OMBM: Optimized memory bandwidth management for ensuring QoS and high server utilization

Hanul Sung, Jeesoo Min, Sujin Ha, Hyeonsang Eom
Department of Computer Science and Engineering,
Seoul National University, Seoul, Republic of Korea
Email: husung@dcslab.snu.ac.kr, jsmin@dcslab.snu.ac.kr, sjha@dcslab.snu.ac.kr, hseom@cse.snu.ac.kr

***Abstract*—Latency-critical workloads such as web search engines, social networks and finance market applications are sensitive to tail latencies for meeting Service Level Objectives (SLOs). Since unexpected tail latencies are caused by sharing hardware resources with other co-executing workloads, a service provider executes the latency-critical workload alone. Thus, the data center for the latency-critical workloads has exceedingly low hardware resource utilization. For improving hardware resource utilization, the service provider has to co-locate the latency-critical workloads and other batch processing workloads. However, because memory bandwidth cannot be provided in isolation unlike cores and cache memory, the latency-critical workloads experience poor performance isolation even though the core and the cache memory are allocated in isolation to the workloads. To solve this problem, we propose an optimized memory bandwidth management approach for ensuring Quality of Service (QoS) and high server utilization. By providing isolated shared resources including memory bandwidth to the latency-critical workload and co-executing batch processing ones, our proposed approach guarantees SLOs and improves hardware resource utilization. Firstly, we predict the size of the memory bandwidth to meet the SLO for all Queries Per Seconds (QPSs) while executing the latency-critical workload with minimal pre-profiling. Then, our approach allocates the amount of the isolated memory bandwidth that guarantees the SLO to the latency-critical workload and the rest of the memory bandwidth to co-executing batch processing workloads. As a result, our proposed approach can achieve up to 99% SLO assurance and improve server utilization up to 6.5x.**

I. Introduction

Latency-critical workloads such as web search engines, social networks and finance market applications are sensitive to tail latencies for meeting the SLOs between a service provider and clients [1]. If hardware resources, CPUs, cache memory and a memory controller are shared with other batch processing workloads, the SLOs for the latency-critical ones cannot be met due to unexpected tail latencies [2, 3, 4]. Therefore, the service providers for the latency-critical workloads use the entire data centers only for them to meet the SLOs [5]. For this reason, the level of hardware resource utilization for the data center is 10% $\sim$ 45% and it is wasted with most of the resources unused [6, 7].

To address this problem, many researchers have conducted studies for not only guaranteeing SLOs but also improving the hardware resource utilization by co-locating the latency-critical workloads and the batch processing ones in academic and industry as well [5, 6, 8, 9]. In order to guarantee the SLOs of the latency-critical workloads, the interferences caused by sharing the hardware resources should be eliminated. Currently, the cores and the cache memory among the shared resources can be used, being physically isolated for each workload, but memory bandwidth is not yet provided in such a way, without the memory controller being physically isolated [10].

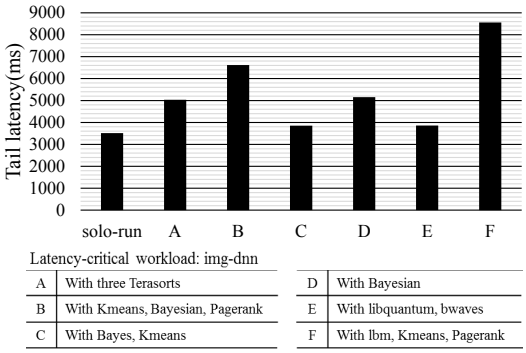


Fig. 1. Performance isolation result for a latency-critical workload

Figure 1 shows results of performance isolation for a latency-critical workload. The first bar in the figure shows a tail latency when the latency-critical workload is executed alone using the isolated core and cache memory. And the other bars indicate tail latencies when the workload is executed with other batch processing ones using the same amount of the isolated resources as for the first bar. The corresponding batch processing workloads are mentioned below the graph. All of the tail latencies for the co-executing batch processing workloads are different from that obtained when the latency-critical workload is executed alone despite the use of the isolated core and cache memory. In other words, the SLO of the latency-critical workload is not guaranteed due to these unexpected tail latencies [11, 12]. Because the batch processing workloads access a large amount of memory and worsen the contention for the memory controller, the tail latencies increase when the memory bandwidth is shared with co-executing batch processing workloads.

