# Showstopper: The Partial CPU Load Tool

Andrej Podzimek[1,3], Lydia Y. Chen[2], Lubomír Bulej[1], Walter Binder[1], and Petr Tůma[3]

[1]Faculty of Informatics, University of Lugano
[2]IBM Zurich Research Lab
[3]Department of Distributed and Dependable Systems, Charles University in Prague

***Abstract*—Provisioning strategies relying on CPU load may be suboptimal for many applications, because the relation between CPU load and application performance can be non-linear and complex. With the knowledge of the relation between CPU load and application performance, resource provisioning strategies could be tuned to a particular application, but the required knowledge is difficut to obtain, because classic benchmarking is not suited for performance evaluation of partial-load scenarios. As a remedy, we present *Showstopper*, a tool capable of achieving and sustaining a predefined partial CPU load (or replay a load trace) by controlling the execution of arbitrary CPU-bound workloads. By analyzing performance interference among applications running in colocated virtual machines, we demonstrate how *Showstopper* enables systematic and reproducible exploration of the platform- and application-specific relation between CPU load and application performance.**

## I. Introduction

Strategies for automatic resource management in virtualized environment are mainly based on resource utilization—cloud service providers even offer support for custom rule-based strategies for spawning virtual machine instances in response to resource utilization. However, because storage and networking resources in cloud environment are often hidden behind high-level abstractions, their utilization is difficult to interpret, because it lacks an obvious cap. Most resource provisioning decisions are thus based on CPU utilization, which is commonly provided by operating systems and has an obvious interpretation.

The problem with decisions based on CPU load is that they generally follow best practices and rules of thumb, e.g., spawning a new virtual machine when system CPU load exceeds 80%, which may be suboptimal for many applications. For example, Martinec et al. [1] have recently shown that the throughput of the ActiveMQ JMS broker can more than double when CPU utilization increases from 80% to 95%, with a choke point still in a safe distance at almost three times the throughput at 80% utilization. Although this cannot be generalized, it supports our earlier findings [2] that the relation between CPU utilization and application throughput is complex and difficult to predict. Consequently, there is room for significant improvement in utilization of computing resources for certain applications.

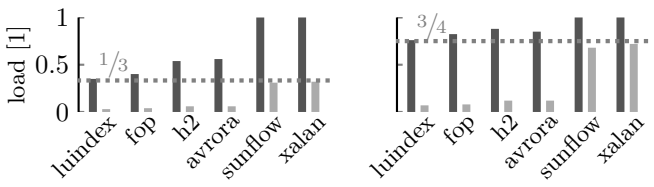
Tapping these resource reserves requires knowledge of the dependency of an application's throughput on CPU utilization on a particular platform. This is difficult to obtain, because traditional benchmarking is focused on determining the maximum throughput that can be achieved on a particular system with all resources at the application's disposal. However, in real-world situations, most systems only exhibit partial utilizations during normal operation, which is difficult to reproduce during benchmarking. While most data centers collect CPU utilization logs, these are difficult to correlate with application throughput. To overcome this problem, we have recently presented Showstopper [2], a methodology and tool for benchmarking that allows replaying CPU load traces in a fast-forwarding and reproducible fashion to obtain an application throughput trace corresponding to the CPU load trace.

In this paper, we demonstrate the use of Showstopper in analyzing performance of CPU-intensive workloads under partial load and we specifically focus on analyzing performance interference among applications sharing hardware resources in a common scenario with collocated virtual machines. Unlike existing approaches, Showstopper is capable of achieving and sustaining arbitrary partial CPU loads using arbitrary CPU-bound workloads. Consequently, it enables systematic exploration of the relationship between CPU load and the performance of a particular application on a given platform—without Showstopper, such experiments would be very difficult to perform, because most benchmarks do not support partial-load scenarios.

## II. Background

Achieving and sustaining a partial CPU load is difficult, especially considering systems with many-core CPUs, NUMA, and power management strategies, in addition to a variety of workload characteristics, and operating system scheduling policies. We discuss these difficulties in detail in our prior work [2]. Here we mainly review and illustrate the contrast between naïve approaches to achieving partial loads and the Showstopper approach.

