Università della Svizzera italiana | Faculty of Informatics

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, Università della Svizzera italiana

[2]IBM Zurich Research Lab

[3]Department of Distributed and Dependable Systems, Charles University in Prague

Problem Description

Induce a partial CPU load using existing applications. What is a controlled partial CPU load good for?

- Partial load experiments with reproducible results
- Relation between CPU load and application performance
- Relation between CPU load and power consumption
- Impact of a changing CPU load in a neighboring virutal machine
- Optimal CPU pinning and frequency scaling configurations
- Effects and benefits of frequency scaling, TurboBoost and NUMA

CPU Load is the fraction of time CPUs spend executing threads.

Duty Cycle is the fraction of time an application can run.

Given a workload capable of hogging a CPU, how can we control the average CPU load and sustain a partial load?

By the number of workload instances:
What if the workload runs a varying number of threads? Only a limited number of partial load values and only one distribution of load across CPUs ("all or nothing") can be achieved. Dark columns in Figure 1 show that the approach works for mostly single-threaded applications (luindex, fop), but fails otherwise.

By the duty cycle of one workload instance:
What if there are not enough threads to saturate all CPUs? Then the CPU load will be lower than the duty cycle. Light columns in Figure 1 show that the approach works for heavily multi-threaded applicartions (sunflow, xalan), but fails otherwise.

By the duty cycle of #CPUs workload instances:
What if each workload instance runs multiple threads? Multi-threadedness causes the load to exceed the duty cycle, unless we synchronize duty cycle enforcement across all instances, Yet we do **not** want any synchronization among parallel workloads. Figure 2 shows that the approach works for the single-threaded luindex (△), tends towards higher loads for the slightly multi-threaded h2 (+) and spins out of control for the heavily multi-threaded sunflow (×). The experiment uses a synthetic sine-shaped load trace and enforces a duty cycle equal to the target CPU load.

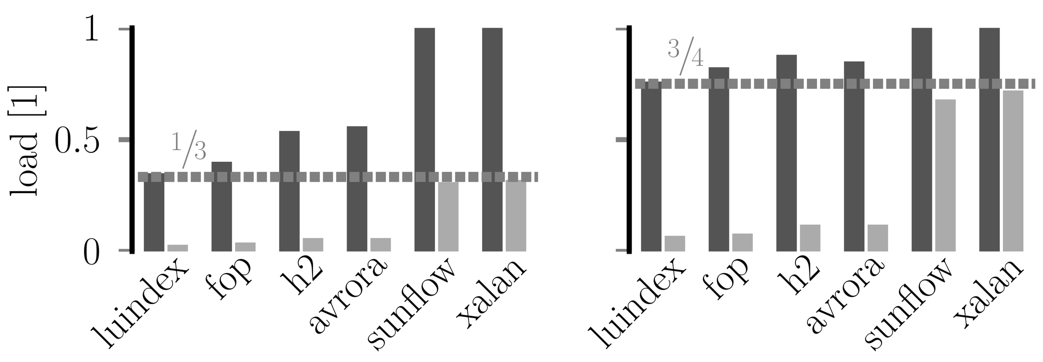

Figure 1: Two naïve approaches to load control: A number of workload instances proportional to #CPUs without duty cycle enforcement and one workload instance granted a duty cycle equal to the target CPU load. Target CPU loads are 1/3 (left) and 3/4 (right). Displayed are the observed CPU load averages over a 1-minute run after 3 minutes of warm-up.

