2017 4th International Conference on Information Science and Control Engineering

Locpart: A Latency Optimized Cache Partitioning for Cloud Data Centers

Hanfeng Qin

School of Computer Science and Technology
Huazhong University of Science and Technology
Wuhan, 430074, China
Email: hanfengtsin@hust.edu.cn

***Abstract*—As continual evolution of virtualization technology and cloud computing paradigm, modern data centers consolidate various application to run with less servers. However, co-locating applications is challenging, especially for mixed workloads of both latency-critical and batch programs. It is required to address the unpredicted performance and *quality-of-service* (QoS) due to contention for the shared *last-level caches* (SLLC). In this paper, we present a novel latency optimized cache partitioning policy, Locpart, which requires neither hardware modification nor complicated software design. Locpart prioritizes latency-critical services over batch workloads on generating cache partitioning schemes with conservative allocation to gurantee QoS. At the same time, Locpart seeks to balance the rest cache ways among the workloads to achieve a maximum global performance. Evaluation with workloads mixes of emerging latency-critical as well as conventional batch programs show that Locpart guarantees the QoS and improves the global system performance in parallel significantly.**

***Index Terms*—Cache Partitioning; Latency-Critical; Cache Allocation Technology;**

I. Introduction

As continual evolution of virtualization technology and cloud computing paradigm, modern data centers consolidate various applications with less servers. However, it is challenging to consolidate applications on the same CMP, especially for those user-facing services co-located with conventional batch workloads. One of the key problems attributes to the unpredicted performance and *quality-of-service* (QoS) due to contention for the shared last-level caches (LLC) [1], [2]. Typical user-facing services, including online search, speech recognition, and machine translation, require strict low tail latency, e.g., 99^{th} percentile latency no longer than milliseconds [3]. A common solution to guarantee QoS is to run these services solo, which limits the resource utilization of data centers. Prior studies [4], [5] show that the resource utilization in data center is quite low, e.g., an average utilization achieves only around 6% to 12% in Google's data center [6]. Such poor utilization challenges the consolidation of cloud workloads, and increases the total cost of ownership for IT service consequently.

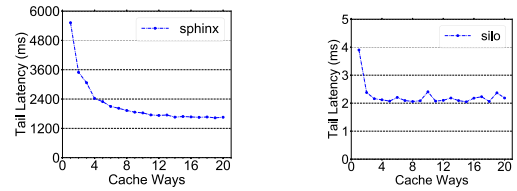
Prior efforts turn to restrict access to shared resources, e.g., partitioning cache explicitly [4], [7] among cloud workloads by restricting the available amount of shared cache lines that an applications can access when it co-runs with other workloads. Although it has been studied extensively during the past decades, cache partitioning has ever been an academic concept, and works only with simulators [7], [8], [9], [10]. Recently, it comes to reality as Intel introduces *cache allocation technology* (CAT) [11] in its latest Xeon processors. CAT enables software-controlled cache partitioning explicitly in program code to divide a high-associative LLC into several disjoint small-associative regions. However, CAT only provides a mechanism which enables user to manage caches among services. Neither further partitioning decisions nor schemes are offered by CAT. On the other hand, most prior works in cache partitioning on simulators depend on either complicated hardware modification [2], [7] or complicated software design [12], [13]. To make use of CAT in practice and simplify complicated operations on conducting cache performance tuning, it is required to exploit an intelligent policy.

In this paper, we present Locpart, a latency optimized cache partitioning policy cloud data center that requires no extra hardware modification or complicated software design. Locpart prioritizes latency-critical services over batch workloads on generating cache partitioning schemes with conservative allocation to guarantee QoS. At the same time, Locpart seeks to balance the rest cache ways among the workloads to achieve a maximum global performance. We implement Locpart on real product system equipped with Intel's CAT. Locpart consists of a performance monitor, cache partitioning engine, and cache partitioning driver. The performance monitor collects runtime statistics to activate the partitioning engine on demands. The cache partitioning engine executes the partitioning algorithm to generate cache allocation schemes. The cache partitioning driver enforces the schemes to be operated on the underlying caches with system calls. With many open source technologies, Locpart is built within around 1000 lines of C source codes. Evaluations with various kinds of latency-critical workloads show that Locpart guarantees the QoS of latency-and improves the global system performance as well for mixed workloads of both latency-critical and batch programs.