Given a system with $N$ CPUs and a benchmark representing a certain workload, there are several ways to approximate the desired partial system load. We consider (1) running the benchmark in multiple instances, with the number of instances proportional to $N$ (the n-bench method), (2) running a single benchmark instance and interrupting it periodically to enforce a certain duty cycle (the 1-DC method), and (3) running $N$ benchmark instances while enforcing a duty cycle (the N-DC method).




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

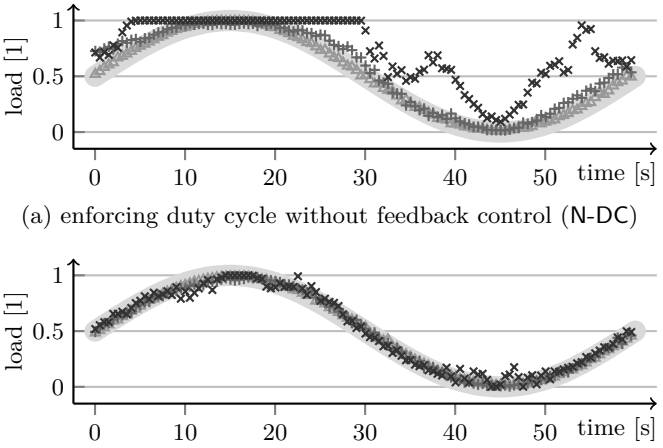
Figure 1: Naïve approaches to load control, n-bench (dark) and 1-DC (light): CPU load average over a 1-minute run after 3 minutes of warm-up.

These methods have obvious drawbacks. The n-bench method can only target partial loads with granularity of $1/N$, and the ability of all the naïve methods to achieve and sustain the desired partial load is highly dependent on the workload characteristics. For example, the 1-DC method is not suitable for single-threaded workloads, while the n-bench and N-DC methods will have trouble controlling heavily multi-threaded workloads.

We illustrate these drawbacks in Figure 1, showing the n-bench and 1-DC methods mostly failing to achieve target loads of 1/3 (1a) and 3/4 (1b) when running DaCapo [3] workloads on a 12-CPU system. As expected, results of both methods strongly depend on the workloads: luindex and fop can be considered single-threaded, because a single instance is unable to induce loads much higher than $1/N$. In contrast, sunflow and xalan can be considered heavily multi-threaded, because a single instance is capable of saturating the system. Finally, avrora and h2 fall somewhere in between—both are multi-threaded, but the number of runnable threads varies quickly, making them a poor fit for all the naïve methods.

To explore the relationship between CPU load and application throughput, a partial load must be sustained with sufficient accuracy. This requires (1) spawning a sufficient number of workload instances (capable of saturating the system when left uncontrolled) and (2) varying the duty cycle of the controlled workload based on feedback from the system (instead of using a fixed duty cycle corresponding to the target partial load). The need for feedback control is not immediately obvious, but it is necessary to maintain control in face of dynamically changing application behavior, background load, and scheduling noise.

To illustrate the importance of feedback control, in Figure 2 we compare two different methods used to enforce a synthetic sine-shaped load trace on 12 workload instances on a 12-CPU system. Figure 2a shows the result of using the N-DC method, with the duty cycle set to the target load dictated by a trace generator. As expected, the method works well with the luindex (single-threaded) workload, tends to higher loads with the h2 (non-saturating multi-threaded) workload, and fails completely with the sunflow (heavily multi-threaded) workload. In contrast, Figure 2b shows Showstopper controlling the same workloads using a feedback control mechanism, which compensates both for short-term load fluctuations (caused by a quickly varying number of runnable threads) and for long-term bias (related




(a) enforcing duty cycle without feedback control (N-DC)

(b) using Showstopper's feedback control to adapt duty cycle

Figure 2: Following a synthetic sine-shaped load trace (luindex △, h2 +, sunflow ×). The light grey band represents the target load.