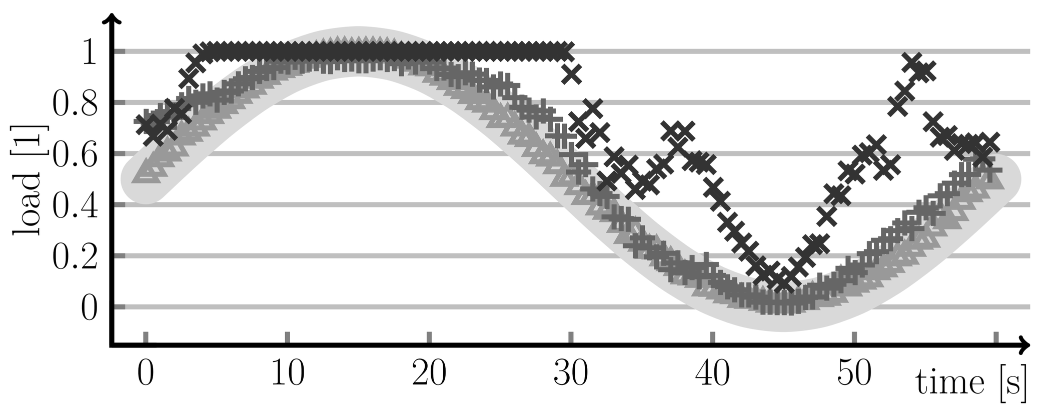

Figure 2: Another naïve approach to load control: #CPUs workload instances are granted a duty cycle equal to the CPU load target. The duty cycle follows a synthetic sine-shaped load trace. Displayed are CPU load observations taken every 500 ms for luindex (△), h2 (+) and sunflow (×), illustrating the imprecision of this crude load control mechanism. The center of the light gray band represents the target CPU load.

Our Solution: Showstopper

Showstopper is a software tool that uses feedback control mechanisms to achieve and sustain a partial CPU load.

- We start #CPUs workload instances using Showstopper.
- Showstopper's **feedback controller** responds to system load.
- The controller's **input** is the CPU load observed on the system.
- The controller's **output** is the duty cycle to enforce.
- The controller adjusts the duty cycle to meet the load target.

Figure 3 shows how Showstopper resolves the load control problems shown in Figure 2. Even #CPUs instances of the heavily multi-threaded sunflow can be forced to follow the required CPU load trace.

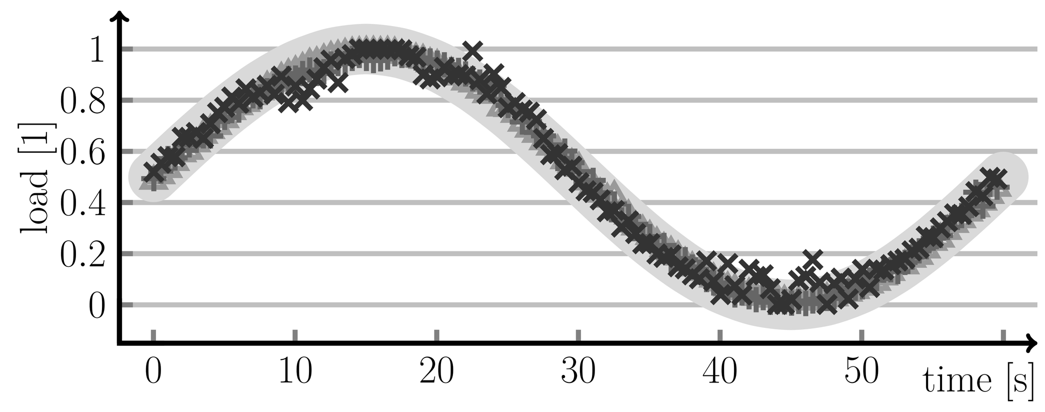

Figure 3: Showstopper is used to follow a synthetic sine-shaped load trace, controlling #CPUs workload instances and using feedback control mechanisms to adapt the duty cycle. Displayed are load observations taken every 500 ms (luindex △, h2 +, sunflow ×).

