ticks or microstate accounting cycles) over time and the shorter its feedback base periods can be without introducing an excessive amount of noise. Values too high (e.g. units of seconds) would cause the control modules to oscillate between 0% and 100% of duty cycle at a very low frequency, due to the delay between the control modules' actions and the observation of their outcome. Values too low would supply the control modules with random noise rather than consistent load

observations. A value of 333 ms appears to be acceptable for all experiments we run.

Dithering quanta and feedback base periods should be "on par" with each other. Long dithering quanta justify long feedback base periods, because decisions made on each of Showstopper's SIGSTOP and SIGCONT events impact a relatively long period of time ahead; therefore the source of feedback information should be stable and based on a longer observation. Short dithering quanta (in general) require a faster response of the feedback module to CPU load changes, in order to avoid slow (yet extreme) oscillations in the control module. Although it is feasible to choose dithering quanta in the millisecond range, the duration of feedback base periods has a lower bound dictated by the pace at which the given system can gather meaningful CPU load information (hundreds of milliseconds in most cases).

The κ parameter of the control module should be increased as long as it improves control stability on the particular workload configuration. In our experiments, we set $\kappa = 2$ for the PI controller and $\kappa = 3$ for the geom controller, independently of the number of controlled benchmarks. Higher values of κ do not bring any further advantage in terms of load control stability.

4. Experiment: load control accuracy

Our first experimental section focuses on the ability of Showstopper to accurately maintain desired load levels. We use selected benchmarks from the DaCapo suite as the controlled workloads—the selection includes benchmarks with an average iteration duration short enough (less than 10 s, approximately) to obtain stable throughput estimates, and excludes those with built-in time limits (such as tradesoap, tradebeans, and tomcat), which tend to expire when the benchmark is throttled.

For the sake of brevity, we only present selected (representative) results for the avrora, fop, h2, luindex, sunflow, and xalan benchmarks. These can be roughly classified into three categories: fop and luindex have a low degree of multi-threading; h2 and avrora are mildly multi-threaded (a single instance is unable to saturate our machines); xalan and sunflow are highly multi-threaded parallel benchmarks. Detailed benchmark descriptions can be found in [22].

To measure the load control accuracy, we monitor system load with a 500 ms period using cpuload[4] and calculate the control error as the difference between the target load and the load reported by cpuload at the same moment. The resulting error values fall into the interval of $[-1, 1]$, and are used to calculate a *mean error* and a *mean squared error* metrics. Mean errors, denoted $\varnothing\mathbf{e}$, are reported as load percentage (multiplied by 100%), and indicate control accuracy. Mean squared errors, denoted $\varnothing\mathbf{e^2}$, are multiplied by 10 000, and indicate control stability, or how much the instantaneous load observations fluctuate around the target load.

4.1. Platform configuration

We present results of load accuracy experiments run on a physical system denoted PM1. PM1 is an IBM System x3650 M4 server with 32 GB of RAM and one Intel Xeon E5-2620 processor (6 cores, 12 logical processors) at 2.0/2.5 GHz, running Fedora 18 with kernel version 3.7.8 and OpenJDK 1.7.

In many performance-related experiments, the underlying hardware platform is often configured to provide the most stable performance results. This usually involves turning off many advanced features, such as symmetric multi-threading or power and thermal management with advanced sleep states, frequency scaling or boosting. This is not required by our approach—to mimic a realistic production environment, we make no hardware configuration adjustments and leave all advanced features enabled.

However, when running the benchmarks, we take steps to avoid start-up transients in the measurements. All benchmarks are granted at least 3 min of warm-up time before load control and load observations begin. We also configure the benchmarks to store their data on a RAM disk, to avoid disk contention and related phenomena, because we only focus on CPU load.

4.2. Controller configurations

We evaluate four Showstopper configurations, corresponding to combinations of the geom and PI variants of the control module, and the even and uneven variants of the space module. To improve resilience to fluctuations in the estimates of actual load, the PI and geom controllers are configured with $\kappa = 2$ and $\kappa = 3$, respectively.

The feedback and dithering modules are configured with a base period $\delta = 333$ ms and a base quantum $\tau = 107$ ms, respectively. Shorter base quanta yield a better apparent load control accuracy, but increase the scheduling overhead. We emphasize that the choice and configuration of the dithering module should reflect the workload Showstopper is supposed to mimic.

Where applicable, we also evaluate the two baseline methods mentioned in Section 2.1. The first, here denoted N-bench, achieves partial loads by executing N benchmarks in parallel so that the ratio of the number of benchmarks to the number of processors equals the target load. The second, here denoted CDC, achieves partial loads by executing as many benchmarks as there are processors, and enforcing a constant duty cycle equal to the target load, without any feedback. These methods represent our baselines, because without Showstopper, they would have to be used to achieve partial loads.

In general, we denote the benchmark runs managed by Showstopper as controlled and the runs using the baseline methods as uncontrolled.

4.3. Results: constant load traces

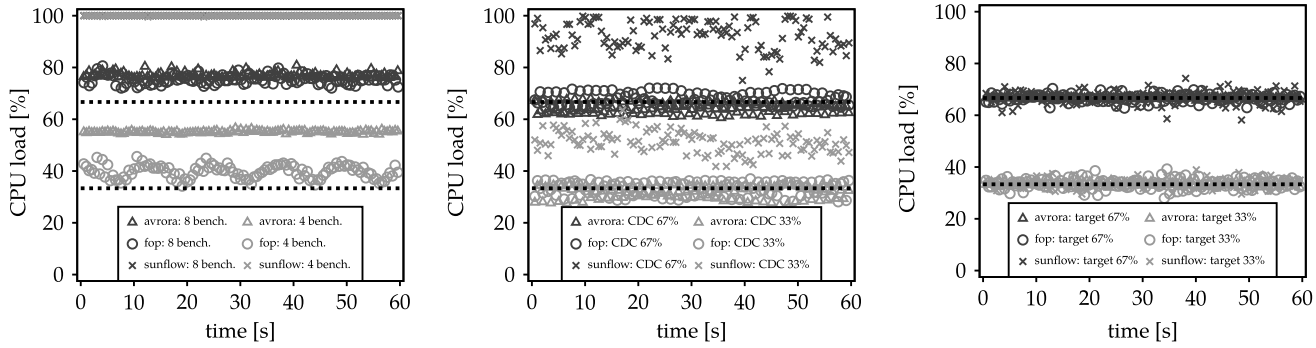
The first set of experiments evaluates the load control accuracy with 3-min constant-load traces. The measurements were performed for target loads of 25%, 33%, 50%, 66%, and 75% on all platforms, but here we only present results for the PM1 platform, and the fop, avrora, and sunflow benchmarks.

To illustrate the difference between Showstopper and the baseline methods, we plot the load observed during 1 min of execution for target loads of 33% and 66% with N-bench, CDC, and Showstopper, in Figs 6(a), (b), and (c), respectively. The results for Showstopper correspond to the configuration with the PI control module and the even space module.

The N-bench method is clearly inappropriate for heavily multi-threaded benchmarks such as sunflow, which saturates the processors completely with only 4 instances. The method works better with benchmarks exhibiting low to mild multi-threading, such as fop and avrora, but the observed load still exceeds the target in both cases.

The CDC method appears to work reasonably well for fop and avrora—there are no extreme deviations from the target load, only signs of harmonic interference. In the case of sunflow, the CDC method manages to exert some control over the benchmark, but the observed load still significantly exceeds the target. This is because the runnable periods of the benchmark instances can

[4] A tool we developed solely for efficient CPU load monitoring—the average of collected load samples is identical to outputs from standard tools, e.g., sar and mpstat.



(a) N-bench with 4 and 8 benchmarks. (b) CDC with 12 benchmarks. (c) Showstopper with 12 benchmarks.

Fig. 6. Comparing observed vs. target loads of 33% and 66% on a 12-processor Linux system (platform PM1).

Table 1
Load-control accuracy for target loads of 25%, 33%, 50%, 66%, and 75% on a 12-processor Linux system (platform PM1).

		Uncontrolled baseline				Showstopper runs with feedback control							
		N-bench		CDC		geom, uneven		geom, even		PI, uneven		PI, even	
		$\varnothing e$	$\varnothing e^2$	$\varnothing e$	$\varnothing e^2$	$\varnothing e$	$\varnothing e^2$	$\varnothing e$	$\varnothing e^2$	$\varnothing e$	$\varnothing e^2$	$\varnothing e$	$\varnothing e^2$
25%		−5.3	31.6	−0.13	4.67	−0.05	1.78	0.68	6.5	0.008	1.98	−0.14	3.63
33%		−6.79	53.8	0.03	6.26	−0.03	2.54	0.66	8.31	0.005	1.67	−0.15	5.25
50%	fop	−9.47	95.2	−0.13	7.54	−0.06	3.76	0.004	11.9	−0.08	2.82	−0.03	3.49
66%		−8.74	81.3	−0.87	6.41	−0.06	3.32	−0.04	5.76	−0.003	3.07	0.05	2.74
75%		−7.38	59.3	−1.57	6.68	−0.04	3.03	−0.08	3.79	0.002	2.04	0.02	2.56
25%		−20.8	433.0	0.94	4.72	−0.03	0.65	0.42	3.34	−0.003	0.47	−0.17	0.77
33%		−21.7	472.0	1.39	7.36	0.01	0.46	0.05	3.11	−0.008	0.21	−0.12	0.56
50%	avrora	−17.9	321.0	1.89	10.4	0.03	0.44	0.07	1.15	0.003	0.33	−0.003	0.2
66%		−10.8	121.0	2.38	11.2	−0.03	0.36	0.003	0.19	−0.002	0.25	0.005	0.13
75%		−9.77	97.5	1.55	6.35	−0.02	1.1	−0.02	0.33	0.006	0.74	0.004	0.2
25%		−74.8	5600.0	−49.9	2620.0	−0.53	56.2	−0.06	5.38	0.66	21.3	−0.19	2.58
33%		−66.6	4440.0	−18.7	375.0	−1.43	103.0	−0.06	7.28	0.72	31.7	−0.18	2.84
50%	sunflow	−50.0	2500.0	−15.3	266.0	−1.71	128.0	−0.03	13.9	−0.45	12.8	−0.16	4.0
66%		−33.3	1110.0	−25.6	685.0	−2.21	205.0	−0.14	26.6	−0.64	13.5	−0.24	7.74
75%		−25.0	625.0	−25.0	625.0	−1.91	188.0	−0.26	28.4	−0.05	60.0	0.04	11.3

overlap, using more processor capacity due to multi-threading than the duty cycle might suggest.

In contrast, Showstopper can dynamically adjust the duty cycle as the benchmarks change their behavior, still allowing their runnable periods to overlap randomly, similarly to real-life workloads. Consequently, the target load is achieved and sustained with greater accuracy and without a systematic error.

For target loads of 25%, 33%, 50%, 66%, and 75%, we summarize the results in Table 1. The mean error for the N-bench method reveals a strong bias towards higher loads even for single-threaded benchmarks, such as fop. The CDC method achieves reasonable accuracy and stability with the fop and avrora benchmarks, but is unable to accurately control the load generated by sunflow.

The Showstopper runs exhibit a considerably higher control over the loads produced by the benchmarks, regardless of the controller variant. The PI controller generally provides better stability and usually slightly better accuracy, compared to the geom controller.

For single-threaded and mildly multi-threaded benchmarks such as fop and avrora, the uneven work distribution tends to provide a slightly better accuracy than the even variant; however, each of them corresponds to a different workload type (see Section 3.2.4). The even variant brings the advantage of an equal duty cycle for all benchmarks.

For the heavily multi-threaded sunflow benchmark, the geom controller exhibits considerable instability with uneven work distribution, and a slightly better accuracy with even work distribution compared to the PI controller. However, the PI controller exhibits superior stability overall, even though it tends to get worse as load increases.

4.4. Results: variable load traces

The second set of experiments evaluates the load control accuracy with one particular 1620-s trace, which records a series of nine load averages over 180 s periods. We replay the trace in 1080, 540, 360, 180 and 90 s, and evaluate Showstopper using the four controller configurations (see Section 4.2).

We do not evaluate the baseline load control methods, because N-bench cannot be used with variable-load traces. The measurements were performed on all platforms, but here we only report results for platform PM1 for the sake of brevity.

4.4.1. Results for 12-processor Linux system (PM1)

To illustrate load control on a variable-load trace, we plot the load observed while fast-forwarding a trace with the sunflow and avrora benchmarks in Fig. 7. Showstopper was configured to use the PI controller with even work distribution to control the load. The top row shows the original trace replayed in 1080 s, while the bottom row shows the trace replayed in 540 s, 360 s, and 180 s. The target load corresponds to the center of the grey band.

The results for a variable-load trace follow the general trend observed with the constant-load traces—controlling a heavily multi-threaded benchmark such as sunflow is more difficult and therefore the result is less stable. In contrast, the controlled load produced by the avrora benchmark follows the trace very precisely, without significant fluctuations.

For replay times of 1080 s, 360 s, and 90 s, we summarize the results for the four Showstopper configurations in Table 2. Overall, Showstopper follows the load trace accurately (mean error within

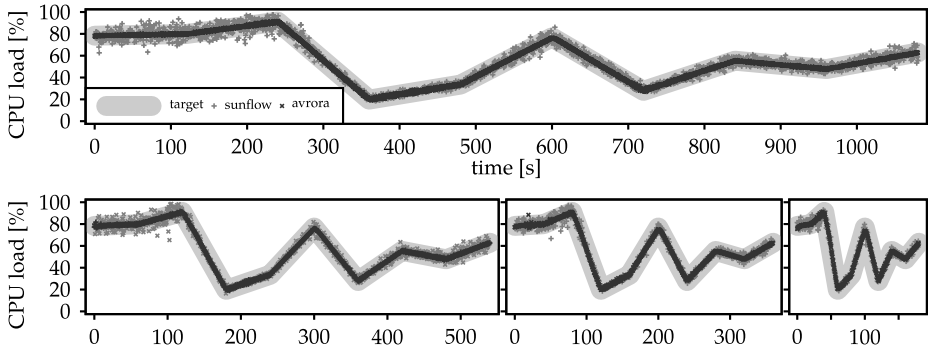


Fig. 7. Comparing observed vs. target load while fast-forwarding a load trace on a 12-processor Linux system (platform PM1). The center of the wide grey band represents the CPU load target.