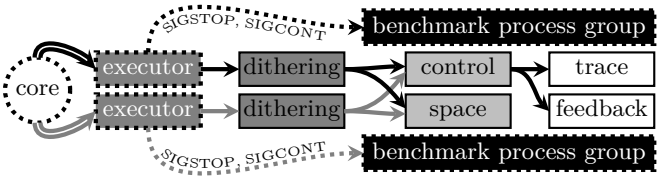



Figure 3: A Showstopper-driven benchmark. Rectangles with dotted and solid borders represent processes and Showstopper modules, respectively. Double lines represent the POSIX process hierarchy. Dotted lines indicate communication via POSIX signals. Solid lines represent library calls within a process.

to the ratio of the number of runnable threads and the number of CPUs). Note especially the absence of the uncontrolled saturation with the sunflow benchmark.

## III. Showstopper

The Showstopper tool implements a classic feedback control loop in a modular fashion, with different module types responsible for different aspects of the loop. This arrangement allows rapid experimenting with alternate implementations of various parts of the control algorithm.

We illustrate the Showstopper architecture in Figure 3 on an experiment with two controlled workloads (black components, provided by users) running in parallel. Showstopper consists of a core process and a set of executor processes. The core process initializes shared memory areas for communication among executors and keeps track of executors it spawns. Each executor is responsible for one controlled workload and all its child processes—it starts it and dynamically controls its duty cycle[1] to achieve and sustain the target load.

Each executor represents an instance of a local (per-benchmark) control loop cooperating with other executors in a distributed fashion to achieve global (system-wide) target load. The difference between the current target load (from the trace module) and the estimated actual load (from the feedback module) is used by a control module to

[1] enforced by controlling the duration of the benchmark's runnable and stopped states using the SIGSTOP and SIGCONT signals

compute the global average duty cycle to be enforced by Showstopper. To determine the contribution of individual benchmarks to the global duty cycle, a space module calculates a local duty cycle for each benchmark so that the average of local duty cycles matches the global duty cycle. A dithering module then transforms the local duty cycle into durations of runnable and stopped states to be enforced by an executor. A more detailed description of each module type can be found in our earlier work [2]. In the following, we highlight features of interest to users.

When using Showstopper to control experimental workloads, users have to provide the target loads through an appropriate trace module. Currently, Showstopper supports loading traces from files, or generating synthetic (constant, sine-shaped, random) traces.

Users are free to experiment with different control implementations—due to a wide variety of workload characteristics, there is no absolute winner in terms of stability and accuracy of load control. However, we would like to highlight a new control module, which implements a fully-fledged proportional-integral (PI) controller. The PI controller generally shows improved stability (in terms of mean squared error) over other controller implementations and serves as a reasonable default.

Finally, a dithering module specifies the scheduling-related behavior of the workload. Showstopper provides two implementations, fixedq and averageq. fixedq uses a (short) fixed time quantum as a basic time unit, alternating stopped and runnable states of that duration, as necessary to achieve the local duty cycle (hence the relation to *dithering*). averageq, on the other hand, operates with (almost) continuous time quantities, splitting a fixed time quantum $q$ into the durations of stopped and runnable states according to the local duty cycle; e.g., for a duty cycle of $1/3$, querying runnable and stopped state durations would yield $q/3$ and $2q/3$, respectively. Both modules enforce the same duty cycle, each of them in a different way.

Figure 4 shows an example of using Showstopper to run a complex benchmark following a custom load trace. The sstrace tool compiles the trace into a format readable by Showstopper. Users can explicitly specify Showstopper's modules and their options (as in Figure 4) or use the defaults. Showstopper automatically spawns and controls 12 instances of luindex with scratch directories `scratch.00` through `scratch.11`. The resulting CPU load can be observed using a load monitor, as shown in Figure 5. Despite running 12 workload instances (and a desktop environment) on an 8-CPU system, Showstopper follows the trace accurately, without bias towards higher loads.

## IV. Applications of Showstopper

Showstopper can be easily used to explore the dependency of application performance on system load. We illustrate this by having Showstopper run and control a set of workloads using a synthetic "staircase" CPU load trace which enforces a range of different partial loads so that a statistically significant throughput observation can be collected for each load level. Because throughput

```
$ sstrace > saw.trace     $ showstopper -n 12              \
0    1/10                     -t file -T saw.trace         \
20s  6/10                     -f realtime -F 103m          \
40s  1/10                     -c rbpid -s even             \
1M   9/10                     -d averageq -D 51m           \
80s  4/10                   java -jar dacapo.jar -n 99999  \
100s 9/10                     -scratch-directory=scratch.% \
2M   1/10                     luindex
```

Figure 4: Starting Showstopper to control 12 instances of luindex and to follow a synthetic hand-written load trace. '%' is replaced by a workload instance number.