Example Application

Let us define and re-play a synthetic load trace, using **Showstopper** to start and control 12 instances of luindex. Figure 4 shows the process of encoding a load trace and starting Showstopper. Figure 5 shows the resulting sytem load, as seen in the KDE load monitoring tool. Note that Showstopper performs a linear interpolation between load and time values. There is no tendency towards higher loads, although we run **12 workload instances on an 8-CPU system**. **Feedback control** algorithms compensate for both temporary fluctuations and long-term bias.

```
$ 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: First a load trace definition is encoded into a format readable by Showstopper (left). Next Showstopper starts 12 instances of a workload and follows the load trace (right). (The '%' symbol is replaced by a workload instance number.)

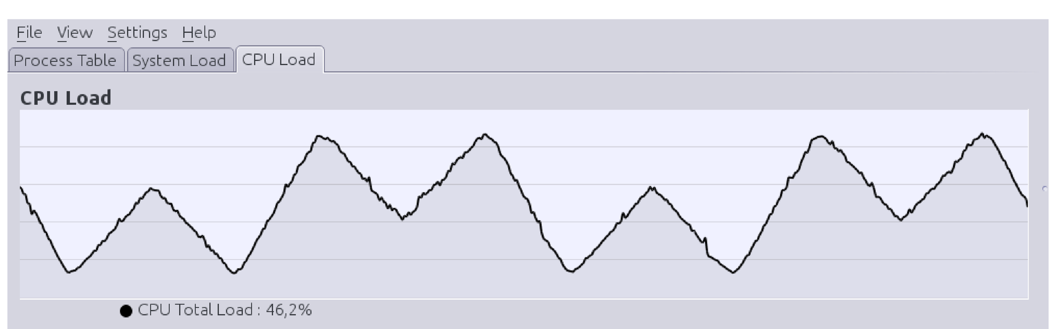

Figure 5: Showstopper following the load trace prepared in Figure 4 (in a loop, with linear interpolation), observed by the KDE load monitor (ksysguard) with a sampling interval of 500 ms.

Partial Load Experiments

Partial load experiments investigate the impact of CPU load on

- performance of virtual machines (VM) sharing a physical host,
- performance counters (cache and NUMA misses, CPI),
- power consumption, application performance, ...