Table 2
Error metrics of replaying loads on Linux using six different benchmarking times.

		geom, uneven		geom, even		PI, uneven		PI, even	
		$\varnothing\mathbf{e}$	$\varnothing\mathbf{e}^2$	$\varnothing\mathbf{e}$	$\varnothing\mathbf{e}^2$	$\varnothing\mathbf{e}$	$\varnothing\mathbf{e}^2$	$\varnothing\mathbf{e}$	$\varnothing\mathbf{e}^2$
90 s	fop	−0.03	1.78	0.12	6.22	−0.02	1.57	−0.02	2.88
180 s		−0.03	1.96	0.14	6.85	−0.007	1.8	−0.05	3.19
360 s		−0.02	2.27	0.14	6.95	−0.02	2.06	−0.01	3.36
540 s		−0.04	2.33	0.09	7.1	−0.005	2.4	−0.06	3.4
1080 s		−0.04	3.09	0.13	8.87	−0.004	3.03	−0.02	3.46
1620 s		−0.03	3.17	0.07	9.2	−0.06	3.53	−0.05	5.11
90 s	avrora	−0.009	0.47	0.09	0.96	−0.01	0.34	−0.04	0.31
180 s		−0.01	0.5	0.19	1.09	−0.02	0.4	−0.06	0.36
360 s		−0.009	0.59	0.12	1.12	−0.03	0.53	−0.05	0.42
540 s		−0.02	0.81	0.16	1.29	−0.04	0.8	−0.08	0.79
1080 s		−0.03	1.25	0.05	1.4	−0.06	1.38	−0.1	1.34
1620 s		−0.05	1.44	0.15	2.81	−0.1	3.28	−0.14	3.3
90 s	h2	−0.15	11.2	0.2	6.24	0.07	10.6	−0.14	3.06
180 s		−0.15	13.6	0.19	6.46	−0.01	3.23	−0.08	3.49
360 s		−0.17	14.5	0.22	6.52	−0.03	3.93	−0.11	3.77
540 s		−0.18	5.6	0.17	6.7	−0.01	4.23	−0.14	3.82
1080 s		−0.08	5.96	0.19	6.78	−0.07	4.53	−0.13	4.13
1620 s		−0.11	7.74	0.11	9.39	0.02	7.67	−0.23	4.67
90 s	luindex	−0.01	1.12	−0.006	1.15	−0.005	0.81	0.005	0.48
180 s		−0.01	1.26	−0.006	1.27	−0.002	0.85	0.009	0.51
360 s		−0.003	1.42	0.02	1.38	−0.009	1.27	−0.01	0.65
540 s		−0.01	1.54	0.0007	1.41	−0.01	1.27	−0.002	0.76
1080 s		−0.009	1.85	−0.002	1.44	−0.03	1.32	−0.03	1.16
1620 s		0.02	1.93	0.06	3.52	−0.04	2.41	−0.05	2.37
90 s	sunflow	−1.52	125.0	−0.14	17.6	−0.18	19.7	−0.08	10.3
180 s		−1.61	130.0	−0.14	18.9	0.15	22.1	−0.36	11.7
360 s		−1.56	134.0	−0.15	19.9	0.18	27.8	−0.14	8.09
540 s		−1.63	136.0	−0.14	20.2	−0.003	31.5	−0.28	8.77
1080 s		−1.68	146.0	0.007	21.6	−0.14	33.5	−0.33	9.32
1620 s		−1.73	162.0	−0.15	21.6	0.2	45.4	−0.25	9.65
90 s	xalan	−0.18	6.91	−0.13	12.5	−0.24	5.38	−0.02	11.3
180 s		−0.15	7.04	−0.14	13.6	−0.28	5.54	−0.18	6.53
360 s		−0.26	7.16	−0.13	14.1	−0.23	5.78	−0.16	6.67
540 s		−0.2	7.43	−0.12	14.8	−0.4	6.98	−0.24	6.87
1080 s		−0.11	7.9	−0.12	15.1	−0.009	8.61	−0.28	7.81
1620 s		−0.16	7.92	0.06	23.0	−0.06	8.61	−0.32	8.92

3% from zero), while exhibiting high stability (low mean squared errors).

In the case of the h2 and luindex benchmarks, the stability decreases with increased replay speed (shorter benchmarking time). However, we observe no such trend for most of the other benchmarks. Interestingly, the mean squared errors tend to be higher overall for certain benchmarking times, such as 540 s. This phenomenon may be related to undesired harmonic interference among Showstopper's processes and other system components and is yet to be investigated.

Overall, we can conclude that the experiments demonstrate a universally high load control accuracy, with some influence of the controller configurations depending on the controlled workload. As outlined in Section 3.2.6, Showstopper's CPU load feedback is more stable on systems with a higher number of CPUs. However, Showstopper works on a variety of systems, including small ones. Hardware configurations under test range from a uniprocessor laptop made in 2003 and a dated dual-CPU desktop (used for development and initial testing of Showstopper) to a 128-CPU Power7 server. Showstopper's modular nature and rich set of

configuration options can be leveraged to quickly assemble and fine-tune a load control mechanism suitable for a given situation and purpose.

5. Experiment: throughput reconstruction

After establishing the accuracy of our load control tool, we proceed to experiments that reconstruct the relationship between utilization and throughput, discussed in Section 2.2. In these experiments, we use the average throughput as the metric of interest, defined as the number of iterations completed by all running instances of a benchmark per second, averaged over the observation periods.

5.1. Platform configuration

To provide sufficient platform heterogeneity, we present throughput reconstruction results obtained on two different physical systems. The first system, PM1, is described in Section 4.1.

The second system, denoted PM2, is an IBM Power 750 Express server; we use a dedicated logical partition comprising 8 Power7 processor cores (32 logical processors) at 3 GHz and 64 GB of RAM, running IBM J9 VM (build 2.6, JRE 1.7.0) on AIX 6.1.

5.2. Reconstructing the dependency of throughput on load

Since the dependency of throughput on load is non-trivial and not necessarily monotonic, we infer it point-by-point by measuring the average application throughput and average observed load while enforcing a broad range of constant partial loads.

We experiment with the four Showstopper configurations from Section 4.2 and compare the results with baseline results obtained using the N-bench method and a variant of the CDC method, referred to as 1CDC. The 1CDC method only spawns a single benchmark instance, which provides a more appropriate baseline for the heavily multi-threaded benchmarks, such as sunflow and xalan. The N-bench method is used with low and mildly multi-threaded benchmarks, fop, avrora, h2, and luindex. For each benchmark, the baseline method capable of covering the widest range of partial loads is chosen: 1CDC combined with mildly multi-threaded benchmarks generally fails to induce high CPU loads, even at a 100% duty cycle, because it cannot surpass a maximum load dictated by the level of parallelism in the benchmarks. N-bench combined with heavily multi-threaded benchmarks fails to induce low CPU loads, because it always yields a 100% CPU load for any number of benchmarks.

The purpose of the baseline experiments is to verify that their load-throughput dependency estimate is aligned closely enough with the estimate obtained using Showstopper's feedback control. This confirms that the use of Showstopper's feedback control does not distort the dependency estimate in a significant way.

We use the same platform configuration as in Section 4.1. Measurements are performed during 3-min benchmark runs after 3-min warm-up periods to reach steady state execution.

5.2.1. Results for 12-processor Linux system (PM1)

The results for a 12-processor Intel-based system running Linux are shown in Fig. 8. The throughput observations for the four Showstopper configurations and for the 1CDC method correspond to target loads from 8.33%, with increments of 8.33%. The observations for the N-bench method correspond to runs with 1–12 benchmarks.

The relation between the observed average load and the average throughput is often non-linear and clearly application-dependent. For heavily multi-threaded benchmarks, such as xalan and sunflow, there is an almost linear dependency between load and throughput.

The fop, luindex, sunflow, and xalan benchmarks exhibit certain sensitivity to Showstopper configurations, with some of them deviating from the baseline. Interestingly, the controllers with uneven work distribution tend to provide results closer to the baseline. The PI controller with even work distribution generally performs the best with respect to load control; the choice of controller does not have much impact with respect to throughput estimation, as long as uneven work distribution is used. This can be explained by the similarity of the uneven work distribution to the N-bench baseline for low to mildly multi-threaded benchmarks. Additionally, compared to the even variant, which always interrupts the execution of all benchmarks, the uneven variant only interrupts the execution of one benchmark at a given time. The differences in context-switching overhead disappear with avrora, which has an exceptionally high rate of inherent context switching.

5.2.2. Results for 32-processor AIX system (PM2)

The results for a 32-processor Power7-based system running AIX are shown in Fig. 9. The throughput observations for the four Showstopper configurations and for the 1CDC method correspond to target loads from 8.33%, with increments of 8.33%. The observations for the N-bench method correspond to runs with 2–32 benchmarks, with increments of 2. In contrast to Linux, the AIX operating system provides two load metrics, *physical* and *logical*, both supported by Showstopper. We report results based on the *logical* load.

With the exception of the h2 benchmark, the relationship between load and throughput is similar to the Linux system (platform PM1). The h2 benchmark behaves very differently on the AIX platform, with the throughput observations collected using Showstopper significantly deviating from the N-bench baseline, regardless of the configuration. We are investigating this phenomenon; so far we only note that the behavior of throughput as a function of load can exhibit significant platform dependency.

The throughput observations collected with Showstopper configurations using the uneven work distribution tend to provide results closer to the baseline, compared to the even variant. This is in line with the observations on the 12-processor Linux system (platform PM1).

5.3. Reconstructing throughput for fast-forwarded load traces

Besides constructing the dependency between utilization and throughput using point-by-point measurements, it may be useful to reproduce an entire throughput trace in an experiment. Here, we go beyond examining the ability to reproduce a variable load trace, which was already shown in Section 4.4, and test whether it is possible to reproduce a faster version of a throughput trace to reduce the experiment time. Specifically, we compare the application throughput estimates for a trace replayed at original speed and for its fast-forwarded variants.

We first illustrate the key idea of trace-based throughput estimation on the variable-load trace shown in Fig. 7. While periodically observing the average throughput of a benchmark or an application, we replay the trace at original speed using Showstopper, producing a series of throughput observations. The results for avrora and fop, with Showstopper configured to use the PI controller with even work distribution, are shown in Fig. 10. The shape of the resulting throughput traces is determined by the benchmark-specific dependency of throughput on average load. For avrora and fop, these are shown in Fig. 8(a) and (b), respectively.

To estimate throughput corresponding to a load trace in a more efficient manner, we use a fast-forwarded version of the original

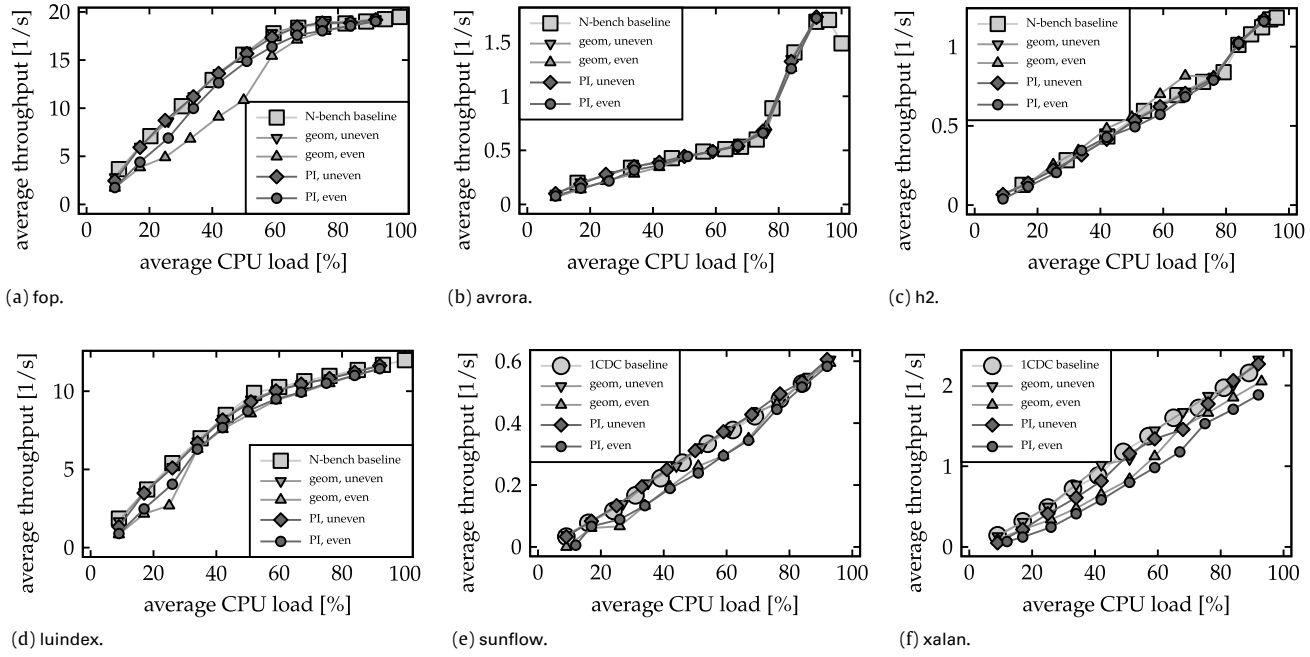


Fig. 8. Dependency of application throughput on partial load for 6 benchmarks on a 12-CPU Linux system (platform PM1).