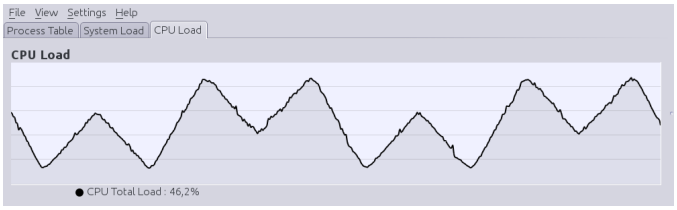



Figure 5: Showstopper following the load trace from Figure 4 (in a loop, with linear interpolation), observed by the KDE load monitor (ksysguard) with a sampling interval of 0.5 s.

measurements require long warm-up periods (especially with Java), the trace starts with a 10-minute uncontrolled period, which warms-up each benchmark run.

Figure 6 shows the result of an experiment with 16 luindex instances running in a 16-CPU KVM-based virtual machine. The target load values first decrease from $19/20$ to $1/20$ and then increase back to $19/20$; the warm-up period is not shown. The symmetry of the trace enables additional results validation and allows detecting performance anomalies[2] in the workload over a long run. We observe that Showstopper follows the "staircase" trace accurately, with only minor fluctuations. The dependency of the combined throughput[3] of the 16 luindex instances on the system CPU load turns out to be non-linear. This appears to be the case—platform and workload specific—for most DaCapo workloads. We omit these results due to space constraints.

To demonstrate using Showstopper for investigating performance interference between colocated virtual machines, we perform two experiments using two 16-CPU VMs running on a 32-CPU physical server. In the first VM we run 16 Showstopper-controlled luindex instances, forced to follow the "staircase" CPU load trace. In the second VM we run 16 uncontrolled h2 instances, which completely saturate the VM. During the experiment, we observe CPU load and application performance in both VMs.

We are interested in how the varying CPU load in the luindex VM influences the performance of the uncontrolled workload in the h2 VM. Specifically, we investigate whether it makes sense (in terms of performance and resource isolation) to pin each VM to a different NUMA node or to leave the VMs unpinned, i.e., to let them migrate freely, as the physical server's kernel scheduler sees fit.

[2]For example, a non-symmetric shape of the throughput trace. This was actually the case with the fop workload (not shown).

[3]For each interval of 6 seconds, we compute the throughput by adding 1 for each iteration that starts and ends within that interval and a value lower than 1 for an iteration spanning multiple intervals—proportionally to the part of the interation overlapping the interval.

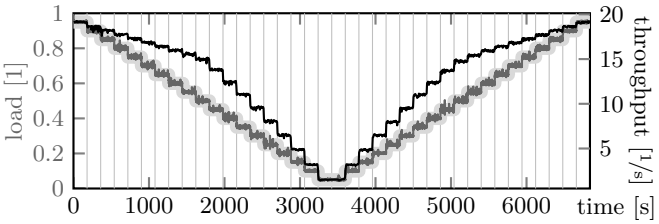



Figure 6: CPU load (——) and throughput (——) of luindex when following the "staircase" trace using Showstopper. The light grey band represents target load.

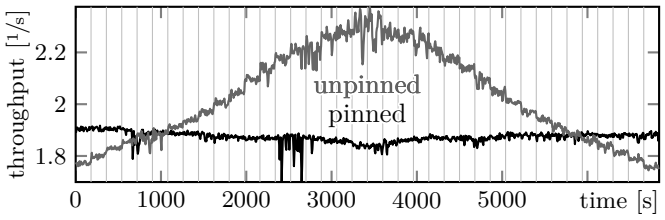



Figure 7: Throughput of uncontrolled h2 running at the same time as luindex in Figure 6 on a neighboring VM. The two VMs are pinned to separate NUMA nodes (——) or unpinned (——).

When pinned to different NUMA nodes, the two VMs should not compete for CPUs, caches, or memory bandwith, and we do not expect significant performance interference. When the two VMs are unpinned, they do not directly compete for CPUs, because the total number of their CPUs does not exceed the number of physical CPUs. However, they may still compete for other hardware resources, such as CPU caches and memory bus bandwidth, and we expect significant performance interference.

