PARTIES: QoS-Aware Resource Partitioning for Multiple Interactive Services

Shuang Chen, Christina Delimitrou, Jose F. Martinez
ASPLOS 2019
Presented by: Nick from CoffeeBeforeArch

Overview

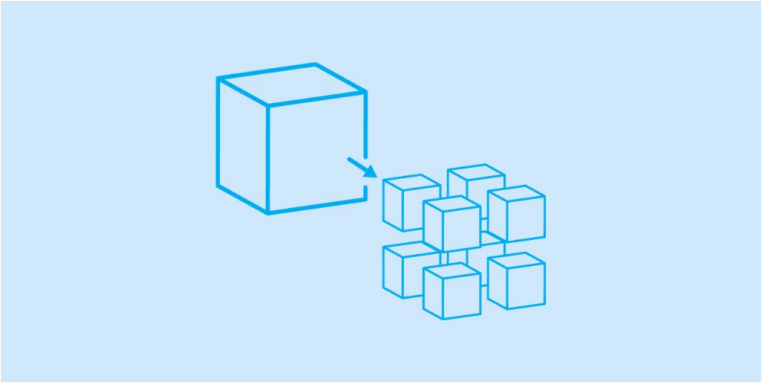
Overview

- Modern datacenters limit co-location of latency critical tasks
  - Limits efficiency
  - Increasing number of jobs are latency sensitive
- Solution?
  - QoS resource manager called PARTIES
  - Dynamically adjusted resources to meet all deadlines

Background

Background

- Methods for handling interference
  - Disallow sharing
  - Avoid co-sharing applications with likely interferences
  - Partitioning resources
- Application trends
  - Shifting from batch -> low latency services
  - Large application -> microservices



Characterization

Characterization (Applications)

- Memcached
  - Memory object caching system (in-memory key-value stores)
- Xapian
  - Web search engine
- NGINX
  - High performance HTTP server
- Moses
  - Statistical Machine Translation
- MongoDB
  - NoSQL database
- Sphinx
  - Speech recognition

Characterization (Applications)

Table 2. Latency-Critical Applications

Application	Memcached	Xapian	NGINX	Moses	MongoDB	Sphinx
Domain	Key-value store	Web search	Web server	Real-time translation	Persistent database	Speech recognition
Target QoS	600us	5ms	10ms	15ms	300ms	2.5s
Max Load under QoS	1,280,000	8,000	560,000	2,800	240	14
IPC	0.74	1.16	0.67	0.74	0.40	0.79
User / Sys / IO CPU%	13 / 78 / 0	42 / 23 / 0	20 / 50 / 0	50 / 14 / 0	0.3 / 0.2 / 57	85 / 0.6 / 0
Instr Cache MPKI	23.25	2.34	27.18	6.25	33.07	7.32
LLC MPKI	0.55	0.03	0.06	10.48	0.01	6.28
Memory Capacity (GB)	9.3	0.02	1.9	2.5	18	1.4
Memory Bandwidth (GB/s)	0.6	0.01	0.6	26	0.03	3.1
Disk Bandwidth (MB/s)	0	0	0	0	5.1	0
Network Bandwidth (Gbps)	3.0	0.07	6.2	0.001	0.01	0.001

Resources and Isolation Mechanisms

Resources and Isolation Mechanisms

Table 3. List of experimental setups for studying resource interference (left), and isolation mechanisms per resource (right).