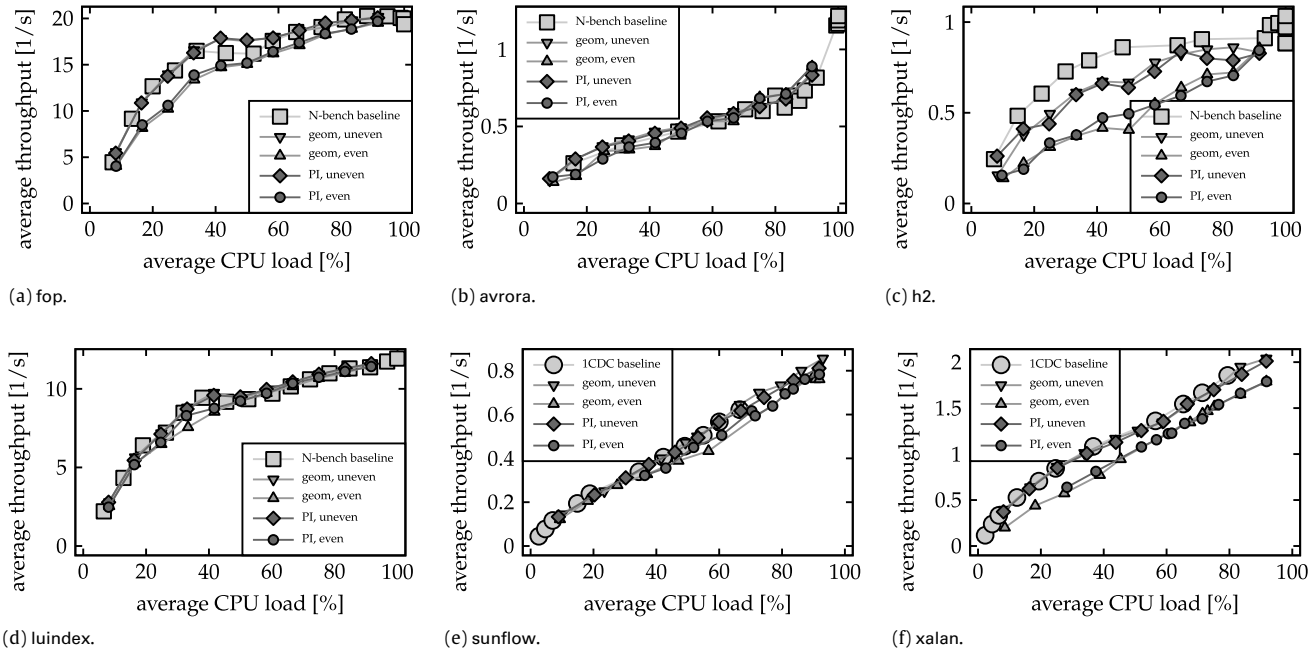


Fig. 9. Dependency of application throughput on partial load for 6 benchmarks on a 32-CPU AIX system (platform PM2).

trace to obtain results in a fraction of the original trace duration. To determine whether we can rely on such estimates, we evaluate different variants of the trace in Fig. 7, using the four Showstopper configurations on the two physical machine platforms. Ideally, the throughput estimates for all trace variants should be similar.

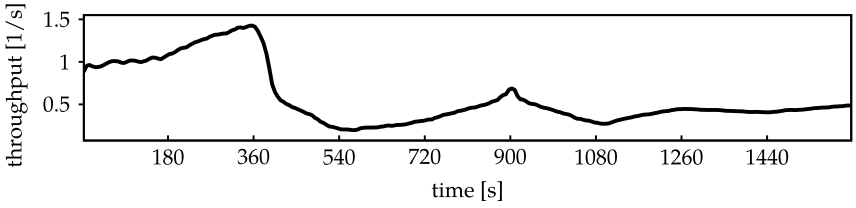
5.3.1. Results for 12-processor Linux system (PM1)

Results for the 12-processor Linux system are shown in the left part of Table 3. We observe that in most cases, the differences in throughput between the original trace and the fast-forwarded variants are negligible. When Showstopper is configured to use uneven work distribution, the throughput estimates for the fop benchmark increase by almost 20% as the benchmarking time decreases, regardless of the controller used.

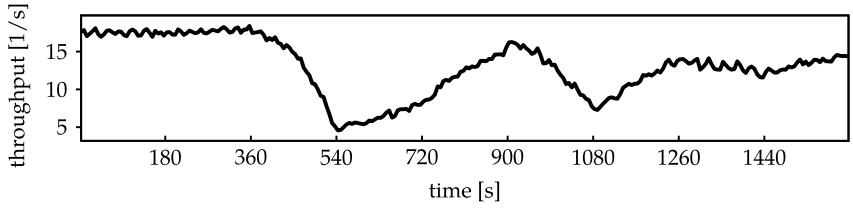
Nevertheless, in general, the results for the Linux system indicate that fast-forwarding load traces provides a viable approach to efficient estimation of application throughput corresponding to given load traces.

5.3.2. Results for 32-processor AIX system (PM2)

Results for the 32-processor AIX system are shown in the right part of Table 3. We observe that fop, h2, and avrora appear to be more sensitive to fast-forwarding, with fop being the most prominent case, regardless of the controller and work distribution. In contrast to results from the 12-processor Linux system (PM1), the throughput estimates for the benchmarks mostly decrease with decreasing benchmarking time.



(a) avrora throughput.



(b) fop throughput.

Fig. 10. Throughput estimates for avrora and fop corresponding to a variable-load trace from Fig. 7 replayed on a 12-processor Linux system (platform PM1) at original speed.

Although a more careful tuning of Showstopper's configuration may be needed on the AIX system, our approach remains viable and reasonably accurate for all benchmarking times except the shortest one.

6. Experiment: workload pinning

This set of experiments represents a major study of the impact of (1) CPU utilization, (2) workload type, and (3) CPU pinning on performance interference between colocated workloads. The complexity of the experimental setup reflects the number of factors (including different virtualization environments) varied during this experiment. For this reason, we start our description of the experimental setup with a base experiment that we develop into a complete experiment run that takes into account all the factors.

6.1. Experimental setup

The base experiment is designed to evaluate performance interference between a pair of colocated workloads. We execute each workload in a separate *virtualized environment* (VE) sharing the same physical machine, and measure the performance of the workload executing in one VE (the *foreground* workload) while varying the workload type and load intensity in the other VE (the *background* workload).

We use Showstopper to control the execution of the background workload so that it follows a predefined CPU utilization trace which performs a sweep over a range of partial CPU loads from 5% to 95%. The foreground workload is not controlled in any way, and we measure its performance (throughput) at peak CPU utilization. To analyze the impact of CPU pinning on the performance interference between the two colocated workloads, we repeat the base experiment with different CPU pinning configurations. Additionally, we perform the whole set of experiments using two different Linux-based VEs—a full-fledged virtual machine (KVM), and a resource isolation framework (LXC).

We focus solely on scenarios with no obvious competition for CPU resources (in terms of time, preemption and CPU overcommitment), so that the observed performance interference can be attributed only to the contention inherent to the hardware under test rather than to preemption[5] and time sharing. In particular, the total number of CPUs in the KVMs is equal to the total number of physical CPUs and the LXCs are pinned to disjoint sets of CPUs, the disjoint union of which comprises all CPUs of the physical server.[6]

6.1.1. Workload selection

We experiment with a broad variety of workloads targeting the JVM. We choose the JVM as an application platform for workloads under test because of its popularity in many areas, ranging from cell phones to cloud services. The set of workloads in our study is based on 10 different benchmarks—six from the well-known DaCapo suite [22], and four from the extended ScalaBench suite [33]. Even though workloads from both suites execute on the JVM, the Java and the Scala workloads exhibit different memory usage patterns, contributing to workload diversity. This choice of workloads results in 100 possible combinations of foreground and background workloads. For brevity, here we only present 25 possible combinations of 5 workloads (avrora, h2, luindex, scalac, specs), chosen to represent a diverse set of workload types. Benchmarks with short iteration times are preferred—because we estimate application throughput from iteration durations, shorter iterations yield more precise throughput estimates over short periods of time (units of minutes) and make changes in throughput easier to observe. For complete results, we refer the reader to extensive supplementary material,[7] which also provides the high-level power and performance metrics together with additional low-level metrics and hardware counter observations.

6.1.2. Platform configuration

As the physical machine, denoted PM3, we use a Dell M620 server with 128 GB of RAM and two Intel Xeon E5-2680 processors (8 cores each, 32 logical processors altogether) at 2.7/3.5 GHz, running the Fedora 20 Linux distribution on the physical machine, on the KVM virtual machines, and inside the LXC containers. The software environment includes the 3.14 Linux kernel series and the OpenJDK 1.7 JVM. As before, we enable all power and performance management technologies common in production environments.

[5] Time sharing is easy to predict, model and avoid, unlike other types of contention, exposed and explored by our study.

[6] We use the term "CPU" in a way common for the Linux kernel and performance monitoring tools, i.e., a CPU is an entity capable of executing a thread. The hardware platform used for our experiments has two CPUs/threads per *core*, eight cores residing on a *chip*, and two chips in total.

[7] http://d3s.mff.cuni.cz/resources/colloc2014.

Table 3
System-wide average throughput (iterations per second) on the Linux system (left) and on the AIX system (right). The left column displays the benchmarking time.

		Throughput: Linux system				Throughput: AIX system			
		geom, uneven	geom, even	PI, uneven	PI, even	geom, uneven	geom, even	PI, uneven	PI, even
1620 s	fop	131.0	120.0	130.0	130.0	172.0	169.0	173.0	170.0
1080 s		142.0	122.0	138.0	134.0	174.0	167.0	172.0	169.0
540 s		148.0	123.0	146.0	132.0	170.0	164.0	171.0	165.0
360 s		151.0	128.0	150.0	133.0	168.0	160.0	167.0	161.0
180 s		153.0	122.0	152.0	142.0	162.0	147.0	162.0	149.0
90 s		154.0	125.0	153.0	136.0	155.0	134.0	154.0	135.0
1620 s	avrora	5.93	5.51	5.91	5.64	4.95	4.71	4.9	4.78
1080 s		5.98	5.52	5.9	5.62	4.95	4.66	4.9	4.74
540 s		5.88	5.55	5.92	5.68	4.93	4.78	4.9	4.75
360 s		5.85	5.52	5.78	5.5	4.75	4.42	4.85	4.58
180 s		5.85	5.65	5.85	5.6	4.7	4.8	4.65	4.8
90 s		5.9	5.7	5.9	5.6	4.4	3.4	4.2	3.2
1620 s	h2	5.84	5.83	5.83	5.58	6.25	6.55	6.39	6.12
1080 s		5.89	5.93	5.77	5.54	6.97	6.45	6.43	6.02
540 s		5.68	6.08	5.72	5.5	6.58	5.65	6.28	6.07
360 s		5.78	6.08	5.72	5.48	5.72	6.02	7.38	5.62
180 s		5.8	6.0	5.55	5.45	6.7	5.2	6.4	4.95
90 s		5.5	5.9	5.7	5.4	5.9	3.7	6.0	4.2
1620 s	luindex	90.2	78.3	89.7	78.0	93.1	88.0	92.9	89.8
1080 s		90.1	81.9	90.1	78.1	92.5	87.6	92.3	89.5
540 s		89.8	77.9	90.3	77.5	91.6	86.4	92.6	88.3
360 s		89.6	76.5	89.8	77.2	91.2	86.5	91.4	87.9
180 s		89.8	77.4	89.8	77.5	91.0	86.0	91.2	86.8
90 s		89.4	76.9	90.0	84.8	90.5	85.2	90.7	84.9
1620 s	sunflow	3.45	2.82	3.38	2.84	5.49	5.14	4.87	5.94
1080 s		3.46	2.82	3.42	2.88	5.54	5.08	4.86	5.93
540 s		3.45	2.85	3.35	2.85	5.45	5.15	4.92	6.0
360 s		3.5	2.78	3.32	2.78	5.62	5.05	4.85	6.1
180 s		3.5	2.65	3.35	2.6	5.4	4.9	4.9	5.95
90 s		3.5	2.8	3.4	2.9	5.4	5.6	4.9	6.3
1620 s	xalan	13.0	10.1	12.4	10.6	12.7	11.2	12.9	13.6
1080 s		12.0	10.3	12.8	10.6	12.8	11.0	12.7	13.6
540 s		11.7	10.2	12.9	10.4	12.9	11.1	12.4	13.5
360 s		13.1	10.2	12.5	10.2	12.9	10.7	12.6	13.1
180 s		12.4	10.2	12.5	10.2	12.8	10.6	12.4	12.7
90 s		11.8	10.0	12.3	10.1	12.4	10.7	12.2	13.1

In particular, all load and throughput observations are performed with frequency scaling, CPU low-power states and TurboBoost turned on.

6.1.3. CPU pinning configurations

The colocation of two VEs opens a large space of possible CPU pinning configurations. We differentiate between four configurations, referred to as "neutral", "per-chip", "per-core", and "per-thread". These configurations are derived from the NUMA topology of our Dell M620 server, and are illustrated in Fig. 11(a)–(d). The numbers reflect the numbering of logical CPUs used by Linux. The background color shows which of the two VEs can execute on the particular CPU.

The "neutral" configuration (Fig. 11(a)) does not enforce an explicit pinning of virtual CPUs to physical CPUs, but each VE is allowed to use at most 16 CPUs at a time. The operating system kernel is free to schedule the virtual CPUs (processes in the host system) to arbitrary physical CPUs and migrate them as it sees fit. This configuration is available with KVM-based VEs only, because the number of CPUs available to an LXC-based VE can only be limited by explicitly pinning the VE to a set of CPUs.