In order to address the problem, we propose an optimized memory bandwidth management approach for meeting SLOs and improving the hardware resource utilization without any runtime monitoring and profiling by assuming the contention for the shared memory bandwidth is the worst. The proposed approach increases the hardware resource utilization by co-

978-1-5090-6558-5/17 $31.00 © 2017 IEEE
DOI 10.1109/FAS-W.2017.158

locating the latency-critical workload and the batch processing ones. And it also meets SLOs by providing the isolated shared resources including the memory bandwidth to the latency-critical workload. To satisfy the SLOs of the latency-critical workload, pre-profilings are required for determining the memory bandwidth for the latency-critical workloads. However, there are two problems regarding this in practice. First, the QPSs for the latency-critical workloads are diverse [13] and the SLOs required by service providers also vary. Thus, it is not possible to figure out the memory bandwidth which guarantees various SLOs for all QPSs through pre-profilings. To reduce enormous pre-profilings, we perform the minimal pre-profilings and create two prediction graphs for predicting the amount of memory bandwidth to meet all SLOs for all QPSs. Second, the tail latencies are changed sensitively according to the memory bandwidth usage characteristics of the co-executing batch processing workloads. Thus, whenever the co-executing batch processing workloads are changed, the prediction lines have to be drawn newly for meeting the SLOs. This is also impossible in practice.

To address the problems, in our proposed approach the prediction lines are mad on the assumption that the contention for the shared memory bandwidth is worst.

By applying the proposed approach, the service provider may meet up to 99% SLOs of the latency-critical workload and improve the server resource utilization by up to 6.5x.

II. Background

A. Memory bandwidth isolation

Unlike the cores and the cache memory, there is no technique which divides the memory bandwidth physically. Because of this, our approach utilizes a memory bandwidth reservation system called memguard [10] which is implemented on the operating system for providing the isolated memory bandwidth. To measure the memory bandwidth usage for each core, the system monitors the LLC miss performance counters. If the core overuses the memory bandwidth than the reserved size, the system asks the operating system scheduler to help the core not use the memory bandwidth anymore. The operating system scheduler dequeues all of workloads on the core and prevents them from using the memory bandwidth until the next time slice. The system cannot provide physically isolated memory bandwidth to each workload, but it produces similar effects.

III. OMBM Design

A. OMBM Overview

To address the above-mentioned problems, we propose an optimized memory bandwidth management approach called *OMBM* for meeting SLOs and improving the hardware resource utilization. Figure 2 illustrates an overview of *OMBM*. The *OMBM* is divided into two components, 1)*predictor* and 2)*allocator*. The *predictor* performs pre-profilings for deciding the memory bandwidth to be allocated to the latency-critical workload with the least sampled QPSs. Based on the results of pre-profiling, the *predictor* creates two prediction graphs

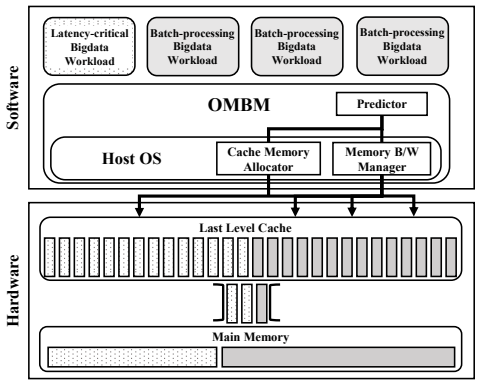


Fig. 2. Proposed approach architecture

which predict the memory bandwidth which can achieve the desired tail latency for all QPSs. Through the prediction graphs, the *predictor* reports the amount of the memory bandwidth which meets the SLOs, to the *allocator*. The *allocator* is informed of the memory bandwidth by the *predictor*, and allocates it to the latency-critical workload and the rest to the batch processing ones.