Shared Resource	Method of Generating Interference	Isolation Mechanism	Software/Hardware Isolation Tool
Hyper-thread	8 compute-intensive microbenchmarks are colocated on the same hyperthreads as LC applications.	**Core Isolation**	Linux's *cpuset cgroups* is used to allocate specific core IDs to a given application.
CPU	8 compute-intensive microbenchmarks are colocated on the same physical cores as the LC applications, but different hyperthreads.		
Power	12 compute-intensive microbenchmarks (power viruses) are mapped on the 12 logical cores of the 6 idle CPUs.	**Power Isolation**	The ACPI frequency driver with the "userspace" governor to set selected cores to a fixed frequency (1.2-2.2GHz with 100MHz increments), or with the "performance" governor to run at turbo frequency (the highest possible frequency based on load, power consumption, CPU temperature, etc [15]).
LLC Capacity	We launch a cache-thrashing microbenchmark which continuously streams an array the size of the LLC. It runs on an idle core in the same socket as the LC application.	**LLC Isolation**	Intel's Cache Allocation Technology (CAT) [7, 32] is used for LLC way partitioning. It indirectly regulates memory bandwidth as well because memory traffic is highly correlated with cache hit rate. There is no mechanism available on the evaluated server platform to partition memory bandwidth directly.
LLC Bandwidth	We launch 12 cache-thrashing microbenchmarks whose aggregate footprint is the size of the LLC, i.e., each thrashes 1/12 of the LLC.		
Memory Bandwidth	We launch 12 memory-thrashing microbenchmarks, generating 50GB/s of memory traffic (upper limit on the evaluated machine).		
Memory Capacity	We launch one memory-thrashing microbenchmark that streams 120GB out of the 128GB memory (8GB is reserved for OS).	**Memory Isolation**	Linux's *memory cgroups* is used to limit the maximum amount of memory capacity per container.
Disk Bandwidth	We launch *dd* with *of=/dev/null*.	**Disk Isolation**	Linux's *blkio cgroups* is used to throttle the maximum disk read bandwidth per container.
Network Bandwidth	We launch one *iperf3* client on an idle core, and direct its traffic to an idle machine running the *iperf3* server using 100 connections, each at 100Mbps bandwidth.	**Network Isolation**	Linux's *qdisc* traffic control scheduler [14] with hierarchical token bucket (HTB) queueing discipline is used to limit the egress network bandwidth.

Application Limitations

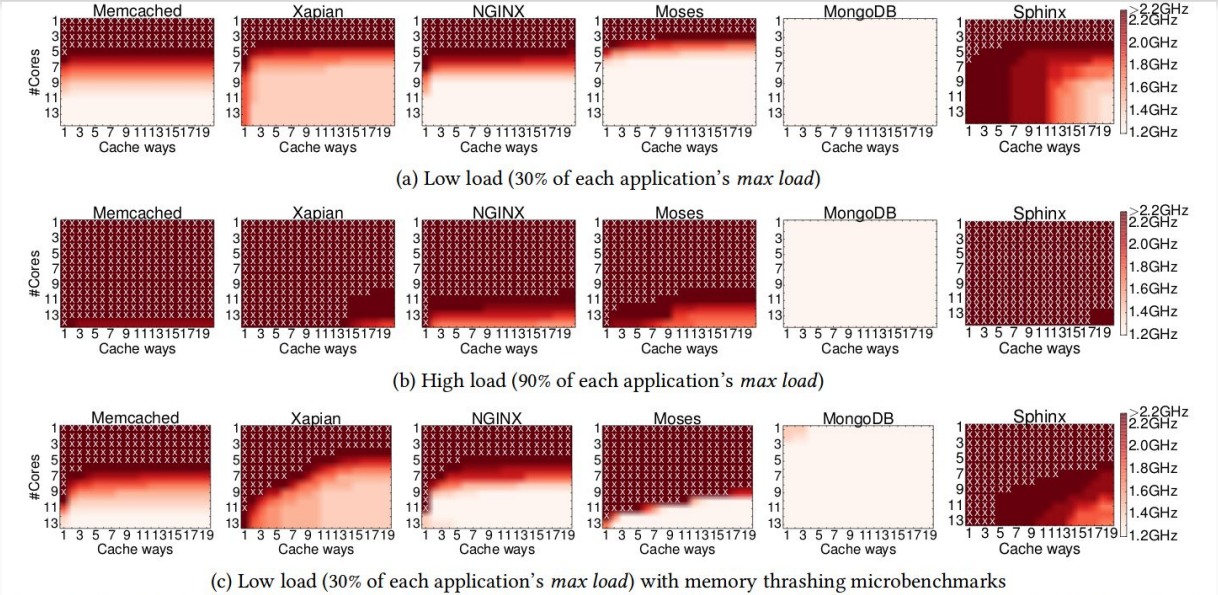
Application Limitations

Table 4. Impact of resource interference. Each row corresponds to one type of resource. Values in the table are the maximum percentage of *max load* for which the server can satisfy QoS when the LC application is running under interference. Cells with smaller numbers/darker colors mean that applications are more sensitive to that type of interference.