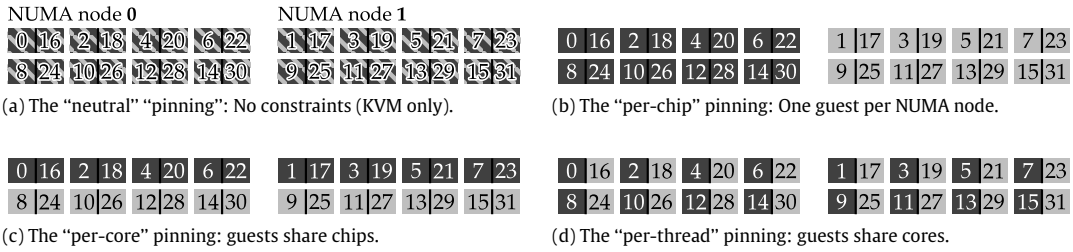
The three remaining pinning configurations can be used with both KVM and LXC. The "per-chip" configuration constrains each of the two VEs to a separate NUMA node (Fig. 11(b)). The "per-core" configuration forces the two VEs to share chips, i.e., each of them is pinned to four cores of each of the two chips (Fig. 11(c)). Finally, the "per-thread" configuration forces the two VEs to share cores and chips, i.e., each of them is pinned to one CPU (SMT thread) of each of the 16 cores (Fig. 11(d)).

6.1.4. Putting it all together: An experiment run

Each of the two colocated VEs runs a set of processes of the same type, i.e., JVMs running the same DaCapo or ScalaBench benchmark repeatedly. The number of processes is equal to the number of CPUs available to a VE, i.e., 16. The two VEs combined can therefore hog all CPUs of the physical system. To control the CPU load induced by the background workload, we throttle the background workload using Showstopper. Running in the same VE as the controlled workload, Showstopper modulates the workload's execution to achieve and sustain a predefined partial CPU load, regardless of the multi-threaded nature of the workloads and dynamic changes in their behavior. We measure the throughput of the foreground workload under 19 different background CPU loads, ranging from 0.05 to 0.95 with an increment of 0.05.

We illustrate the output obtained from our experiments on an example with specs in the foreground and luindex in the background. Fig. 12 shows the results for the two colocated KVM-based VEs under four CPU pinning configurations. Each plot represents an experiment run with the displayed pinning configuration. The broad "staircase" line corresponds to the target CPU load (left y-axis) in the background VE, enforced by Showstopper. The grey lines overlaying the target load trace show the observed background CPU load.[8] The black lines show the through-

[8] We always report the observed CPU load values rather than Showstopper's load target, although both values are expected to be in close correspondence. Load control may be prone to oscillations in certain configurations, especially for low target CPU loads and multi-threaded workloads; however, Showstopper handles most situations with reasonable stability and precision.



(a) The "neutral" "pinning": No constraints (KVM only).

(b) The "per-chip" pinning: One guest per NUMA node.

(c) The "per-core" pinning: guests share chips.

(d) The "per-thread" pinning: guests share cores.

Fig. 11. Pinning of LXC and KVM guest VEs on a dual-socket x86_64 server. Squares represent logical CPUs (as numbered by the Linux kernel). Vertical black lines separate CPUs sharing the same core. Thin gaps separate cores within the same chip. A thick gap in the middle separates the two chips. Background colors distinguish the two guest VEs.

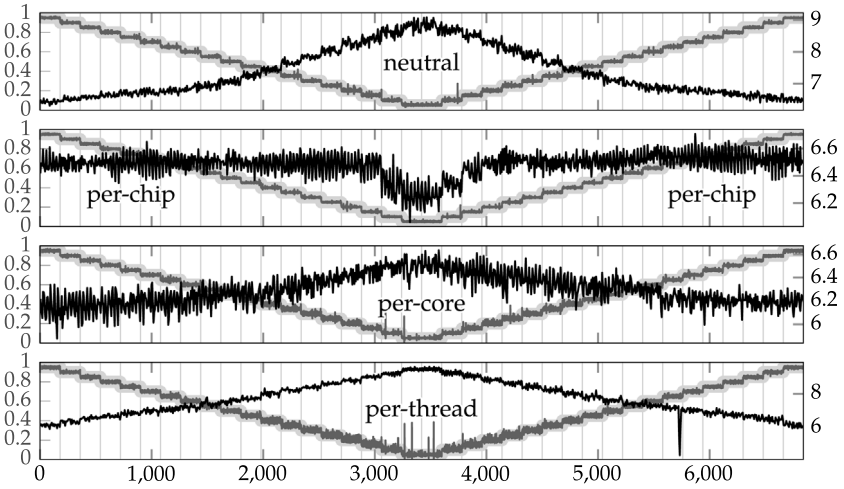


Fig. 12. [s] → [1], [s] → [s^{-1}] Time line of specs throughput (right vertical axis) depending on luindex load (left vertical axis) under "neutral", "per-chip", "per-core", "per-thread" pinnning of two colocated KVM VEs.

put of the foreground workload (right y-axis),[9] corresponding to the background CPU load represented by the grey lines. Because the "staircase" load trace is symmetric, averaging over its "steps" yields 38 [load, throughput] value pairs per experiment. The symmetry of the trace makes it possible to detect long-term performance changes of workloads or the execution environment. Under a steady performance, "steps" with the same background load yield the same foreground and background throughput.

Because some of our Java and Scala-based workloads require a long warm-up period (sometimes even 10 min) to reach steady state performance, we exploit the ability of Showstopper to follow predefined load traces to elide long warm-up periods preceding each partial load experiment. The "staircase" trace allows us to take all throughput and performance measurements under 19 background load levels for a particular workload pair in a single experiment run, preceded by only one warm-up period. This approach significantly reduces the duration of our experiments.

For the large volume of experimental results, we choose a much more condensed form of presentation in the following sections, showing only the average CPU load and average throughput values for each "step" of the "staircase" trace. The duration of a step is 3 min. The first and last 30 s of each step are excluded from the average load and throughput computations to ensure that various transient fluctuations around the step boundary are not included in the average values.

The results and conclusions in this section stem from a set of more than 1000 long-running experiments based around the "staircase" artificial load trace followed by Showstopper. We first investigate the dependency of foreground and background throughput on background load in Section 6.2. We define a *performance interference metric* to describe the interference between colocated VEs in Section 6.3. Section 6.4 analyzes the overall system throughput. Section 6.5 concludes with energy efficiency.

6.2. Background load and foreground throughput

In a scenario without explicit competition for CPU time, we first explore whether (and to what extent) the varying CPU load in the background VE impacts application throughput in the foreground VE where a workload runs uncontrolled, using all the assigned CPU resources.

Fig. 13 shows how foreground throughput depends on background CPU load in two colocated KVMs for all 25 possible pairs formed by the 5 selected workloads. Each row of plots shows how a particular foreground workload interacts with different background workloads represented by columns. Each plot displays the results of four experiments, carried out under the "neutral", "per-chip", "per-core" and "per-thread" CPU pinnings. Two data sets are plotted for each experiment, corresponding to the two symmetric parts of the "staircase" trace (triangles pointing downwards (⟶⟵) and upwards (⟵⟶), respectively). In an ideal case, the two data sets should overlap. Certain flawed benchmarks are prone to performance degradation over time, especially fop—its performance steadily decreases during all experiments.[10] The symmetry of the "staircase" trace helps us reveal such issues.

[9] For Fig. 12, we compute throughput estimates for adjacent intervals of 6 s as follows: All per-interval estimates are initialized to zero. To each estimate, we add 1 for each workload iteration that starts and also ends within the interval. For an iteration spanning multiple intervals, we add values that sum to 1 to the intervals' throughput estimates, according to the fractions of the iteration overlapping the intervals.

[10] Shown in our supplementary plots.

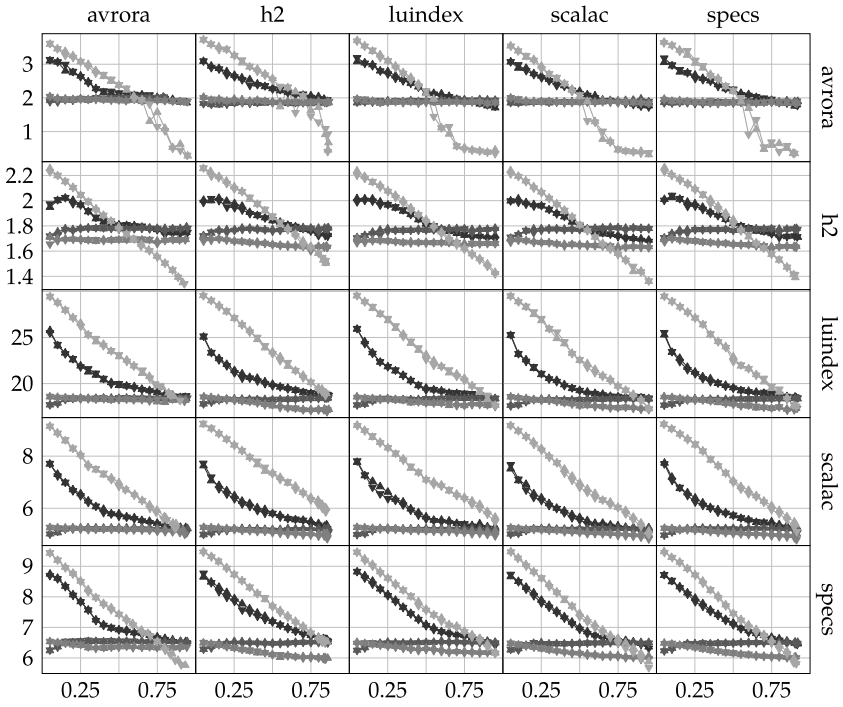


Fig. 13. [1] → [s^{-1}] KVM: Throughput of the foreground (uncontrolled) workload depending on the CPU load induced by the background (controlled) workload; "neutral" (—▾— —▴—), "per-chip" (—▾— —▴—), "per-core" (—▾— —▴—) and "per-thread" (—▾— —▴—) CPU pinning.

The key lesson learned from Fig. 13 stems from the ordering of the CPU pinning configurations in terms of performance. The "per-thread" pinning outperforms other configurations in a majority of cases; only avrora and h2 are sometimes an exception, especially under high background loads. The "neutral" (unpinned) configuration mostly yields the second best performance, showing that the default kernel scheduler uses CPUs efficiently, even with no precise knowledge about the workload colocation. The "per-chip" and "per-core" pinning configurations provide a stronger level of resource isolation (maintaining a steady throughput in face of changing background load), yet exhibit the lowest throughput altogether. The results for LXC are similar in most cases; only the behavior of avrora differs (see footnote 10).

According to low-level hardware performance counters, the "per-thread" pinning has certain unique characteristics. With decreasing background load, the absolute number of cache misses mostly increases for all three levels of caches. However, the cache miss rate mostly decreases for the L1 cache, remains steady for the L2 cache and increases for the L3 cache. Moreover, the number of pipeline stalls decreases with the decreasing background load. This confirms that the "per-thread" pinning enables the foreground VE to benefit from additional pipeline and cache resources (e.g., twice the L3 cache capacity compared to the "per-chip" pinning) whenever the background VE is less active. Therefore the "per-thread" pinnning yields a performance advantage, despite the increased NUMA inter-node and L3 cache traffic that can be attributed to the spreading of the foreground VE across both NUMA nodes and all cores. The "per-core" configuration exhibits weaker performance than "neutral" and "per-thread", yet a much smaller impact of background load on foreground throughput. The "per-chip" configuration mostly yields a very small negative (somewhat counter-intuitive) dependency of background load on foreground throughput and provides the strongest level of resource isolation overall.

We also test the impact of Showstopper's configuration (e.g., time quanta used in its duty cycle enforcement strategy) on our results. The data presented in this article are gathered using a configuration that sets the runnable and stopped periods of the controlled processes to fractions of a 107 ms time quantum, i.e., a duty cycle of 50% corresponds to alternating of 53.5 ms long stopped and runnable periods. When repeating our experiments with different Showstopper configurations, we still obtain almost identical results. For example, results of the entire set of 25 workload combinations repeated with a base quantum of 53 ms (instead of 107 ms) can be inspected in our supplementary plots.

Motivated by Fig. 13, we ask two additional questions: (1) Does the improved foreground throughput in certain pinning configurations come with a performance penalty in the background (a lower throughput at the same CPU load)? (2) How does background throughput depend on background CPU load? Fig. 14 answers both questions. Rows represent background workloads under colocation with various uncontrolled foreground workloads represented by columns.[11] Fig. 14 confirms earlier observations that the dependency of throughput on CPU load (in the same VE) is non-linear in most cases [10]. Obviously, the "per-thread" configuration comes with a performance penalty for the background VE, while providing a significant performance improvement to the foreground VE. The performance penalty is relatively strong for background loads around 0.5, yet almost negligible for both high and low background loads.[12] It appears to be weaker for LXC than for KVM, stronger for luindex and the Scala benchmarks and almost negligible for h2 and avrora. The latter correlates with the fact that "per-thread" is not always the foreground performance winner for avrora and h2, as shown in Fig. 13.

In Section 6.4, we investigate whether the foreground performance improvement related especially to the "per-thread" pinning can outweigh the background performance penalty, i.e., whether unusual pinning configurations can indeed provide a noticeable overall performance advantage over more common choices.