We highlight our key contributions as follows.

- We present a novel latency optimized cache partitioning policy, Locpart, which requires neither extra hardware modification nor complicated software support. Locpart prioritizes latency-critical workloads over batch programs

978-1-5386-3013-6/17 $31.00 © 2017 IEEE
DOI 10.1109/ICISCE.2017.97



(a) sphinx (LC) (b) silo (LC)

(c) img-dnn (LC) (d) mcf (BATCH)

Fig. 1. Sensitive of tail latency as well as performance to cache ways for latency-critical, and batch workloads, respectively

on generating cache partitioning schemes with conservative allocation to guarantee QoS. At the same time, it also seeks a maximum global performance by balancing the rest cache ways among workloads.

- We implement Locpart on real product system by making use of the Intel's cache allocation technology (CAT) and conduct comprehensive evaluation with mixed workloads consist of various emerging real latency-critical and batch programs. Experimental result show Locpart can guarantee the QoS while improving the global system performance.

II. Background and Motivation

A. Cloud Data Center Workloads

In terms of the diverse performance sensitivity with shared resources, cloud workloads can be classified into two categories. The first one are user-facing services which require immediately response on demands and are sensitive to latency, especially tail latency [3]. The second one are conventional batch workloads which run as a background services, and hardly interact with user frequently. Fig. 1 demonstrate how LLC affects the tail latencies of several typical user-facing services as well as performance of a batch program. To prevent the QoS loss from contention of shared resource, consequently, latency-critical services are usually deployed alone without co-runner, which results a poor resource utilization no more than 20% as mentioned in prior studies [4], [5]. Considering the huge costs of IT infrastructure, recent research efforts focus on controlling and assigning shared resource to mixed workloads is growing [4], [7].

B. Intel Cache Allocation Technology

Prior research efforts on cache partitioning either work with simulators [8], [9], [10] or require complicated modification to operating system [12], [14]. Practical cache partitioning on real system comes to reality until Intel introduces *cache allocation technology* (CAT) [11] in its Xeon processor E5

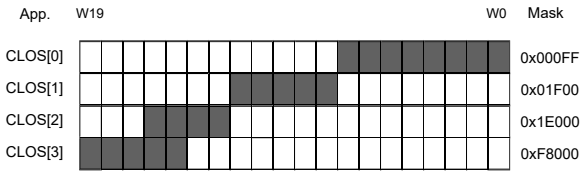


Fig. 2. Cache ways are partitioned among applications tagged as different CLOSes. Bit mask is used to enforce the partitioning scheme.

v3 product family. CAT enables software-controlled cache partitioning explicitly in program code by leveraging a way-based partitioning mechanism to divide a high-associative LLC into several disjoint small-associative regions where a program is restricted to allocate lines. Since CAT does not require any modifications to the operating system or kernel to take advantage of it, there have been some efforts to exploit such benefits [4], [15]. The main concepts in CAT is the use of *class of service* (CLOS) abstraction to prioritize resource allocation among programs. User can control the LLC lines a program can cover via binding a CLOS to each core and assigning a bit mask to the CLOS. Fig. 2 demonstrates how CAT partitions cache ways among applications in a 20-way associative cache. Four programs are tagged with different CLOSes, respectively. Each way is associated with a bit mask to indicate whether it is allowed to allocate on cache miss to the corresponding CLOS. For example, program tagged as `CLOS[0]` can allocate any lines in the 8 lower lines while program tagged as `CLOS[3]` can only allocate in the higher 4 lines.

Cache partitioning has been exploited to guarantee the QoS of user-facing services. However, prior work either require extra hardware modification [2], [7] or operating system redesign [13]. On the other hand, CAT does not provide any smart partitioning decisions yet. In this paper, we exploit Intel CAT to build a runtime system that supports a latency optimized cache partitioning for cloud data centers.