	Memcached	Xapian	NGINX	Moses	MongoDB	Sphinx
Hyperthread	0%	0%	0%	0%	90%	0%
CPU	10%	50%	60%	70%	100%	30%
Power	60%	80%	90%	90%	100%	70%
LLC Capacity	90%	90%	70%	80%	100%	30%
LLC Bandwidth	0%	60%	70%	30%	90%	0%
Memory Bandwidth	0%	50%	40%	10%	70%	0%
Memory Capacity	0%	100%	0%	0%	20%	80%
Disk Bandwidth	100%	100%	100%	100%	10%	100%
Network Bandwidth	20%	90%	10%	90%	90%	80%

Sensitivity Analysis (Cache Ways)

Sensitivity Analysis (Cache Ways)



(a) Low load (30% of each application's *max load*)

(b) High load (90% of each application's *max load*)

(c) Low load (30% of each application's *max load*) with memory thrashing microbenchmarks

Figure 2. Sensitivity to resource allocation under different loads and interference sources. Each column/row represents a fixed number of cache ways/cores assigned to an application using the core and LLC isolation mechanisms. Each cell represents the minimum frequency needed to meet QoS under a given number of cores and cache ways. The darker the color, the higher the required frequency. Cells with cross marks mean that QoS cannot be satisfied even at the highest possible frequency.

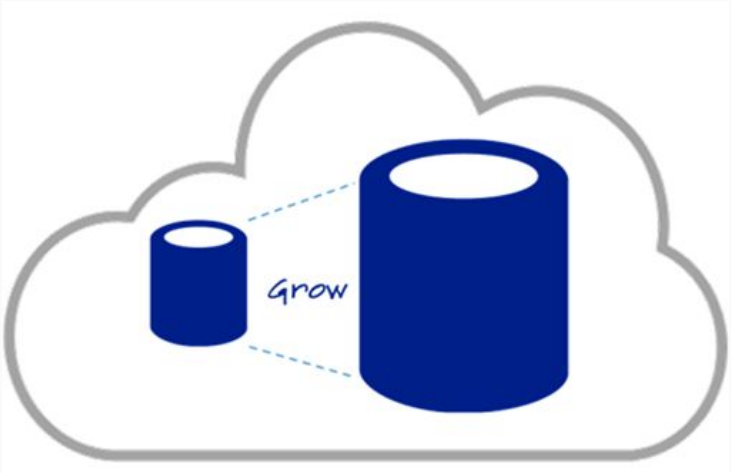
Design of PARTIES

Design of PARTIES

- Resource allocation is dynamic and fine-grained
- No a priori application knowledge or profiling
- Fast recovery from poor decisions
- Migration of tasks are a last resort

Design of PARTIES (Controller)

- Equal partition of managed resources to start
- Sampling every 500ms
  - Make adjustments based on tail latency slack
  - Upsize on low/negative slack
  - Reduce resources when comfortably meeting deadlines
- Timer to track how long QoS violation has been occuring



Design of PARTIES (Controller)

- Pick random resource to adjust first
  - If latency does not improve, move to next resource
  - Checks to memory bandwidth
    - Memory resource have near immediate results
    - Compute resource must drain buffers

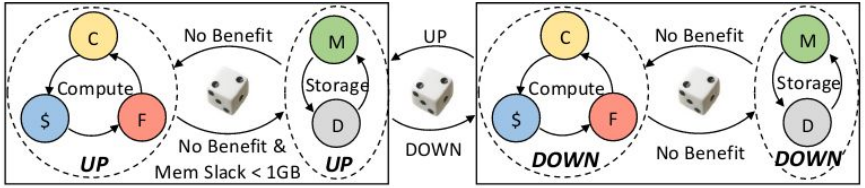


Figure 4. Transitions (*arrows*) between actions (*nodes*) in function *next_action*. For each UP or DOWN direction, tradeable resources are grouped into *trading wheels* (compute and storage). Transitions between wheels within the same direction happen when options for the current wheel have been exhausted. Transitions between directions (opposite sides in the figure) happen when the controller moves from *upsize* to *downsize* or vice versa.