11 The data stems from the same set of experiments as in Fig. 13.

12 The LXC counterpart of Fig. 14 in supplementary plots shows that LXC exhibits a behavior similar to KVM.

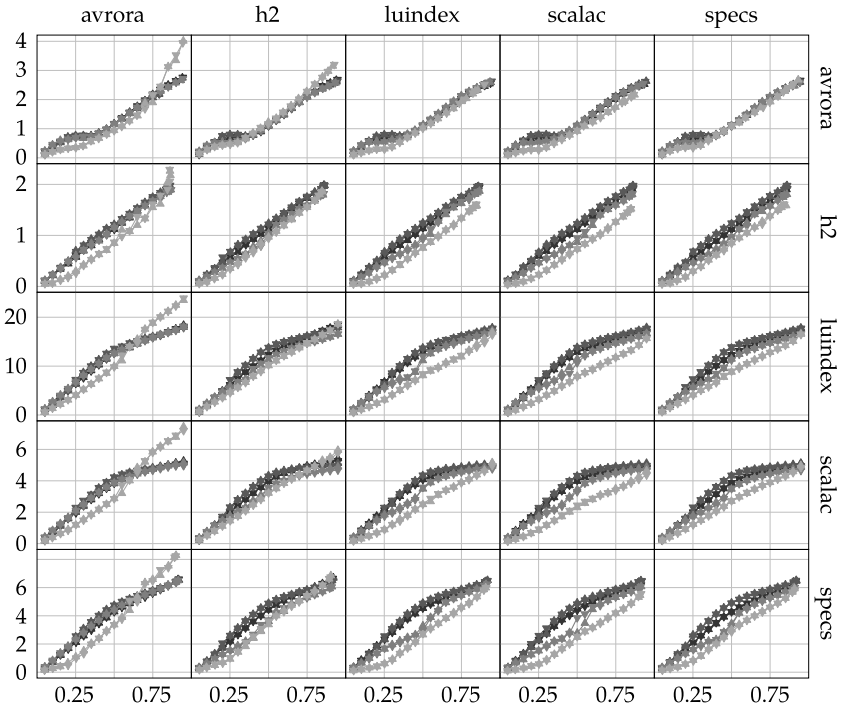


Fig. 14. [1] → [s^{-1}] KVM: Throughput of the background (controlled) workload depending on the CPU load induced by that workload; "neutral" (—▾— —▴—), "per-chip" (—▾— —▴—), "per-core" (—▾— —▴—) and "per-thread" (—▾— —▴—) CPU pinning.

6.3. The performance interference metric

To (crudely) represent the performance interference between the background CPU load and foreground application throughput by a single numeric value, we introduce a "performance interference metric" based on correlation. The metric not only provides a clear summary of the plots in Fig. 13, but also allows us to present more results in a compact form.

Our metric is based on correlation between CPU load in the background VE and throughput in the foreground VE. For each set of 38 average load and throughput pairs (one pair per "staircase" trace "step") obtained from an experiment, we derive two linear models from the two symmetric halves of the "staircase" trace, producing two pairs of *slope* and *intercept* values. Our performance interference metric is defined as $-100\frac{\text{slope}}{\text{intercept}}$. We thus obtain two metric values per experiment and present their arithmetic average, also reporting on their similarity and statistical significance.

The formula $-100\frac{\text{slope}}{\text{intercept}}$ is based on the following intuition: The intercept value represents a hypothetical maximum foreground throughput with an idle neighboring VE. We normalize the slope by the corresponding intercept, obtaining a "normalized slope" comparable among different benchmarks with different iteration durations. Finally, we multiply by -100 to obtain a "slowdown factor" resembling a percentage, with values up to 100.[13] Negative values of our metric represent a (counter-intuitive) foreground speed-up with increasing background load.

Admittedly, the dependency of foreground throughput on background load is in general non-linear, especially for most colocations with avrora in the foreground and also for the "neutral" pinning configuration. However, we use the slope and intercept values merely for the sake of our metric rather than in their rigorous statistical sense. Our metric helps us, among others, to illustrate the negative dependency of foreground throughput on background load under the "per-chip" pinning, as well as other phenomena that may not be immediately obvious from the plots. We are aware of cases where the linear model does not fit with a strong enough statistical significance and we report them—in a certain sense, the cases where the linear model does not fit confirm the fundamental motivation behind our experiments, showing that there is often no simple (linear) model describing partial load scenarios.

Table 4(a), (b), (c) and (d) show our performance interference metric for KVM experiments under the four pinning configurations, as presented in Fig. 13. Table 5(a), (b) and (c) correspond to the LXC experiments under the three pinning configurations available with LXC.

To facilitate the comparison of results and identification of patterns among them, we set the background color of the table cells based on their content (the interference metric value). Lighter background represents a lower value of the performance interference metric in the particular table cell, i.e., the darker the cell, the stronger the negative impact of background load on foreground throughput. Values reported for avrora are excluded from the color ranges (in all tables) to provide a broader color range for the less extreme values. The background colors are bound together in such a way that all the four tables with KVM results (4(a), (b), (c), (d)) share a common color space and the same holds for the three tables of LXC results (5(a), (b), (c)). This makes all the pinning configurations easily comparable in terms of performance interference: Tables with the "per-core" and "per-chip" configuration contain values close to zero, even mostly negative in the "per-chip" case. The "neutral" and "per-thread" configurations, by contrast, yield high values of the performance interference metric (corresponding to dark cell background).

As already mentioned, we report on the statistical significance of our metric, focusing on two criteria, namely (1) whether the two metric values obtained from the two symmetric stages of an experiment are close enough, i.e., whether the angle between the two normalized slope estimates differs by less than 3°, and

[13] We may (rarely) obtain values over 100 when the linear model overestimates the slope. This phenomenon sometimes coincides with a strong sensitivity of avrora to background load on KVM, contrasting with avrora on LXC.

Table 4
Performance interference metric on KVM.

	avrora	h2	luindex	scalac	specs
avrora	**−6.0**	**−3.0**	**−3.7**	−3.1	**−2.5**
h2	**−1.9**	**−2.2**	**−2.9**	**−2.9**	**−2.5**
luindex	**−2.4**	**−2.5**	**−2.9**	**−3.0**	**−2.9**
scalac	−0.9	−1.9	−1.6	−1.8	−1.9
specs	−1.9	**−2.3**	**−2.4**	**−2.5**	**−2.9**

(a) "per-chip" CPU pinning

	avrora	h2	luindex	scalac	specs
avrora	**5.7**	**6.5**	**3.4**	**4.7**	**4.2**
h2	0.2	**5.3**	**1.9**	**3.9**	**4.4**
luindex	**2.2**	**10.2**	**5.6**	**8.1**	**8.7**
scalac	**3.0**	**7.4**	**4.5**	**6.6**	**6.9**
specs	**2.6**	**9.8**	**5.9**	**8.2**	**8.8**

(b) "per-core" CPU pinning

	avrora	h2	luindex	scalac	specs
avrora	**91.2**	**86.2**	**106.3**	**105.9**	**105.6**
h2	**44.0**	**40.5**	**39.0**	**42.3**	**43.4**
luindex	**44.2**	**45.1**	**43.6**	**45.8**	**47.1**
scalac	**49.1**	**43.6**	**42.3**	**48.4**	**49.1**
specs	**43.0**	**39.1**	**37.5**	**42.4**	**44.1**

(c) "per-thread" CPU pinning

	avrora	h2	luindex	scalac	specs
avrora	**44.9**	**44.3**	**50.8**	**49.3**	**48.6**
h2	**16.2**	**18.4**	**19.6**	**20.0**	**19.7**
luindex	**28.4**	**26.3**	**29.6**	**26.3**	**26.2**
scalac	**34.3**	**33.6**	**34.7**	**33.6**	**33.0**
specs	**28.7**	**28.9**	**30.7**	**30.4**	**29.9**

(d) "neutral" CPU pinning

Table 5
Performance interference metric on LXC.

	avrora	h2	luindex	scalac	specs
avrora	−2.5	**−4.8**	**−6.9**	**−3.5**	**−6.1**
h2	−1.7	**−3.3**	−2.3	**−2.9**	−2.5
luindex	**−1.4**	**−3.2**	**−2.3**	**−2.4**	**−3.3**
scalac	−1.2	−2.2	−1.9	−2.5	−3.1
specs	−1.8	**−3.6**	**−3.0**	**−3.0**	**−3.8**

(a) "per-chip" CPU pinning

	avrora	h2	luindex	scalac	specs
avrora	−3.1	**−4.9**	0.2	0.3	−1.0
h2	**1.9**	**5.4**	**3.7**	**5.4**	**6.0**
luindex	−0.5	**5.5**	**2.0**	**4.3**	**3.7**
scalac	−0.6	**2.3**	**0.4**	**1.9**	**2.1**
specs	**−2.8**	**2.7**	0.2	**2.7**	**3.1**

(b) "per-core" CPU pinning

	avrora	h2	luindex	scalac	specs
avrora	12.5	−16.2	−2.1	3.2	1.8
h2	**41.5**	**41.5**	**42.9**	**44.4**	**45.3**
luindex	**40.4**	**41.5**	**40.4**	**41.7**	**43.1**
scalac	**45.6**	**40.9**	**40.4**	**44.2**	**45.5**
specs	**37.5**	**35.0**	**33.3**	**38.1**	**39.0**

(c) "per-thread" CPU pinning

(2) whether the linear model is statistically significant, as reported by the R tool, i.e., whether both slope and intercept values for both halves of the "staircase" trace are estimated with a p-value below 0.05. When both criteria fail, the corresponding number is typeset in *italics*. Failures of only the first (symmetry) or the second (p-value) criterion are represented by sans-serif and standard font, respectively.[14] Results for which both criteria are met are typeset in **bold**.

6.4. Throughput of the entire system

We show in Section 6.2 that certain CPU pinnings can improve the foreground throughput for a majority of partial CPU loads. However, we also reveal (and show in Fig. 14) that the foreground throughput improvement comes at the cost of a background throughput degradation. This leads to an obvious question whether the former can "outweigh" the latter, i.e., which pinning configuration is actually more performance-efficient overall.

We inspect the "overall system throughput" only in configurations where the foreground and background workload are of the same type and the system-wide throughput can therefore be simply summed. Inspecting heterogeneous workload combinations would require an (inherently inaccurate) throughput normalization.

For the five "symmetric" workload combinations, we first show the total amount of work accomplished by the entire system on each "step" of our "staircase" trace. This time line is shown in Fig. 15, separately for KVM and LXC, but with a common vertical axis range and scale for an easy performance comparison between KVM and LXC. We can see that the "per-thread" pinning often exhibits a performance advantage under low background loads. Under high background loads, the "per-chip" or "neutral" configurations sometimes yield a better performance. Furthermore, the throughput with LXC is higher than with KVM in all cases but avrora.

Fig. 16 shows the overall throughput data normalized against the commonly used "per-chip" pinning configuration (so that "per-chip" corresponds to 100%). Clearly the relative performance of the "per-core" pinning configuration varies with background load, but mostly stays close to the "per-chip" performance. On the other hand, the "per-thread" and "neutral" configurations tend to outperform the "per-chip" configuration with background loads below a certain threshold (around 50%), performing slightly worse once the background load surpasses the threshold. As usual, avrora exhibits strong irregularities that cannot be easily interpolated from measurements taken with idle and fully loaded neighboring VEs.

In summary, pinning "per-chip" when the background VE is loaded and "per thread" otherwise can potentially improve overall performance in this particular situation. We observe that the default Linux kernel scheduler (the "neutral" configuration) exhibits decent performance across a broad range of situations, yet performs worse than the ideal combination of "per-chip" and "per-thread". The "per-core" pinning yields inferior performance in most cases, with a rare exception of avrora on LXC.

6.5. Energy efficiency

Having explored the performance implications of CPU pinning, we now focus on its power implications, i.e., we investigate whether a CPU pinning that exhibits better performance requires more power and whether CPU pinnings differ from the energy efficiency standpoint.

[14] Failures in both criteria occur in the supplementary plots. Failures in the symmetry criterion are mainly due to flaws in benchmarks, e.g., in fop.

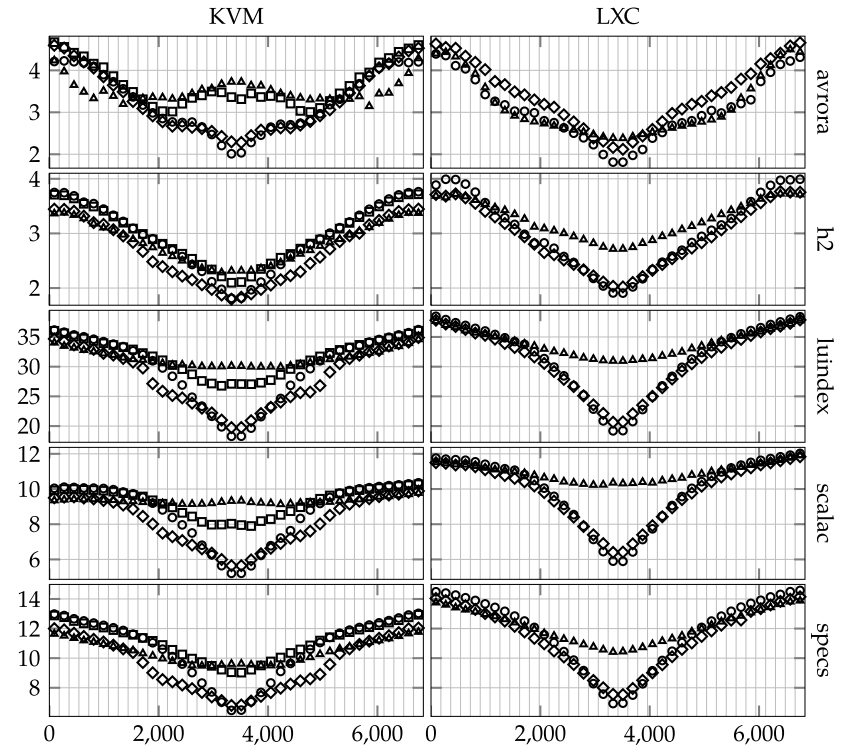


Fig. 15. [s] → [s^{-1}] Time line of absolute system throughput along the "staircase" load trace for 5 symmetric workload colocations under the "neutral" (□), "per-chip" (○), "per-core" (◇) and "per-thread" (△) pinning.