The results in Figure 7 show the performance of the uncontrolled h2 workload in time, corresponding to the controlled luindex workload (shown in Figure 6) running in the neighboring VM. We observe that pinning the VMs to distinct NUMA nodes has a negative effect on performance when the VM running the luindex workload is lightly loaded (i.e., below 3/5). This is most likely because a pinned VM can not migrate to idle processors on the other NUMA node and thus benefit from additional space in their (lightly contended) cache. When the luindex VM is heavily loaded, pinning the VMs to different NUMA nodes provides almost perfect CPU resource isolation, allowing the h2 VM to retain its performance by avoiding heavy resource contention.

Surprisingly, the h2 workload benefits slightly-but-visibly from the neighboring luindex workload with the two VMs pinned to different NUMA nodes. At this point, we can only speculate of the reasons behind this phenomenon—one hypothesis being that a more active neighbor may assist in keeping the system out of low-power states.[4]

In general, the performance of the unpinned VMs appears to be better in most cases, except when both VMs are heavily loaded. The tradeoff between an almost perfect resource isolation and maximum performance due to full utilization of the available CPU resources suggests that CPU pinning should be used adaptively, only when CPU loads exceed certain levels.

## V. Related Work

A workload generator tool is vital for resource allocation studies. Existing tools either control only one workload at a time [4], [5] or do not support partial loads [6]. Load generators and control mechanisms in [7] mimic a telecommunication server in terms of both CPU load and cache misses; however, this requires a built-in load generator instead of controlling arbitrary workloads. A partial load generator closest to Showtopper is KRASH [8], generating a controlled system load and replaying load traces. KRASH provides only a subset of Showstopper's flexible functionality, e.g., there are no counterparts to Showstopper's *space* and *dithering* module types.

[4]We use standard production hardware settings, i.e., TurboBoost and frequency scaling are enabled, so is SMT and power-related features of modern CPUs, as is the case for most production workloads Showstopper is supposed to mimic.

## VI. Conclusion

The nature of experiments enabled by Showstopper is highly relevant to the study of performance in virtualized environment, especially in the context of workload consolidation and VM colocation on modern hardware platforms.

In this paper, we used Showstopper to systematically explore the relationship between system CPU load and application performance in different scenarios, commonly found in virtualized environment. Obtaining these insights without Showstopper would be difficult at best, because most existing benchmarks and applications do not explicitly support performance evaluation at partial system loads—Showstopper allows using them without modifications. These insights are necessarily platform and application specific, but we believe that the knowledge of application performance behavior at partial load can contribute to better provisioning strategies and VM placement decisions.

Showstopper runs on the Linux and AIX operating systems and can be easily ported to POSIX systems.

*Acknowledgements:* This work was supported by the Rectors' Conference of the Swiss Universities project Sciex-NMS-CH 12.211, by the Swiss National Science Foundation projects CRSII2_136225 and 200021_141002, and by the EU project ASCENS 257414.

## References

[1] Martinec, T., et al., "Constructing performance model of JMS middleware platform," in *Proc. 5th ACM/SPEC Intl. Conf. on Performance Engineering*. ACM, 2014, pp. 123–134.
[2] A. Podzimek and L. Y. Chen, "Transforming system load to throughput for consolidated applications," in *MASCOTS*. IEEE CS, 2013, pp. 288–292.
[3] Blackburn, S. M. et al., "The DaCapo benchmarks: Java benchmarking development and analysis," in *OOPSLA*. ACM Press, 2006, pp. 169–190.
[4] "cpulimit," http://github.com/opsengine/cpulimit.
[5] "cpuloadgen," http://github.com/ptitiano/cpuloadgen.
[6] "stress," http://freecode.com/projects/stress.
[7] M. Jägemar, S. Eldh, A. Ermedahl, and B. Lisper, "Towards feedback-based generation of hardware characteristics," in *7th International Workshop on Feedback Computing*, 2012.
[8] S. Perarnau and G. Huard, "Krash: Reproducible CPU load generation on many-core machines," in *IPDPS*. IEEE, 2010, pp. 1–10.