B. Predictor

For guaranteeing SLOs and improving the hardware resource utilization, the service provider needs to know the amount of memory bandwidth to be given to the latency-critical workloads and the batch processing ones in advance. Due to the lack of a hardware support, the memory allocation technique used in this paper is implemented with CPU scheduling. Thus, the memory bandwidth is not allocated completely physically isolated to each workloads, even if the same amount of memory bandwidth is allocated to the same latency-critical workload, the performance of executing alone is different from that of executing with other batch processing workloads. And because the contention for memory bandwidth varies depending on the memory bandwidth usage characteristics of co-executing batch processing workloads, the performance is different depending on the co-executing batch processing workloads. In addition, tail latencies have effects different from memory bandwidth interference, according to QPSs. For these reasons, a huge amount of pre-profiling is needed to prepare for all concurrent executions with any batch processing workloads. However, this is difficult in practice.

To address this problem, we propose two ideas. First, in order to meet SLOs strictly, we assume the worst case where the contention for memory bandwidth is the greatest and predict the memory bandwidth for the latency-critical workload for all QPSs. To create the extreme contention for memory bandwidth, STREAM benchmarks [14] causing the tremendous contention for memory bandwidth are executed on all cores except for the core where the latency-critical workload is run. And then we perform pre-profilings in this situation. Second, to avoid performing pre-profilings for all QPSs, the *predictor* selects the QPSs to pre-profile with a

Algorithm 1 Pseudo-code for first prediction graph

Input: $BWS = \{bw_1 \sim bw_n\}$: Memory Bandwidth
DQX : Max QPS value of target latency-critical workload
DQN : Min QPS value of target latency-critical workload
ETV : Exit threshold value of pre-profiling
Output: $FPG = \{fpl_1 \sim fpl_n\}$: First prediction graph
Data: MAX, MIN, MED : Max, min and medium QPSs pre-profiling range
CBW : Current allocated bandwidth to the target latency-critical workload
IC_{XD}, IC_{DN} : Margin of increase values from MAX to MED and from MED to MIN
$TL = \{tl_{DQN} \sim tl_{DQX}\}$: Tail latency

1: **for** $i \leftarrow 0$ to n
2: $CBW \leftarrow bw_i$
3: $MAX \leftarrow DQX$, $MIN \leftarrow DQN$
4: **while**
5: $MED \leftarrow \frac{MAX-MIN}{2}$
6: $tl_{MAX} \leftarrow$ Tail latency of max QPS generated by pre-profiling
7: $tl_{MIN} \leftarrow$ Tail latency of min QPS generated by pre-profiling
8: $tl_{MED} \leftarrow$ Tail latency of medium QPS generated by pre-profiling
9: $TL \leftarrow TL \cup tl_{MAX} \cup tl_{MIN} \cup tl_{MED}$
10: $IC_{XD} \leftarrow tl_{MAX} \div tl_{MED}$
11: $IC_{DN} \leftarrow tl_{MED} \div tl_{MIN}$
12: **if**$(IC_{XD} > IC_{DN})$
13: $MIN \leftarrow MED$
14: **else**
15: $MAX \leftarrow MED$
16: **if**$(MAX - MIN < ETV)$
17: break
18: Creating fpl_i by connecting all of tail latencies of TL
19: $FPG \leftarrow FPG \cup fpl_i$

divide and conquer method and perform pre-profilings with these sampled QPSs. Then the *predictor* creates the two prediction graphs consisting of multiple prediction lines based on results of the pre-profiling. The graphs show the predicted amounts of memory bandwidth to meet SLOs for all QPSs. The first graph is created in advance and the second one is created based on the first graph at runtime. The first graph shows the predicted tail latencies corresponding to all QPSs by adjusting the allocated memory bandwidth, and the second one shows the predicted tail latencies according to the memory bandwidth at specific QPSs.

In order to create the first prediction graph, we use the characteristics of the latency-critical workloads. The tail latencies of the latency-critical workloads remain constant for a while and then increase rapidly as the QPS increases. For this reason, the number of QPSs to be pre-profiled can be minimized. In addition, we create the prediction line by connecting the results of pre-profiling in a straight line. Because the tail latencies increase on an exponential scale, the actual tail latencies can be lower than the predicted line and the *predictor* can predict the amount of memory bandwidth for meeting SLOs. The second graph is continually updated whenever current QPS changes based on the first prediction graph at runtime.