III. Latency Optimized Cache Partitioning

A. Problem Formulation

Above all we formulate this cache partitioning problem as follows. Suppose that we co-locate a latency-critical program with a batch program on a W-way associative LLC. Each program i is assigned W_i lines, then we have:

$$\sum W_i \leq W \tag{1}$$

We define $P_i(w)$ as performance of program i running with w cache ways in a interval of T. We normalize performance as follows. For batch programs, $P_i(w)$ is computed with the number of effective instructions[1] retired per interval that equals to $\frac{N_{inst}}{T}$. For latency-critical services, $P_i(w)$ is measured with

[1]We count effective instructions only for those contributing to useful task. For example, instructions used to synchronize multiple threads in multi-threaded programs are not considered.

throughput which is calculated as $\frac{N_{req}}{T}$ where N_{req} records the responding requests in the past interval of T. We have:

$$P_i(w) = \begin{cases} \frac{N_{inst}}{T}, & \text{for batch program} \\ \frac{N_{req}}{T}, & \text{for latency-critical program} \end{cases} \quad (2)$$

The global system performance G is defined as sum of both latency-critical service and batch program, similar as the effective machine utilization used in [4]. We have:

$$G = \sum P_i(w) = \frac{\sum N_{inst} + \sum N_{req}}{T} \quad (3)$$

We use sequence $L = \{L_i\}$ to denote the latencies of N_{req} requests, and tail latency TL is defined as a tail percentile of the *cumulative distribution function* (CDF) of L. Suppose the latency critical services are enforced with a QoS of L_{SLO} that is specified as*service level objectives* (SLO), then we have

$$TL \leq L_{slo} \quad (4)$$

For example, in Fig. 1(d), it is required that 99^{th} percentile of the response latency of an OLTP request is less than 2 milliseconds in `silo`.

B. Algorithm Design

We design an algorithm, *Locpart*, to solve the optimized objective function (3) as depicted in algorithm 1. Locpart computes a partitioning scheme denoted as cache line bitmasks of programs. However, such scheme is highly dependent on current running state and the dynamic demands on LLC. In our implementation on real product system, we activates this algorithm with some interval delays, which is configured with a user-programmable timer denoted as T_{sched}. We also facilitate Locpart with two helper procedures, `AllocLLC` and `GetTL`, which wrap the cache line allocation logic and the tail latency computation process, respectively.

We describe the workflow of Locpart as follows. Initially, each program is assigned with an equal portion of the available cache lines. Locpart works periodically to execute cache scheduling decision with an interval of T_{sched} (line 2–7). We first validate whether the allocation scheme violates the SLO (line 10–12). Locpart is designed with QoS as the first objective. We reclaim more lines from the batch programs, and allocate them to latency-critical services on SLO violation occurs. The cache line re-allocation is controlled with an user-specified step of w. If no violation are detected, we then consider to maximize the system performance via allocating lines to batch programs (line 13–15). To tolerate the runtime QoS variance, we do not allocate lines to batch programs immediately, instead we leverage an user-programmed threshold of $Util_{th}$ and trigger the allocation only with a higher QoS guarantee than $Util_{th}$. In our evaluation, the parameters of w and $Util_{th}$ are set to 1 and 0.95, respectively.

C. System Implementations

We implement Locpart on a real system equipped with Intel CAT running ArchLinux, a freely customized Linux distribution. Fig. 3 shows the fundamental functional modules.

Algorithm 1 Latency Optimized Cache Partitioning Algorithm

Require: CLOS Id for LC and batch program denoted as *LCId* and *BatId*; User defined LC tail latency SLO L_{SLO}; User defined batch cache allocation threshold $Util_{th}$; Cache re-allocaton step w; Tail latency percentile $TL_{percentile}$
Ensure: partitioning scheme denoted as a sequence of bitmask

```
1: function MAIN
2:     loop
3:         if need to schedule then
4:             Schedule()
5:             T_next ← T_next + T_sched
6:         end if
7:     end loop
8: end function
9: function SCHEDULE
10:    if (L_SLO − GetTL(TL_percentitle)) ≤ 0 then
11:        AllocLLC(LCId, w, true)
12:        AllocLLC(BatId, w, false)
13:    else if (L_SLO − GetTL(TL_percentitle))/L_SLO ≥ Util_th then
14:        AllocLLC(LCId, w, false)
15:        AllocLLC(BatId, w, true)
16:    end if
17: end function
18: function ALLOCLLC(CLOSId, w, Up)
19:    if Up = true then
20:        modify bitmask of CLOSId to increase w lines
21:    else
22:        modify bitmask of CLOSId to decrease w lines
23:    end if
24: end function
25: function GETTL(percentile)
26:    profiling latency distributions L of LC
27:    return CDF(L, percentile) ▷ CDF calculates cumulative distribution
28: end function
```