Design of PARTIES (Controller)

- Selecting a victim
  - Avoid OOM errors
  - No ping-ponging between multiple up-sizing applications
- Selection based on highest tail latency slack
- Select non-QoS applications first (of course)

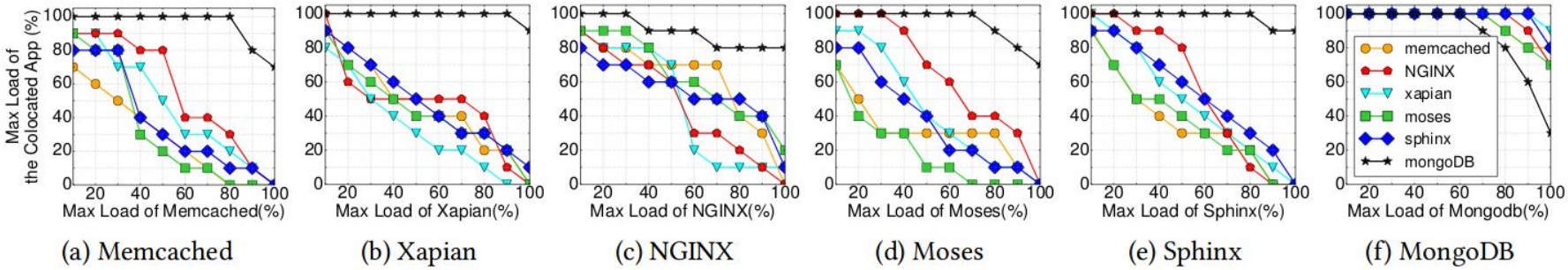
Misc. Questions

Characterization (Applications)

- What does PARTIES need to know?
- How to monitor latency?
- Determining controller parameters?
- Cost of upsizing on other applications?
- Job migration?
- Job scheduler influence?

Results

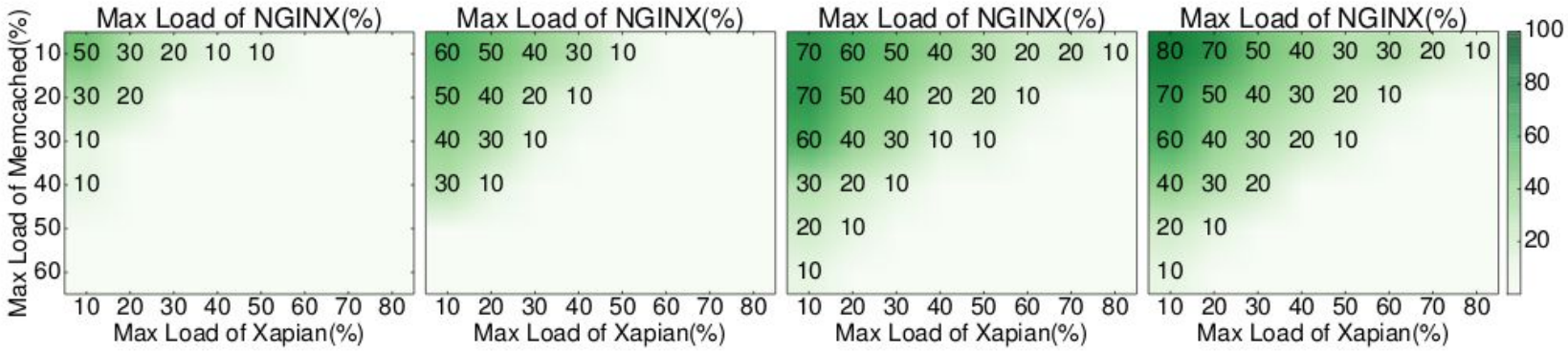
Results (Co-Location)



(a) Memcached (b) Xapian (c) NGINX (d) Moses (e) Sphinx (f) MongoDB

Figure 5. Colocation of 2 LC applications. Each plot shows the result of colocating one application (App_1) with each of the six studied applications (App_2). Each line shows the maximum percentage of App_2's *max load* (y-axis) that can be achieved without a QoS violation when App_1 is running at the fraction of its own *max load* indicated on the x-axis.