Two algorithms for creating prediction graphs are described in Algorithms 1 and 2. Algorithm 1 takes four inputs: (1) memory bandwidth for pre-profiling $BWS = \{bws_1 \sim bws_n\}$, (2) maximum QPS value provided by a target latency-critical workload *DQX*, (3) minimum QPS value provided by the target latency-critical workload *DQN*, and (4) threshold value for finishing pre-profiling *ETV*. And it produces an output: the first prediction graph consisting of prediction lines corresponding on the different amounts of memory bandwidth $FPG = \{fpl_1 \sim fpl_n\}$. And it has seven data items: (1) tail latency generated by pre-profiling $TL = \{tl_{DQN} \sim tl_{DQX}\}$, (2) maximum value of range to be pre-profiled MAX, (3) medium value of range to be pre-profiled MED, (4) current allocated bandwidth to the target latency-critical workload CBW, (5) minimum value of range to be pre-profiled MIN, (6) margin of increase from MAX to MED IC_{XD}, and (7) margin of increase from MED to MIN IC_{DN}. At first, the *predictor* starts to create a prediction line constituting the first prediction graph from the smallest amount of memory bandwidth (lines 1 $\sim$ 2). To select the QPSs to be pre-profiled using a binary searching algorithm among the divide and conquer methods, MAX is set to DQX and MIN is set to DQN (line 3). The median of the two values is calculated and put it into the MED, and then the pre-profiling is started with the three values (lines 5 $\sim$ 8). In order to identify the rapidly increasing range accurately, the *predictor* calculates two margins of increase of tail latencies, from MAX to MED and from MED to MIN (lines 10 $\sim$ 11). In order to prepare for the next pre-profiling step, the *predictor* changes MAX or MIN. If IC_{XD} is higher than IC_{DN}, MIN is set to MED. If not, MAX is set to MED (lines 12 $\sim$ 15). In this way, the *predictor* defines the new range and starts pre-profiling. However, if the interval between the MAX and MIN is smaller than ETV, the *predictor* stops pre-profiling (lines 16 $\sim$ 17). When the progressive pre-profiling in the divide and conquer manner is finished, the *predictor* creates a prediction line by connecting tail latencies obtained by pre-profilings (line 18). The *predictor* updates the first prediction graph by repeating the process, and the graph shows the predicted tail latencies according to memory bandwidth for all QPSs (line 19).

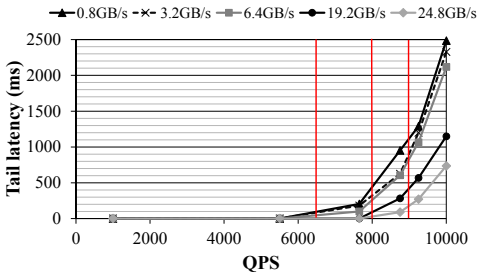


Fig. 3. First prediction graph

Figure 3 shows the first prediction graph. We provide five BWS, 0.8GB/s, 3.2GB/s, 6.4GB/s, 19.2GB/s and 24.8GB/s, thus n is 5. And we set DQX and DQN to 10000 and 1000. In addition, we define a minimum interval to be profiled as 10% and then the ETV is set to 90 through $(DQX - DQN) \times 0.1$. As shown in the figure, firstly MAX, MIN and MED is set to 10000, 1000, 5500 and pre-profiling is executed. Then the *predictor* defines the range which changes more increase rapidly as the new range for next pre-profiling, thus MIN is set to 5500 and MED is set to 7750. The prediction lines are created by repeating pre-profiling and merge them into the first prediction graph.

Algorithm 2 Pseudo-code for the second prediction graph

Input: $FPG = \{fpl_i \sim fpl_n\}$: First prediction graph
$CQPS$: Current QPS value
Output: $SPG = \{spl_{DQN} \sim spl_{DQX}\}$: Second prediction graph
Data: $ETL = \{etl_1 \sim etl_n\}$: Tail latency extracted from FPG

1: $spl_{CQPS} \leftarrow \emptyset$
2: **if**$(SPG \cap spl_{CQPS} == \emptyset)$
3: **for** $i \leftarrow 0$ to n
4: $etl_i \leftarrow$ extract tail latency of $CQPS$ in fpl_i
5: $ETL \leftarrow ETL \cup etl_i$
6: Creating spl_{CQPS} by connecting all of tail latency in ETL
7: $SPG \leftarrow SPG \cup spl_{CQPS}$
8: **else**
9: $spl_{CQPS} \leftarrow extract\ the\ predict\ line\ of\ CQPS\ from\ SPG$
10: $MB\_size \leftarrow$ Predicted memory bandwidth to meet SLO using spl_{CQPS}
11: *MB_Allocator(MB_size)*