With many open source technologies, the Locpart is within around 1000 lines of C source codes. *Performance monitoring* component use an open source performance counter utility on x86 processors, *likwid*[2], to collect runtime statistics by periodically reading the *model specific registers* (MSRs). MSR access is arranged by a standard operating system driver, e.g., *msr* on our Linux platform. The user-space *cache partitioning engine* executes the above partitioning algorithm to generate a cache way allocation schemes for the running mixed workloads. The engine is also equipped with a user programmable timer that activates the partitioning process with pre-defined interval, the default value of which is configured as 500 ms. The engine enforces the schemes via invoking system calls provided by the *cache partitioning driver* to execute cache allocation instructions on the underlying LLC. The driver is designed based on an open source CAT toolkit, *pqos*[3], to partition LLC at user space. In principle, pqos enables programmer to control available cache ways via MSR on a hardware thread basis.

IV. Experimental Evaluations

A. Experimental Setup

We conduct our evaluation with a recently release Intel Xeon CPU E5-2630 v4, which supports CAT inherently. Our platform contains dual-socket such processors where each processor is equipped with 10 hyper-threading superscala cores of 2.2

[2]https://github.com/RRZE-HPC/likwid
[3]https://github.com/01org/intel-cmt-cat.git

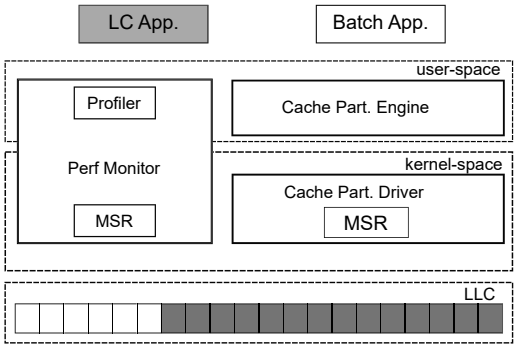


Fig. 3. Locpart implementation overview

Table I
CONFIGURATION PARAMETERS OF THE EXPERIMENTAL PLATFORM

Processor	dual-socket 10 Xeon E5-2630 v4 cores (Broadwell) hyper-threading, 2.2 GHz
L1 Caches	32 KB, 8-way set associative, private, split D/I
L2 Caches	256 KB, 8-way set associative, private, unified D/I
L3 Caches	25 MB, 20-way set associative, shared, unified D/I
Memory	64 GB, DDR3, 2133 MHz
OS	ArchLinux, kernel-4.10.8

Table II
WORKLOAD DESCRIPTION

Type	Benchmark	Description
LC	masstree	high performance in-memory key-value store
LC	sphinx	accurate speech recognition system
LC	xapian	web search engine running queries on Wikipedia
LC	silo	fast in-memory database used in OLTP
LC	img-dnn	handwriting recognition app. based on OpenCV
Batch	mcf	vehicle scheduling with a network simplex algo.

GHz. Each core has its dedicated private split 32 KB 8-way set associative L1 instruction and data caches, and a private 256 KB 8-way set associative L2 cache. The 10 cores within the same socket share 25 MB 20-way set associative last-level cache (LLC). Other detailed configurations are shown in Table I. Locpart activates partitioning engine every 50 ms. We select five latency-critical services from the Tailbench suite [16] as well as one batch application from SPEC CPU2006 suite [17]. All of these programs are compiled using GCC 6.3.1 with –O3 optimization flag.

B. Evaluation Results

Performance comparison: Firstly, we investigate the performance of executing latency-critical workloads mixed with batch program under three different kinds of scenarios, i.e., solo access to LLC as our baseline, uncontrolled contention for LLC with batch program, and restricted access to LLC protected with Locpart. Fig. 4 shows the corresponding results