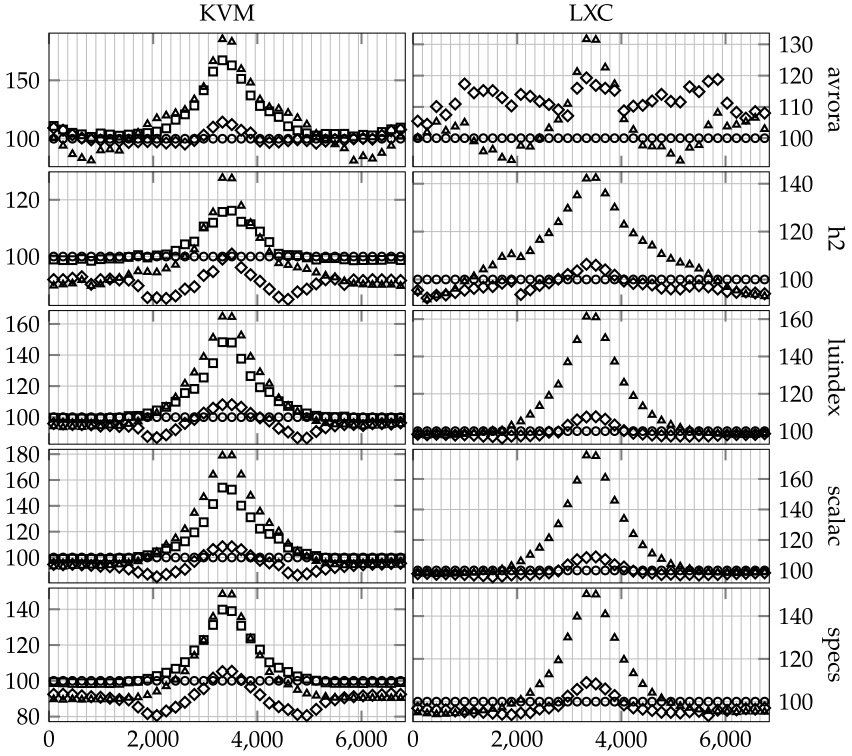


Fig. 16. [s] → [%] Time line of relative system throughput along the "staircase" load trace. Data from Fig. 15 are normalized against the "per-chip" pinning (○) for each "staircase" step. Throughput values for the other pinnings ("neutral" (□), "per-core" (◇) and "per-thread" (△)) are shown in percentages of the "per-chip" value (○). Notice that the vertical axis range and scale is **not** equalized in this case. LXC throughput values follow the axis labels on the right.

We sample the power input of our server (in watts) once per second and for our long-term power consumption computations, we simply consider each sample to be a value in Ws (watt-seconds). Fig. 17 shows a time line of instantaneous power readings and average values computed for each step of our "staircase" trace, illustrating the variance of the power readings. For the

sake of brevity, we only present these values for two symmetric colocations that induced the lowest and highest variance in power consumption over time (luindex and specs).

Fig. 17 aligns the pinning configurations (4 for KVM and 3 for LXC) side-by-side with the same range and scale of the vertical axis, so that power consumption under different pinning configurations and VE types can be compared. The power consumption time lines are very similar in both shape and average values for other workload colocations, both "symmetric" and "asymmetric". Surprisingly, LXC exhibits a slightly higher average power consumption in most cases.

We compare the energy efficiency of KVM and LXC pinning configurations in terms of the number of workload iterations accomplished per watt-hour. Fig. 18 shows the energy efficiency and augments the throughput data in Fig. 15 by taking power consumption into account. Fig. 19, a power-aware counterpart of Fig. 16, shows the relative energy efficiency, normalized in such a way that "per-chip" corresponds to 100%.

We conclude that in colocation scenarios with partial CPU utilization, a carefully chosen CPU pinning configuration can increase the overall system performance and also enhance its energy efficiency. With the setup studied in this article, both improvements can be achieved at the same time by dynamically choosing between the "per-thread" or "per-chip" pinning according to the CPU load in the background VE. The Showstopper methodology provides the necessary means to explore a variety of partial CPU load scenarios and to tailor dynamic CPU pinning to a particular workload and platform.

7. Experiment: virtualization overhead

In the concluding experimental section, we use the fact that Showstopper supports both virtual and physical machines to examine the impact of virtualization on benchmark throughput at different partial loads. Specifically, we compare the virtualization overhead associated with the flexibility of KVM to the bare-metal mode of operation represented by the LXC resource containers.

Virtualization overhead strongly depends on both CPU pinning and workload combination. To illustrate this dependency in detail, we abandon the realm of "symmetric" workload combinations and focus instead on the ratios between the KVM and LXC foreground throughput while running the same workload combination under the same pinning configuration. Obviously, this limits our comparison to the three pinning configurations common to both KVM and LXC.

Because the performance ratios may also vary strongly with changing background load, we compute the ratio between the foreground application performance for each "staircase" step separately and present the minimum, median and maximum of such values, yielding three values for each workload combination. Table 6(a), (b) and (c) show these results. This approach gives a concise overview of how the virtualization overhead depends on workload combination and, perhaps more importantly, to what extent the overhead can vary with a varying background load. Table 6(a), (b) and (c) follow the "heat map" form already used in Section 6.3, which facilitates the comparison of values and identification of patterns in the data. The lighter the cell background, the stronger the slowdown induced by KVM in comparison to LXC. Surprisingly, we can also observe speedups in case of the avrora benchmark—avrora may sometimes perform better under virtualization (KVM) than on bare metal (LXC), especially for the "per-chip" pinning.

It appears that h2, scalac and specs are more affected by virtualization overhead than other benchmarks, especially under the "per-chip" and "per-core" pinnings. The "per-thread" pinning, on the other hand, makes the choice of the background workload

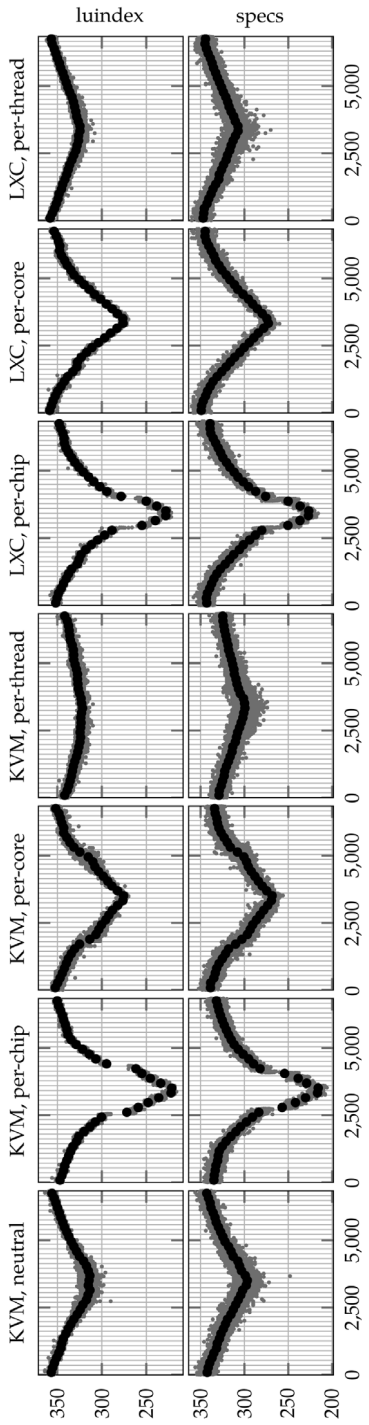


Fig. 17. [s] → [W] Time line of system power consumption along the "staircase" trace for two symmetric workload colocations (luindex/luindex and specs/specs) combined with the KVM and LXC pinning configurations. Displayed are power readings taken every second (·) and averages over each "staircase" trace step (•).

much more influential in terms of virtualization overhead—horizontal patterns in the heat map are far less pronounced in the "per-thread" case. The behavior of avrora is irregular with (mostly) speed-ups under the "per-chip" pinnning, a strong slowdown under the "per-core" pinning and a mixture of both extremes under the "per-thread" pinning.

8. Discussing related work

As a load control tool, Showstopper is directly related to workload generators employed in benchmarks and related frameworks

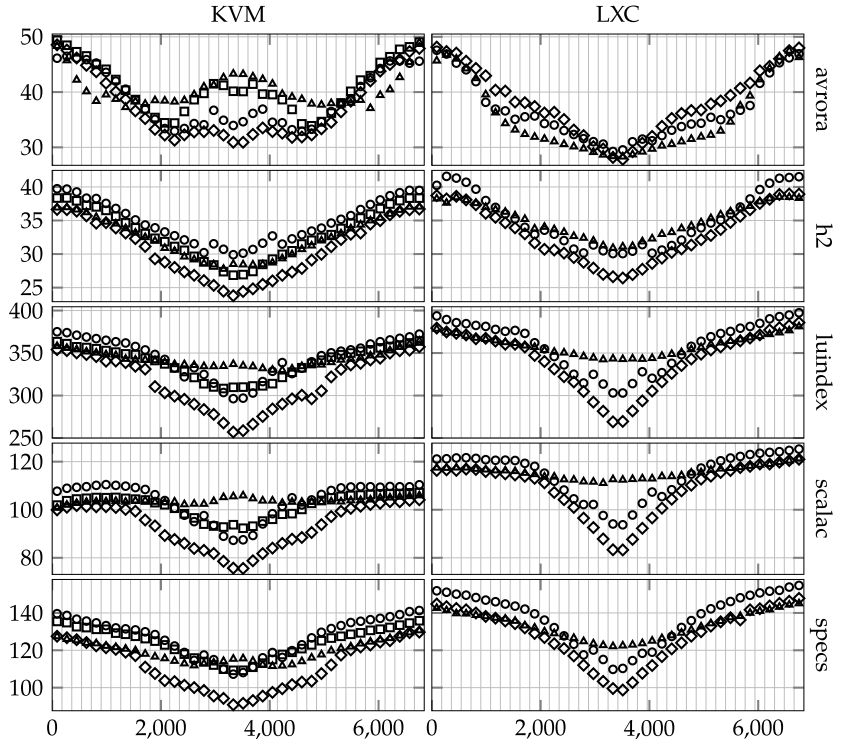


Fig. 18. [s] → [(Wh)$^{-1}$] Time line of absolute energy efficiency along the "staircase" load trace for 5 symmetric workload colocations combined with the "neutral" (□), "per-chip" (○), "per-core" (◇) and "per-thread" (△) pinning.

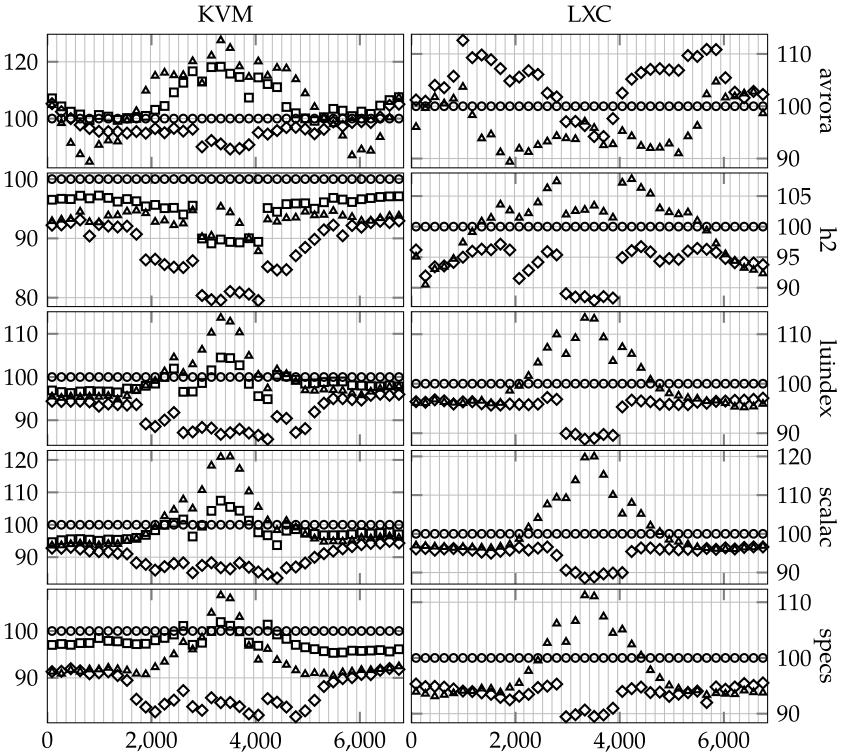


Fig. 19. [s] → [%] Time line of relative energy efficiency along the "staircase" load trace. Data from Fig. 18 are normalized against the "per-chip" pinning (○) for each "staircase" step. Energy efficiency values for the other pinnings ("neutral" (□), "per-core" (◇) and "per-thread" (△)) are shown in percentages of the "per-chip" value (○). Notice that the vertical axis range and scale is **not** equalized in this case. LXC energy efficiency values follow the axis labels on the right.

in performance evaluation experiments. Unlike common workload generators, Showstopper requires other software to implement the workload, which it then controls to achieve particular partial load. Showstopper can therefore complement many workload generators [34] or benchmarks [22,35] that do not support partial loads themselves. From those that do support load control, Showstopper

Table 6
KVM throughput as a fraction of LXC throughput (minimum, median, maximum).

	avrora	h2	luindex	scalac	specs
avrora	1.060	1.016	1.048	1.074	1.031
	1.124	1.088	1.115	1.120	1.106
	1.204	1.133	1.179	1.166	1.141
h2	0.906	0.904	0.899	0.907	0.899
	0.917	0.922	0.915	0.923	0.917
	0.938	0.944	0.945	0.948	0.948
luindex	0.935	0.944	0.944	0.938	0.942
	0.950	0.951	0.952	0.947	0.948
	0.959	0.961	0.962	0.954	0.961
scalac	0.850	0.858	0.851	0.861	0.858
	0.860	0.868	0.862	0.869	0.870
	0.868	0.874	0.866	0.884	0.879
specs	0.890	0.892	0.893	0.891	0.895
	0.900	0.899	0.899	0.895	0.899
	0.909	0.912	0.912	0.908	0.918

(a) "per-chip" CPU pinning

	avrora	h2	luindex	scalac	specs
avrora	0.886	0.875	0.864	0.864	0.859
	0.937	0.912	0.899	0.905	0.899
	1.037	1.025	0.965	0.993	0.983
h2	0.859	0.866	0.868	0.872	0.870
	0.886	0.887	0.890	0.894	0.894
	0.934	0.913	0.923	0.933	0.943
luindex	0.923	0.920	0.921	0.926	0.920
	0.942	0.942	0.937	0.943	0.941
	0.957	0.962	0.957	0.963	0.965
scalac	0.836	0.839	0.835	0.831	0.829
	0.853	0.865	0.848	0.850	0.846
	0.877	0.888	0.879	0.879	0.876
specs	0.856	0.850	0.853	0.850	0.850
	0.865	0.866	0.870	0.863	0.865
	0.906	0.903	0.906	0.899	0.902

(b) "per-core" CPU pinning

	avrora	h2	luindex	scalac	specs
	0.146	0.172	0.185	0.176	0.178
avrora	1.355	1.059	0.849	0.824	0.895
	1.589	1.619	1.609	1.550	1.593
	0.828	0.847	0.836	0.848	0.843
h2	0.854	0.873	0.860	0.867	0.869
	0.877	0.908	0.914	0.909	0.900
	0.935	0.920	0.929	0.928	0.916
luindex	0.964	0.970	0.967	0.959	0.979
	0.988	0.996	0.997	1.007	1.024
	0.859	0.856	0.876	0.854	0.829
scalac	0.884	0.900	0.901	0.887	0.896
	0.917	0.917	0.924	0.927	0.937
	0.846	0.863	0.864	0.870	0.844
specs	0.886	0.894	0.885	0.898	0.893
	0.927	0.923	0.921	0.931	0.934

(c) "per-thread" CPU pinning

differs both in the ability to run arbitrary benchmarks and in the load control accuracy, as illustrated in Section 4. Of the existing load generators, some can only control a limited number of benchmarks [31,36], while others do not use feedback and therefore cannot support arbitrary workloads [21]. Similar differences also distinguish Showstopper from the load control support present in the process scheduling modules of some operating system kernels.