Results (SotA Comparison)

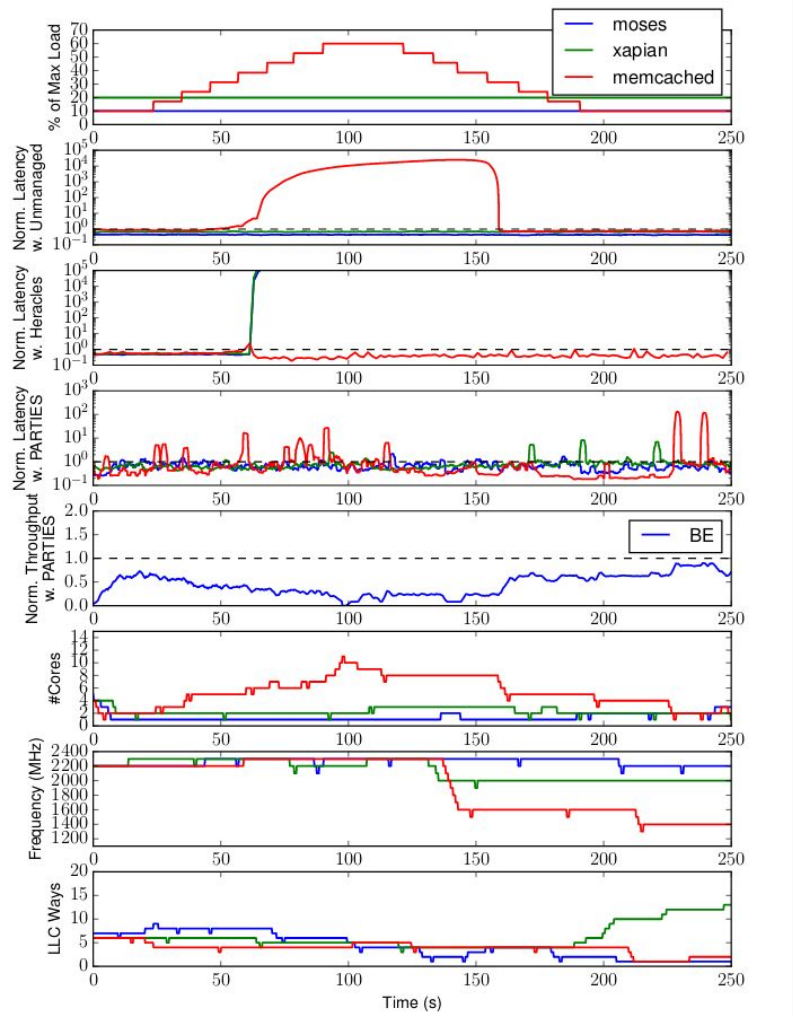


(a) Unmanaged (b) Heracles (c) PARTIES (d) Oracle

Figure 8. Colocation of Memcached (M), Xapian (X) and NGINX (N) with different resource managers. The values in the heatmaps are the max percentage of N's *max load* achieved without QoS violations when M and X run at the fraction of their *max loads* indicated in the y and x axes.

Resource Allocation over Time

- For Unmanaged, Heracles, and PARTIES
- Varying load
- 3 applications (Memcached, Moses, and Xapian)



Conclusion

Conclusion

- Outperforms SotA
- Allows multiple latency critical applications to be co-located
- Leverages current isolation mechanisms
- Settles for an acceptable solution, not best

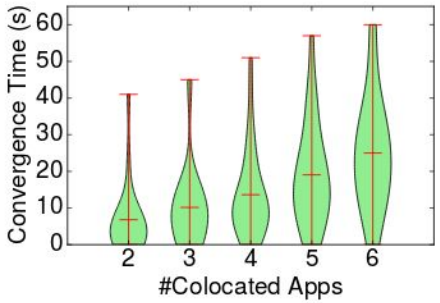


Figure 10. Violin plot of convergence time for 2 to 6-app mixes. Red markers show the min, mean, and max.