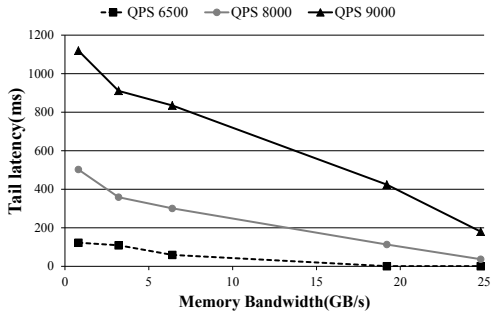


Fig. 4. Second prediction graph

The algorithm 2 describes the procedures for creating the second prediction graph based on the first prediction graph. The second prediction graph consists of multiple prediction lines which predict the tail latencies according to the amount of the memory bandwidth for all QPS. The algorithm has two inputs: (1) the first prediction graph $FPG = \{fpl_1 \sim fpl_n\}$, and (2) current QPS of running the latency-critical workload $CQPS$. And it generates an output: the second prediction graph $SPG = \{spl_{DQN} \sim spl_{DQX}\}$. Also it has a data item: tail latencies extracted from the first prediction graph $ETL = \{etl_1 \sim etl_n\}$. First, the *predictor* detects the current QPS of the latency-critical workload and checks the second prediction line at the current QPS, spl_{CQS} is in the second prediction graph, SPG (line 2). If it is in the graph, the line is extracted from the graph (line 9). If the prediction line is not in there, every tail latency is extracted according to all of memory bandwidth at the current QPS from the first prediction line (lines 3 $\sim$ 5). Then the *predictor* connects all of the tail latencies in ETL for creating the prediction line for the current QPS and merge it into the second prediciton graph (lines 6 $\sim$ 7). Finally, the *predictor* predicts the memory bandwidth to meet the SLO using SPG and reports the predicted result to *allocator* (lines 10 $\sim$ 11).

Figure 4 shows the second prediction graph which is created based on red lines of figure 3. Using the graph, we predict the memory bandwidth as 14GB/s for meeting an SLO, 200ms at 8000 QPS.

TABLE I
DESCRIPTION OF WORKLOADS

Benchmark	Workload	Description
TailBench	masstree	In-memory key-value store
	img-dnn	Handwriting recognition application
	silo	In-memory transactional database
HiBench	kmeans	Kmeans clustering in machine learning
	bayes	Bayesian Classification in machine learning
	pagerank	Link analysis in web search engine
SPECcpu2006	libquantum	Physics / Quantum Computing
	lbm	Computational Fluid Dynmaics, Lattice Boltzmann Method
	bwaves	Computational Fluid Dynamics

C. Allocator

To provide isolated memory bandwidth for the latency-critical workloads and the batch processing workloads, the *allocator* divides the memory bandwidth into two based on the memory bandwidth transferred from the *predictor*. And it provides the amount of the isolated memory bandwidth received from the *predictor* to the latency-critical workload. And the rest of the memory bandwidth is allocated to the batch processing workloads.

We divide the cache memory in a software manner, a page coloring, for providing isolated cache memory. The cache *allocator* divides the cache memory in half and provides them to the latency-critical workloads and the batch processing ones.

IV. EXPERIMENTAL EVALUATION

In this section, we show the accuracy of the prediction graphs which *OMBM* makes, SLOs guarantee and hardware resource usage improvement.

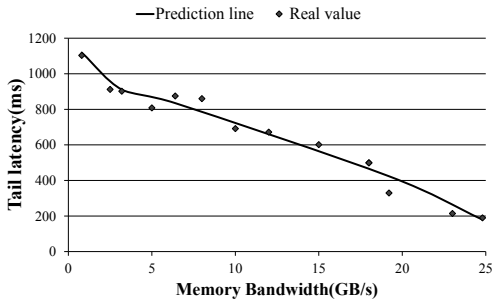
A. Experiment Environment and Benchmark Workloads

In this research, we performed experiments on an Intel Haswell server with a quad-core Intel i7-4770k 3.5GHz processor, 8MB 16-way set associative cache memory and 24.8GB/s memory bandwidth and used Linux 3.14.15 kernel and Hadoop 2.6. We used a Tailbench benchmark [13] developed by MIT as the latency-critical workloads and a HiBench benchmark [15], a representative of map-reduce workloads and SPECcpu2006 which has characteristics to be a good candidate for big data processing benchmarks [16], as the batch processing workloads. Table I shows some details of these workloads.