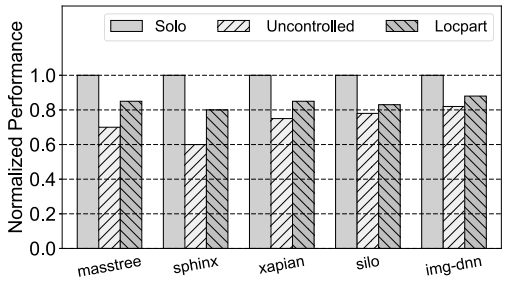


Fig. 4. Normalized aggregate performance of latency-critical workloads

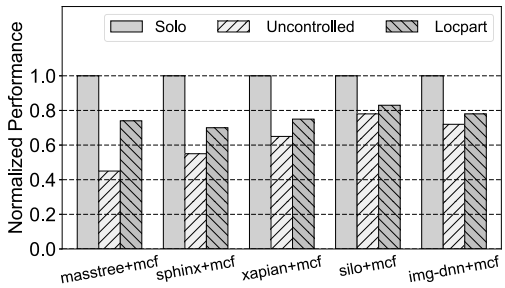


Fig. 5. Normalized global performance of mixed workloads

for each latency-critical program. All performance values are normalized to the one in baseline case, which limits the upper boundary of performance. We can see that without controlled access to LLC, the batch program `mcf` can hurt performance to 18% (of `img-dnn`) at least, and up to 40% (of `sphinx`) at most. Performance degradation attributes to the occupied cache ways by the batch competitor. Locpart prevents uncontrolled preemption from batch program with strict access control. Consequently, reduction in the performance loss is to 12% (of `img-dnn`) at least, and up to 17% (of `silo`). The average performance loss to the solo baseline is 15%.

Then, we evaluate the global performance of the mixed workloads, computed as the sum of both latency-critical and batch programs. Performance values are normalized to the solo baseline of both latency-critical workloads and batch program, respectively. As shown in Fig. 5, the global performance under uncontrolled contention case and Locpart cases are reduced by 37% and 24%. It reveals that co-locating applications unaware of cache contention degrades both performances. Even survived with Locpart, the performance loss is still significant.

Impact of Partitioning Intervals: Locpart generates partitioning schemes periodically with a pre-defined timer. Here we study how performance varies as we adjust the partitioning interval values. We report the normalized performance variation curves in Fig. 6 for two different kinds of latency-critical workloads. We observe significant performance loss in `sphinx` as we increase the partitioning interval. In contrast, performance disturbance is not serious in `xapian`. It can be explained with the different tail latency of programs. `sphinx` has tail latency longer than 1 seconds while `xapian` requires tail latency of

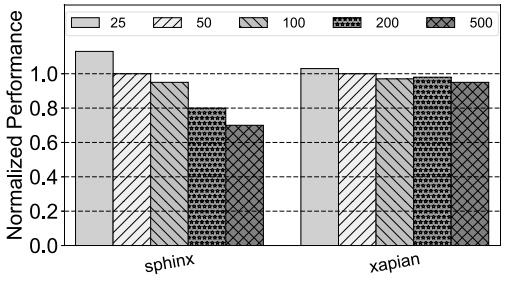


Fig. 6. Normalized performance with various partitioning intervals

several milliseconds. As a result, `sphinx` benefits from ways allocated on frequently partitioning. The rest latency-critical workloads share a similar latency of xapian, which do not benefits significantly from partitioning frequently.

V. Related Works

It is a hot topic to reduce tail latency of user-facing services in data center. Prior work investigated case studies of tail latency. For example, [18] explored the difference of tail latency between real network and analytical models with memcached. [5] analyzed the source of tail latency with workloads running in Google's warehouse data center. A reasonable approach to is to restrict accesses to shared resources explicitly among co-located services. [4] employed control theory to optimize Google's interactive service in case of contention for shared resources. Locpart does not leverage control theory due to its complexity in handling excessive type of shared resources. [7] designed a partitioning cache to address the transient behavior in latency-critical applications, which is orthogonal to our work.

VI. Conclusions

Co-locating latency-critical services with conventional batch workloads is challenge in cloud computing. In this paper, we have presented Locpart, a novel latency optimized cache partitioning policy for cloud data center. Locpart prioritizes latency-critical workloads over batch programs on generating cache partitioning schemes, and makes use of Intel's CAT to enforce the schemes. Evaluation on real product environment with various kinds of emerging latency-critical applications mixed with batch programs show that Locpart guarantees the QoS and improves the global system performance as well.

Acknowledgment

This work is supported by National Science Foundation of China under grant No. 61472150.