We use the "**staircase**" load trace to avoid a separate warm-up period for each partial load experiment. We let the workloads warm up once and use Showstopper to follow a trace with 19 downward steps (95% to 5%) and 19 symmetrical upward steps (5% to 95%). This also provides a "reproducibility check", comparing the results for symmetrical steps with the same load. (The center of the gray band in Figure 6 represents the "staircase" trace.)

Two VMs with 16 CPUs share a physical server with 32 CPUs. The two VMs **do not compete for CPU time**. We intend to investigate other forms of **performance interference**. One VM runs 16 uncontrolled instances of h2. The other VM runs 16 instances of luindex controlled by **Showstopper** according to the "staircase" trace.

We would like to investigate

- How application throuhgput of luindex depends on the CPU load. Figure 6 shows a non-linear dependency.
- How application throughput of the uncontrolled h2 in one VM depends on the CPU load induced by luindex in the other VM.
- Whether it is better to pin the two VMs to separate NUMA nodes or to let them migrate freely. Figure 7 reveals that the performance of the VM running h2 in absence of pinning (—) is superior to the pinned performance (—) whenever the CPU load in the VM running luindex is lower than 80%.
- How CPU pinning of VMs and CPU load influence system power consumption. Figure 8 shows that the unpinned configuration (●) yields a higher power consumption than the pinned one (●) when the VM running luindex is lightly loaded.

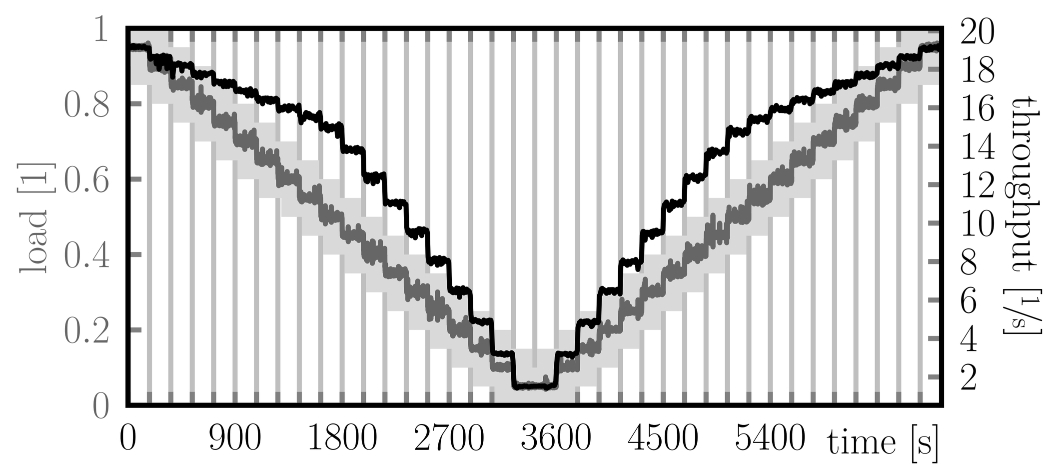