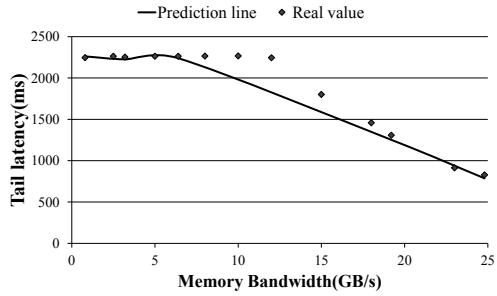
B. Prediction Accuracy

In order to predict the amounts of memory bandwidth to meet SLOs for all QPSs, our approach creates the prediction graphs with minimal pre-profilings using the divide and conquer manner. For meeting SLOs strictly, the real tail latencies under the same contention for memory bandwidth as the prediction line is created need to be close to the prediction line.

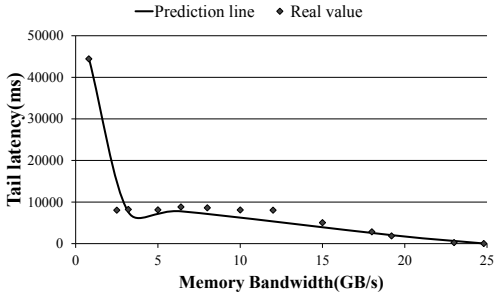
Figure 5 shows the prediction lines of the second prediction graph at a specific QPS and the real tail latencies by adjusting the allocated memory bandwidth. We used silo, masstree and img-dnn of the Tailbench benchmark as the latency-critical



(a) silo



(b) masstree



(c) img-dnn

Fig. 5. Prediction accuracy

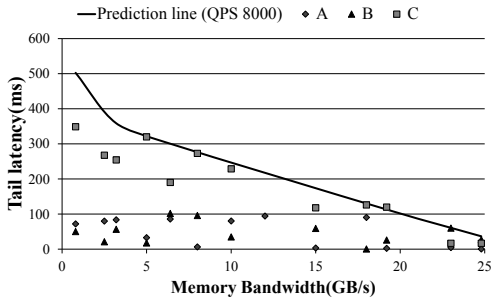
TABLE II
WORKLOAD COMBINATION USED IN EXPERIMENTS

Combination	Workloads
A	Kmeans, Bayesian, Pagerank
B	Kmeans, libquantum, bwaves
C	Kmeans, Pagerank, lbm

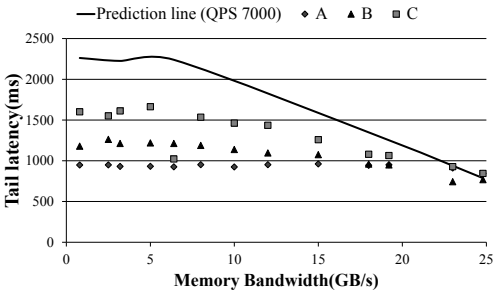
workloads in figure 5(a), 5(b) and 5(c). Even if the workloads have different sensitivities to the memory bandwidth, most of the tail latencies are close to the prediction lines. The prediction accuracy is at least 76%, up to 99% and 92% on average with only thirty pre-profilings. If the amount of the memory bandwidth profiled is increased, the predict accuracy will be improved.

C. SLOs Guarantee and Server Utilization Improvement

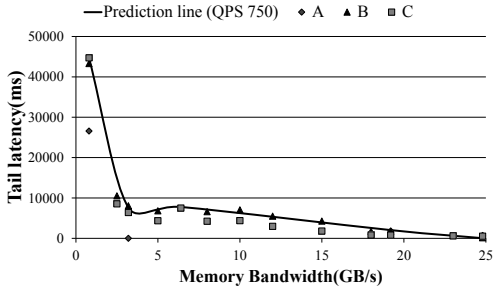
In order to show the effectiveness of our proposed approach, we demonstrate that the actual tail latencies with executing the batch processing workloads are lower than the prediction lines and show the hardware resource utilizations measured with the our approach or not. Since the prediction lines are created in



(a) silo



(b) masstree



(c) img-dnn

Fig. 6. SLOs guarantee running with batch processing workloads

the extreme contention for memory bandwidth, even if the latency-critical workloads execute with any batch processing ones, the actual tail latencies must be equal to or lower than the predicted values.