References

[1] L. Tang, J. Mars, N. Vachharajani, R. Hundt, and M. L. Soffa, "The impact of memory subsystem resource sharing on datacenter applications," in *Proceedings of the Annual ACM/IEEE International Symposium on Computer Architecture (ISCA)*. ACM, 2011, pp. 283–294.

[2] H. Yang, A. Breslow, J. Mars, and L. Tang, "Bubble-flux: Precise online qos management for increased utilization in warehouse scale computers," in *Proceedings of the Annual ACM/IEEE International Symposium on Computer Architecture (ISCA)*, 2013, pp. 607–618.

[3] J. Dean and L. A. Barroso, "The tail at scale," *Commun. ACM*, vol. 56, pp. 74–80, 2013.

[4] D. Lo, L. Cheng, R. Govindaraju, P. Ranganathan, and C. Kozyrakis, "Heracles: Improving resource efficiency at scale," in *Proceedings of the Annual ACM/IEEE International Symposium on Computer Architecture (ISCA)*, 2015, pp. 450–462.

[5] S. Kanev, J. P. Darago, K. Hazelwood, P. Ranganathan, T. Moseley, G.-Y. Wei, and D. Brooks, "Profiling a warehouse-scale computer," in *Proceedings of the Annual ACM/IEEE International Symposium on Computer Architecture (ISCA)*, 2015, pp. 158–169.

[6] D. Lo, L. Cheng, R. Govindaraju, L. A. Barroso, and C. Kozyrakis, "Towards energy proportionality for large-scale latency-critical workloads," in *Proceedings of the Annual ACM/IEEE International Symposium on Computer Architecture (ISCA)*, 2014, pp. 301–312.

[7] H. Kasture and D. Sanchez, "Ubik: Efficient cache sharing with strict QoS for latency-critical workloads," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2014, pp. 729–742.

[8] M. K. Qureshi and Y. N. Patt, "Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches," in *Proceedings of the Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2006, pp. 423–432.

[9] D. Sanchez and C. Kozyrakis, "Vantage: Scalable and efficient fine-grain cache partitioning," in *Proceedings of the Annual ACM/IEEE International Symposium on Computer Architecture (ISCA)*, 2011, pp. 57–68.

[10] R. Wang and L. Chen, "Futility scaling: High-associativity cache partitioning," in *Proceedings of the Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2014, pp. 356–367.

[11] A. Herdrich, E. Verplanke, P. Autee, R. Illikkal, C. Gianos, R. Singhal, and R. Iyer, "Cache qos: From concept to reality in the intel xeon processor e5-2600 v3 product family," in *Proceedings of the IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2016, pp. 657–668.

[12] Y. Ye, R. West, Z. Cheng, and Y. Li, "COLORIS: A dynamic cache partitioning system using page coloring," in *Proceedings of the International Conference on Parallel Achitectures and Compilation Techniques (PACT)*, 2014, pp. 381–392.

[13] A. Belay, G. Prekas, A. Klimovic, S. Grossman, C. Kozyrakis, and E. Bugnion, "Ix: A protected dataplane operating system for high throughputand low latency," in *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2014, pp. 49–65.

[14] J. Lin, Q. Lu, X. Ding, Z. Zhang, X. Zhang, and P. Sadayappan, "Gaining insights into multicore cache partitioning: Bridging the gap between simulation and real systems," in *Proceedings of the IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2008, pp. 367–378.

[15] L. Funaro, O. A. Ben-Yehuda, and A. Schuster, "Ginseng: Market-driven llc allocation," in *Proceedings of the USENIX Annual Technial Conference (ATC)*, 2016, pp. 295–308.

[16] H. Kasture and D. Sanchez, "Tailbench: a benchmark suite and evaluation methodology for latency-critical applications," in *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC)*, 2016, pp. 3–12.

[17] J. L. Henning, "SPEC CPU2006 benchmark descriptions," *ACM SIGARCH Comput. Archit. News*, vol. 34, no. 4, pp. 1–17, 2006.

[18] J. Li, N. K. Sharma, D. R. K. Ports, and S. D. Gribble, "Tales of the tail: Hardware, os, and application-level sources of tail latency," in *Proceedings of the ACM Symposium on Cloud Computing (SoCC)*, 2014, pp. 9:1–9:14.