When it comes to a dynamically-changing partial load, to the best of our knowledge there are no tools that can reproduce a load trace using arbitrary workloads and at the same time provide similar flexibility in the configuration of the feedback-control algorithm. The partial-load generator closest to Showstopper is KRASH [37], which can generate a controlled system load and replay load traces. However, KRASH provides only a subset of Showstopper's flexible functionality, for example there are no counterparts to Showstopper's space and dithering modules.

The importance of analyzing systems in partial-load conditions is illustrated by works of other researchers—Sueur et al. [38] used the Siege tool (a HTTP server stress tester) and a modified version of SPECjbb2005 (an EJB benchmark) to analyze the CPU power consumption under partial loads on multiple CPU types. Load generators and load control mechanisms in [39] mimic a production environment of telecommunication servers not only in terms of CPU load, but also in terms of cache misses. Partial load conditions and the associated utilization traces have been widely used as input to a variety of resource management policies aiming at power consumption optimization, efficient virtual machine provisioning, and workload placement.

Various energy management studies use utilization to estimate [15] and control [40–43] power consumption and cooling demands using strategies like dynamic frequency scaling and powering down of unnecessary servers. When consolidating virtual machines in a data center environment, resource utilization is a crucial parameter for virtual machine placement policies [14,44]. Resource provisioning policies [45,46] are validated using a request workload generated directly from utilization traces collected in production systems.

Pertinent to our methodology is the question of whether an accurate reproduction of load translates into other performance

metrics. Here, particularly encouraging are the results from the power management domain, which indicate CPU utilization is a very accurate predictor of power consumption [47]. Obviously, for applications whose performance is bound by resource contention outside CPU, reproducing CPU utilization is not necessarily sufficient—in such cases, existing studies [48,49] can help estimate the extent of resource sharing influence. We believe such influence needs to be evaluated on a case-by-case basis in any experiment with Showstopper.

The deeper question of relationship between resource utilization and application performance metrics is studied extensively in performance evaluation research. Model-driven approaches, such as queueing models, can accurately capture the application and system performance of modern multi-core systems hosting multiple applications [24,25] by means of detailed workload profiling. Among other works, Pacifici et al. [50] estimate CPU resources utilization for web servers using linear regression models. Wood et al. [26] have developed a regression-based model to capture the differences in resource demand of an application on native and virtualized hardware. Here, we believe our background throughput observations in Section 6 also contribute important insight in the complex relationship between utilization and throughput, which is often simplified in performance modeling studies.

The influence of workload colocation on both energy efficiency and system performance involves many interacting aspects, which are difficult to capture in a holistic manner—the research advances therefore often tackle selected aspects individually, some of which we discuss next.

The basic tension between energy efficiency and system performance motivates workload consolidation methods that guarantee preserving the same quality of service between isolated and colocated deployments. Song et al. [30] present an analytical queueing model that estimates the number of servers required to execute consolidated workload while preserving the same request drop rate as isolated servers would have. In the same vein, Verma et al. [29] observe that colocating workloads with coinciding peaks increases the risk of performance anomalies and develop an algorithm that strives to avoid such situations in consolidation.

Workload consolidation presents the opportunity to cast the relationship between efficiency and performance as a profit optimization problem. Chase et al. [1] provide one of the early examples, pitting the revenue expressed in terms of customer utility functions against the cost of resources. Among more recent research results, Ishakian et al. [51] show that expressing application resource requirements in terms of flexible preferences rather than fixed demands can reduce consolidated resource waste by up to 60%. Another economically motivated research direction is represented by Wang et al. [52], who develop a model for properly attributing costs in consolidated workload scenarios.

The inherent complexity of the workload placement decisions is targeted by research that seeks a practically reasonable combination of decision variables and constraints. Beloglazov et al. [53] show that a bin packing heuristic can reasonably minimize energy consumption under known processing requirements and dynamic voltage scaling. Li et al. [54] solve a similar problem where the processing requirements evolve over time, Dong et al. [55] extend the solution to include the impact of network traffic.

Besides illustrating the variety of concerns, the works mentioned above also support an important general observation—the existing research makes optimal workload consolidation tractable by limiting the set of considered interactions. Particularly useful is the assumption about processor utilization being mostly additive, i.e., assuming that colocating n workloads with individual processor utilizations u_1 to u_n results in an overall processor utilization of $\sum_{i=1}^{n} u_i$. This and similar assumptions are known to be approximate, and are investigated for various shared resources, such as processor caches and energy budget.

In the area of workload consolidation, the impact of processor caches was studied by Govindan et al. [56], who provide a model of cache sharing overhead calibrated on measurements. More studies model cache sharing behavior in varying detail [57–59].

In other research focusing on the energy budget, Raghavendra et al. [60] explain why coordinated power management is necessary and investigate a wide range of coordinated power management scenarios, Sarood et al. [61] show how using more computers with limited power can benefit energy efficiency of certain workloads, and Subramaniam et al. [27] use power limits to improve energy proportionality.

In context of the growing number of possible resource interactions impossible to evaluate comprehensively using resource models, it is essential that we perform experiments evaluating the impact of the distance between theoretical assumptions and the real system behavior on our research results—and this is where we believe our contribution connects to the existing research activities. We document the connection on recent research results, focusing on two aspects of the evaluation that we readily support—workload variety and interaction variety.

A large share of results on workload consolidation relies on simulation for evaluation [2,29,51,53–55,60,62,63]. Coarse granularity simulation often uses processor utilization log replay to approximate realistic workloads, assuming additive character of processor utilization. When considered, power usage is modeled for example as a function of utilization [29,51,53,60] or frequency [54]. Here, our work can contribute to a more realistic evaluation with an empirical model for deriving processor utilization and power consumption under workload colocation conditions—as we show, true utilization and consumption values on contemporary platforms can substantially differ from the results of existing models. More rarely, full system simulation is used for evaluation [2]. With full system simulation, the observations collected using our work may be difficult to attribute to the system elements tracked by the simulator.

As an alternative to simulation, many results on workload consolidation are evaluated using experimental measurements [3,4,8,16–20,30,56,64–68]. We should first observe that in these results, information about certain platform configuration aspects – such as how the frequency scaling mechanism was configured – is often not reported. More than half of the 14 studies examined used platforms that support frequency scaling (8 positive, 4 negative, 1 unclear, 1 disabled). Less than half of the studies (5 positive, 8 negative, 1 unclear) were conducted on platforms that support hardware threads. Only three platforms supported frequency boosting (3 positive, 9 negative, 2 unclear).[15] Additionally, more than half of the studies only used full utilization workloads (8 positive, 6 negative). Only one of the studies [19,68] might have used the full combination of hardware threads, frequency scaling, frequency boosting and partial utilization workloads together. In this context, our work highlights the potential impact when advanced hardware mechanisms are used with partial utilization workloads. Showstopper can conduct evaluation experiments that consider these factors together. Sensitivity of workloads to a variety of virtualized environments [69] and performance-driven utilization control [70] are further examples of emerging research directions that can benefit from a systematic approach to partial load and feedback control provided by Showstopper.

[15] Intel first introduced frequency boosting in 2007 as Dynamic Acceleration and later in 2008 as Turbo Boost.

9. Conclusion

Motivated by the widespread use of workload colocation, which leads to partial utilization of shared resources, and by the lack of a methodology to reproduce partial utilization conditions in experimental environments, we develop Showstopper—a novel, portable, and lightweight tool which controls the execution of multiple processes so as to follow a predefined CPU load trace.

In contrast to the existing practice of implementing workload intensity control in a workload-specific harness (or not at all), Showstopper provides a workload- and platform-agnostic way to achieve and sustain arbitrary partial loads by dynamically adjusting the duty cycle of controlled processes based on system feedback.

In our evaluation of load control accuracy on multiple platforms with six DaCapo benchmarks, Showstopper was able to sustain partial loads within 5% of the target load in 80% of all experiments on all platforms, and in 90% of all experiments on physical Linux-based machines. In general, the accuracy tends to get lower on virtual machines and with difficult-to-control heavily multi-threaded benchmarks.

The accurate load control provided by Showstopper was used to infer the relationship between benchmark throughput and a range of partial loads, showing that this relationship is often platform-dependent, non-linear, and even non-monotonic. Our methodology thus provides an accurate empirical alternative to operational analysis and similar methods used to relate resource utilization and application throughput.

Showstopper allows exploring resource sharing phenomena in detail and under many different scenarios, providing useful data for practical tasks such as system dimensioning or identification of performance anomalies in virtualization solutions. Moreover, Showstopper can reduce experiment time by replaying utilization traces at higher speeds. We show that in most cases, fast-forwarding a trace does not have a significant impact on the average system throughput, which enables obtaining measurements at a fraction of the original trace duration.

Further experimental study based on Showstopper illustrates the impact of workload type, partial background CPU load, and CPU pinning configuration on performance interference between pairs of colocated computationally-intensive workloads. If we only consider experiments in which the background workloads operate at a 5% and 95% CPU utilization, the study confirms results obtainable by traditional benchmarking approaches, in particular the presence of a significant performance interference under heavy background load, depending on the combination of the foreground and background workload types.

However, two other factors controlled in our experiments – the range of partial CPU loads and the CPU pinning configurations – provide the most interesting insights, especially concerning the impact of CPU pinning configurations on performance interference. This suggests that CPU pinning can be used in a dynamic fashion in response to system utilization. On a system with light background loads, the "per-thread" pinning configuration enables more efficient utilization of resources, leading to better performance. On a heavily loaded system, the "per-chip" pinning configuration improves CPU resource isolation, leading to more stable performance. Surprisingly, the "per-thread" pinning configuration appears to be the most predictable across diverse workload combinations.

Besides providing particular novel results, our experiments demonstrate that Showstopper is suitable for conducting large-scale experiments that provide tight control over resource utilization. Showstopper maintained controlled experimental conditions while running benchmarks with no built-in workload intensity control, and despite using advanced power management features that typically interfere with performance experiments.

Acknowledgments

This work is supported by the SNF project 200021_141002 and by Charles University institutional funding SVV-2016-260331.