Figure 6: Observed CPU load (—) and throughput (—) of luindex when following the "staircase" trace using Showstopper. The light gray band represents the target load.

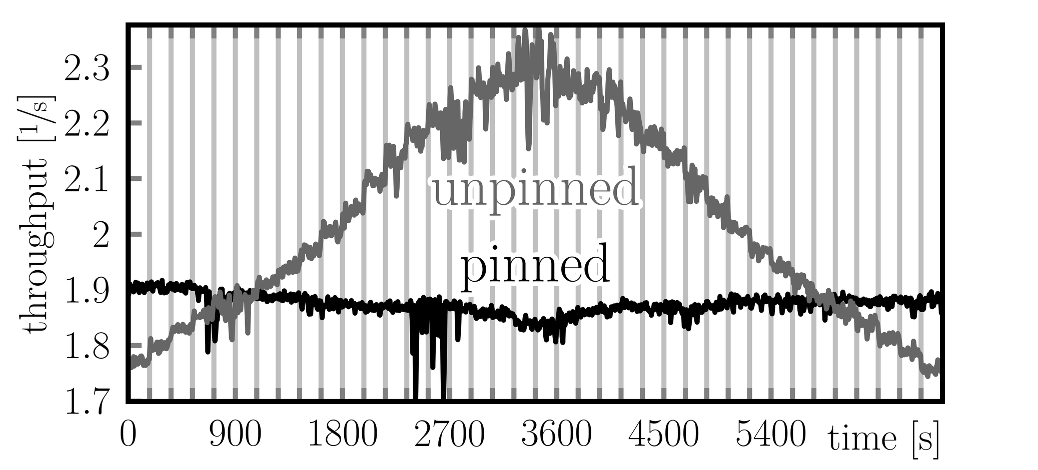

Figure 7: Throughput of uncontrolled h2 running at the same time as luindex in Figure 6 on a neighboring VM when the two VMs are pinned to separate NUMA nodes (—) or unpinned (—).

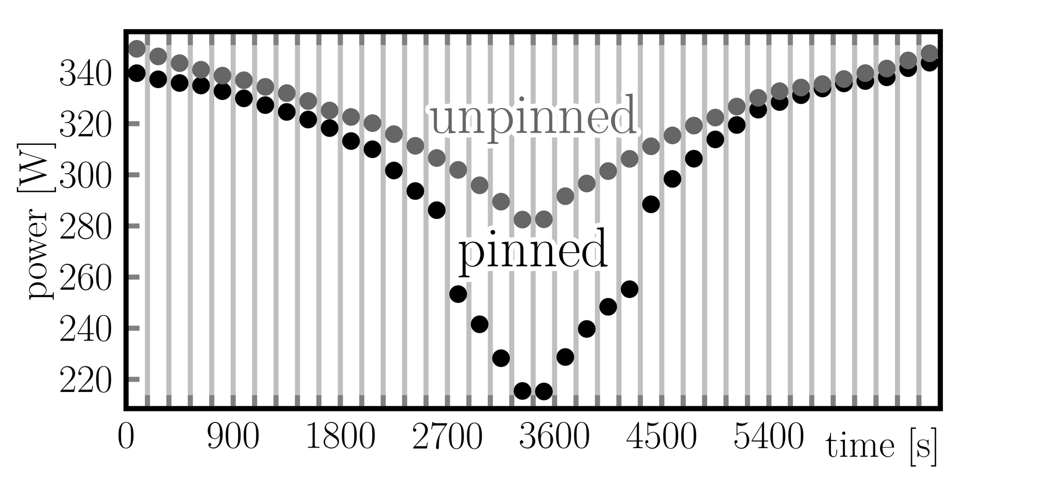

Figure 8: System power consumption whilst running uncontrolled h2 and controlled luindex (as shown in Figures 6 and 7). The two VMs are pinned to separate NUMA nodes (●) or unpinned (●).

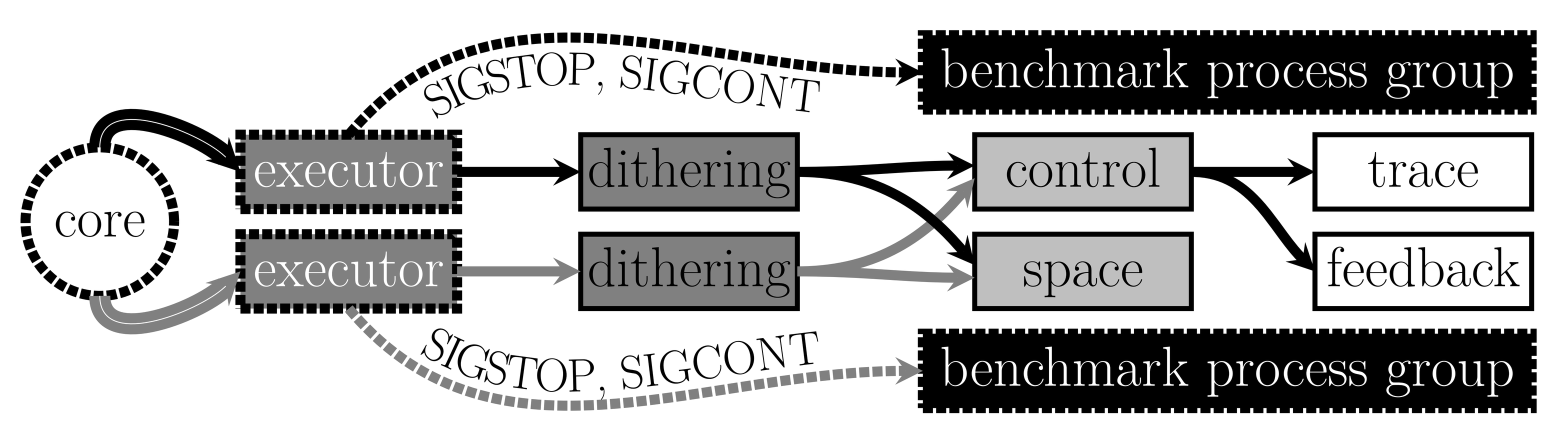

Figure 9: A Showstopper-driven benchmark controlling **two** workload instances.

Figure 9 displays the Showstopper architecture. **Dotted borders** represent processes. **Solid borders** represent Showstopper library modules. **Solid lines** indicate library calls within a process. **Dotted lines** indicate communication via POSIX signals. **Double lines** indicate the POSIX process hierarchy.

- A trace module determines the target CPU load based on a load trace.
- A feedback module obtains the actual CPU load from the operating system.
- A control module computes the average duty cycle, based on the inputs from trace and feedback.
- A space module decides how to distribute the average duty cycle among the parallel workloads.
- A dithering module decides how to distribute the duty cycle of a single workload over time.

There are **multiple implementations** of each module, providing many types of partial CPU load.