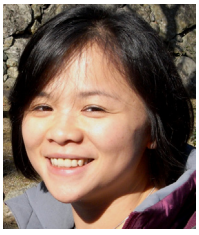
References

[1] J.S. Chase, D.C. Anderson, P.N. Thakar, A.M. Vahdat, R.P. Doyle, Managing energy and server resources in hosting centers, SIGOPS Oper. Syst. Rev. 35 (5) (2001) 103–116.
[2] N. Jerger, D. Vantrease, M. Lipasti, An evaluation of server consolidation workloads for multi-core designs, in: IISWC, IEEE CS, 2007, pp. 47–56.
[3] T. Dey, W. Wang, J.W. Davidson, M.L. Soffa, Characterizing multi-threaded applications based on shared-resource contention, in: ISPASS, IEEE CS, 2011, pp. 76–86.
[4] P. Apparao, R. Iyer, X. Zhang, D. Newell, T. Adelmeyer, Characterization & analysis of a server consolidation benchmark, in: VEE, ACM, 2008, pp. 21–30.
[5] C.S. Wong, I. Tan, R.D. Kumari, F. Wey, Towards achieving fairness in the Linux scheduler, SIGOPS Oper. Syst. Rev. 42 (5) (2008) 34–43.
[6] N. Rauschmayr, A. Streit, Reducing the memory footprint of parallel applications with KSM, in: Facing the Multicore-Challenge III, in: LNCS, vol. 7686, Springer, 2013, pp. 48–59.
[7] E. Rotem, A. Naveh, A. Ananthakrishnan, D. Rajwan, E. Weissmann, Power-management architecture of the Intel microarchitecture code-named Sandy Bridge, IEEE Micro 32 (2) (2012) 20–27.
[8] A. Corradi, M. Fanelli, L. Foschini, VM consolidation: A real case based on OpenStack cloud, Future Gener. Comput. Syst. 32 (2014) 118–127.
[9] P.J. Denning, J.P. Buzen, The operational analysis of queueing network models, ACM Comput. Surv. 10 (3) (1978) 225–261.
[10] A. Podzimek, L.Y. Chen, Transforming system load to throughput for consolidated applications, in: MASCOTS, IEEE CS, 2013, pp. 288–292.
[11] A. Podzimek, L.Y. Chen, L. Bulej, W. Binder, P. Tuma, Showstopper: The partial CPU load tool, in: MASCOTS, IEEE CS, 2014, pp. 510–513.
[12] A. Podzimek, L. Bulej, L.Y. Chen, W. Binder, P. Tøuma, Analyzing the impact of CPU pinning and partial CPU loads on performance and energy efficiency, in: CCGRID, IEEE, 2015.
[13] Cloudwatch. http://aws.amazon.com/cloudwatch.
[14] Z. Gong, X. Gu, PAC: Pattern-driven application consolidation for efficient cloud computing, in: MASCOTS, IEEE, 2010, pp. 24–33.
[15] J. Choi, S. Govindan, B. Urgaonkar, A. Sivasubramaniam, Profiling, prediction, and capping of power consumption in consolidated environments, in: MASCOTS, IEEE, 2008, pp. 1–10.
[16] N. Huber, M. von Quast, M. Hauck, S. Kounev, Evaluating and modeling virtualization performance overhead for cloud environments, in: CLOSER, SciTePress, 2011, pp. 563–573.
[17] P. Padala, X. Zhu, Z. Wang, S. Singhal, K.G. Shin, Performance evaluation of virtualization technologies for server consolidation.
[18] F. Chen, J. Grundy, Y. Yang, J.-G. Schneider, Q. He, Experimental analysis of task-based energy consumption in cloud computing systems, in: ICPE, ACM, 2013, pp. 295–306.
[19] R. Krebs, C. Momm, S. Kounev, Metrics and techniques for quantifying performance isolation in cloud environments, in: QOSA, ACM, 2012, pp. 91–100.
[20] P. Apparao, R. Iyer, D. Newell, Towards modeling & analysis of consolidated cmp servers, SIGARCH Comput. Archit. News 36 (2) (2008) 38–45.
[21] K.D. Lange, J.A. Arnold, H. Block, N. Totura, J. Beckett, M.G. Tricker, Further implementation aspects of the server efficiency rating tool (SERT), in: ICPE, ACM, 2013, pp. 349–360.
[22] S.M. Blackburn, R. Garner, C. Hoffman, A.M. Khan, K.S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S.Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J.E.B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, B. Wiedermann, The DaCapo benchmarks: Java benchmarking development and analysis, in: OOPSLA, ACM, 2006.
[23] T. Martinec, L. Marek, A. Steinhauser, P. Tůma, Q. Noorshams, A. Rentschler, R. Reussner, Constructing performance model of JMS middleware platform, in: ICPE, ACM, 2014, pp. 123–134.
[24] L.Y. Chen, D. Ansaloni, E. Smirni, A. Yokokawa, W. Binder, Achieving application-centric performance targets via consolidation on multicores: myth or reality?, in: Proc. High-Performance Parallel and Distributed Computing, ACM, 2012, pp. 37–48.
[25] D. Ansaloni, L.Y. Chen, E. Smirni, W. Binder, Model-driven consolidation of Java workloads on multicores, in: DSN, IEEE, 2012, pp. 1–12.
[26] T. Wood, L. Cherkasova, K.M. Ozonat, P.J. Shenoy, Profiling and modeling resource usage of virtualized applications, in: Middleware, 2008, pp. 366–387.
[27] B. Subramaniam, W.-c. Feng, Towards energy-proportional computing for enterprise-class server workloads, in: ICPE, ACM, 2013, pp. 15–26.
[28] H. Liu, A measurement study of server utilization in public clouds, in: DASC, IEEE CS, 2011, pp. 435–442.
[29] A. Verma, G. Dasgupta, T.K. Nayak, P. De, R. Kothari, Server workload analysis for power minimization using consolidation, in: ATC, USENIX, 2009, 28–28.
[30] Y. Song, Y. Zhang, Y. Sun, W. Shi, Utility analysis for Internet-oriented server consolidation in VM-based data centers, in: CLUSTER, IEEE CS, 2009, pp. 1–10.
[31] cpulimit. http://github.com/opsengine/cpulimit.
[32] J. Mauro, R. McDougall, Solaris Internals: Solaris 10 and OpenSolaris Kernel Architecture, Prentice Hall, 2006.

[33] A. Sewe, M. Mezini, A. Sarimbekov, W. Binder, Da capo con Scala: Design and analysis of a Scala benchmark suite for the Java virtual machine, in: OOPSLA, ACM, 2011, pp. 657–676.
[34] stress. http://freecode.com/projects/stress.
[35] Spec cpu benchmark suites. http://www.spec.org/cpu.
[36] cpuloadgen. http://github.com/ptitiano/cpuloadgen.
[37] S. Perarnau, G. Huard, Krash: Reproducible CPU load generation on many-core machines, in: IPDPS, IEEE, 2010, pp. 1–10.
[38] E. Le Sueur, G. Heiser, Slow down or sleep, that is the question, in: ATC, USENIX, 2011.
[39] M. Jägemar, S. Eldh, A. Ermedahl, B. Lisper, Towards feedback-based generation of hardware characteristics, in: 7th International Workshop on Feedback Computing, 2012.
[40] L.Y. Chen, A. Das, A. Sivasubramaniam, Q. Wang, R. Harper, M. Bland, Consolidating clients on back-end servers with co-location and frequency control, in: SIGMETRICS/Performance, 2006, pp. 383–384.
[41] J.D. Moore, J.S. Chase, P. Ranganathan, Weatherman: Automated, online and predictive thermal mapping and management for data centers, in: ICAC, 2006, pp. 155–164.
[42] S. Pelley, D. Meisner, P. Zandevakili, T.F. Wenisch, J. Underwood, Power routing: dynamic power provisioning in the data center, in: ASPLOS, 2010, pp. 231–242.
[43] X. Wang, X. Fu, X. Liu, Z. Gu, Power-aware CPU utilization control for distributed real-time systems, in: RTAS, IEEE, 2009, pp. 233–242.
[44] X. Meng, C. Isci, J. Kephart, L. Zhang, E. Bouillet, D. Pendarakis, Efficient resource provisioning in compute clouds via VM multiplexing, in: ICAC, 2010, pp. 11–20.
[45] M. Björkqvist, L.Y. Chen, W. Binder, Dynamic replication in service-oriented systems, in: CCGRID, IEEE, 2012, pp. 531–538.
[46] M. Björkqvist, L.Y. Chen, W. Binder, Opportunistic service provisioning in the cloud, in: CLOUD, IEEE, 2012, pp. 237–244.
[47] X. Fan, W.-D. Weber, L.A. Barroso, Power provisioning for a warehouse-sized computer, SIGARCH Comput. Archit. News 35 (2) (2007) 13–23.
[48] V. Babka, Cache sharing sensitivity of SPEC CPU2006 benchmarks, 2009. http://d3s.mff.cuni.cz.
[49] V. Babka, L. Bulej, M. Decky, J. Kraft, P. Libic, L. Marek, C. Seceleanu, P. Tuma, Resource usage modeling, 2008. http://d3s.mff.cuni.cz.
[50] G. Pacifici, W. Segmuller, M. Spreitzer, A. Tantawi, Dynamic estimation of CPU demand of web traffic, in: VALUETOOLS, ACM, 2006, p. 26.
[51] V. Ishakian, R. Sweha, A. Bestavros, J. Appavoo, CloudPack, in: Middleware, in: LNCS, vol. 7662, Springer, 2012, pp. 374–393.
[52] J. Wang, R. Hua, Y. Zhu, J. Wan, C. Xie, Y. Chen, RO-BURST: A robust virtualization cost model for workload consolidation over clouds, in: CCGrid, IEEE CS, 2012, pp. 490–497.
[53] A. Beloglazov, J. Abawajy, R. Buyya, Energy-aware resource allocation heuristics for efficient management of data centers for cloud computing, Future Gener. Comput. Syst. 28 (5) (2012) 755–768.
[54] J. Li, K. Shuang, S. Su, Q. Huang, P. Xu, X. Cheng, J. Wang, Reducing operational costs through consolidation with resource prediction in the cloud, in: CCGrid, IEEE CS, 2012, pp. 793–798.
[55] J. Dong, X. Jin, H. Wang, Y. Li, P. Zhang, S. Cheng, Energy-saving virtual machine placement in cloud data centers, in: CCGrid, IEEE CS, 2013, pp. 618–624.
[56] S. Govindan, J. Liu, A. Kansal, A. Sivasubramaniam, Cuanta: Quantifying effects of shared on-chip resource interference for consolidated virtual machines, in: SOCC, ACM, 2011, pp. 22:1–22:14.
[57] X. Chen, T. Aamodt, A first-order fine-grained multithreaded throughput model, in: HPCA, IEEE CS, 2009.
[58] V. Babka, P. Libič, T. Martinec, P. Tůma, On the accuracy of cache sharing models, in: ICPE, ACM, 2012, pp. 21–32.
[59] A. Sandberg, A. Sembrant, E. Hagersten, D. Black-Schaffer, Modeling performance variation due to cache sharing, in: HPCA, IEEE CS, 2013, pp. 155–166.
[60] R. Raghavendra, P. Ranganathan, V. Talwar, Z. Wang, X. Zhu, No "power" struggles: Coordinated multi-level power management for the data center, in: ASPLOS, ACM, 2008.
[61] O. Sarood, A. Langer, L. Kale, B. Rountree, B. de Supinski, Optimizing power allocation to CPU and memory subsystems in overprovisioned HPC systems, in: CLUSTER, IEEE CS, 2013, pp. 1–8.
[62] V. Ishakian, A. Bestavros, MORPHOSYS: Efficient colocation of QoS-constrained workloads in the cloud, in: CCGrid, IEEE CS, 2012, pp. 90–97.
[63] Y. Liu, Y. He, M. Li, J. Wang, K. Liu, X. Li, Does wireless sensor network scale? A measurement study on GreenOrbs, IEEE Trans. Parallel Distrib. Syst. 24 (10) (2013) 1983–1993.
[64] J. Zhang, H. Wang, H. Lin, W. chun Feng, Consolidating applications for energy efficiency in heterogeneous computing systems, in: HPCC, IEEE CS, 2013, pp. 399–406.
[65] T. Wood, P. Shenoy, A. Venkataramani, M. Yousif, Sandpiper: Black-box and gray-box resource management for virtual machines, Comput. Netw. 53 (17) (2009) 2923–2938.
[66] A. Scolari, F. Sironi, D.B. Bartolini, D. Sciuto, M.D. Santambrogio, Coloring the cloud for predictable performance, in: SOCC, ACM, 2013, pp. 47:1–47:2.
[67] C. Ragusa, P. Robinson, S. Svorobej, A framework for modeling and executing of infrastructure contention experiments, in: MERMAT, IEEE CS, 2013.
[68] R. Krebs, S. Spinner, N. Ahmed, S. Kounev, Resource usage control in multi-tenant applications, in: CCGrid, IEEE CS, 2014, pp. 122–131.
[69] Z. Estrada, Z. Stephens, C. Pham, Z. Kalbarczyk, R. Iyer, A performance evaluation of sequence alignment software in virtualized environments, in: CCGrid, IEEE CS, 2014, pp. 730–737.
[70] W. Zhang, S. Rajasekaran, T. Wood, M. Zhu, MIMP: Deadline and interference aware scheduling of Hadoop virtual machines, in: CCGrid, IEEE CS, 2014, pp. 394–403.

Andrej Podzimek is a Ph.D. candidate in the Department of Distributed and Dependable Systems, Faculty of Mathematics and Physics, Charles University in Prague. He holds an M.Sc. from the Charles University in Prague. His main research interests are in the areas of concurrent algorithms and data structures, operating systems, and performance evaluation.

Lubomír Bulej is a researcher at the Faculty of Informatics, University of Lugano, Switzerland, currently on leave from his assistant professor position at the Department of Distributed and Dependable Systems, Charles University in Prague, Czech Republic. He holds an M.Sc. from the Czech Technical University in Prague, and a Ph.D. from the Charles University in Prague. His main research interests are in the areas of performance evaluation and modeling, program analysis, and operating systems.



Lydia Y. Chen is a performance analyst at the Energy Management group of IBM Zurich Research Lab. She holds a Ph.D. in Operations Research and Industrial Engineering from Penn State University. She completed her undergraduate studies at National Taiwan University and British Columbia University. Her main research interests are in the areas of performance evaluation, power and workload management, big data and cloud computing and architecture-aware parallel algorithms.

Walter Binder is an associate professor in the Faculty of Informatics, University of Lugano, Switzerland. He holds an M.Sc., a Ph.D., and a Venia Docendi from the Vienna University of Technology, Austria. Before joining the University of Lugano, he was post-doctoral researcher at the Artificial Intelligence Laboratory, École Polytechnique Fédérale de Lausanne, Switzerland. His main research interests are in the areas of program analysis, virtual machines, parallel programming, and Cloud computing.

Petr Tůma is an associate professor and department chair in the Department of Distributed and Dependable Systems, Faculty of Mathematics and Physics, Charles University in Prague, Czech Republic. He holds an M.Sc. from the Czech Technical University, and a Ph.D. from the Charles University in Prague. His main research interests are in the areas of performance evaluation and modeling, and distributed systems.

ID	Title	Pages
424780	Robust partial-load experiments with Showstopper	24

http://fulltext.study/journal/451

FullText.study

Categorized Journals
Thousands of scientific journals broken down into different categories to simplify your search

Full-Text Access
The full-text version of all the articles are available for you to purchase at the lowest price

Free Downloadable Articles
In each journal some of the articles are available to download for free

Free PDF Preview
A preview of the first 2 pages of each article is available for you to download for free

http://FullText.Study