The actual tail latencies and the prediction lines created by our proposed approach are shown in Figure 6. In the figure, the latency-critical workloads, silo, masstree and img-dnn, are executed with co-executing combinations of the various batch processing workloads as described in Table II with different QPSs.

In Figure 6(a), as the prediction values of the combination C are close to the prediction line, we infer that C makes the memory contention similar to the one in the worst case.

As shown in Figure 6(b), differences between prediction values and actual tail latencies are different according the batch-processing workload combination. Since the lack of the hardware isolation supports, we assign the same amount of the memory bandwidth to masstree with software-based isolation technique though, the influences of masstree on the memory contention are different in three combination.

TABLE III
CPU UTILIZATION

	Combination	Latency-critical workload alone (%)	With three batch workloads (%)
Fig.6.(a)	A	17.77	98.57
	B	17.90	73.75
	C	15.07	98.90
Fig.6.(b)	A	24.82	99.57
	B	24.15	98.00
	C	15.52	96.70
Fig.6.(c)	A	17.77	98.57
	B	13.45	68.47
	C	18.92	92.00

Unlike Figure 6(b), Figure 6(c) shows similar values between the actual tail latencies and the prediction line because img-dnn is sensitive to the memory contention. Although the difference between the predicted values and the actual ones is different depending on the type of the co-executing batch processing workloads and QPS, all actual values are close to and smaller than the prediction line.

Table III shows the hardware resource utilizations with and without the our approach in the above situations. The hardware resource utilization is the CPU usage of the server node. As shown in the table, we found out that the hardware resource utilization is improved by about 5.1x on average and up to 6.5x, with C as in Figure 6(a).

V. Related Work

As the age for big data and cloud service has advanced, data centers have been expanded to a large scale. However, it has been incomplete for meeting SLOs and providing effective server management until now [17]. For this reason, many studies have been conducted such as resource isolation approaches [5, 12, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29] and interference-aware management schemes [3, 4, 30, 31, 32, 33, 34, 35, 36].

Kasture et al. proposed a cache memory isolation scheme which allocates the isolated cache memory to latency-critical workloads and batch processing ones [5]. For producing optimal performance, the scheme predicts the amount of the cache memory for the latency-critical workloads by monitoring the cache memory usage. Zhu et al. also proposed a cache memory isolation scheme in a similar way with Kasture et al. [5, 8]. These systems need additional hardware supports for calculating the proper the cache memory size and consider only the cache memory among shared resources. Therefore if interferences caused by the shared memory bandwidth occur, they cannot meet SLOs strictly. Unlike the above-mentioned approaches, Lo et al. took an isolation approach for most shared resources, cores, cache memory, power and a network [9]. For meeting SLOs, the system monitors QPSs persistently at runtime. If monitored QPSs are close to the SLO, then the system adjusts the amount of the shared resources.

Mars et al. investigated performance interferences between a latency-critical workload and a batch processing one by co-locating them [3, 32]. The systems measure QoSs by adjusting memory sub-system contentions and detect the memory sub-system usages used by the batch processing workloads. With the measured results, the systems decide the batch processing workload which executes with the latency-critical workload. Because the systems require all profiled results of the batch processing workloads, whenever new batch processing ones start executing, the systems also start profiling. Delimitrou et al. performed pre-profilings with various server configurations for finding a server to meet SLOs [34]. The system can figure out the proper server node. However since the system does not consider interferences generated by co-executing workloads, if the latency-critical workload executes with other workloads, the system experiences unexpected tail latencies.

Unlike existing upper studies, to the best of our knowledge, we always guarantee SLOs strictly without additional runtime monitoring and profiling with executing any batch-processing workloads.

VI. Conclusion

Since the memory bandwidth isolation techniques have not existed until now, the latency-critical workloads cannot have expected tail latencies even though the workloads use isolated the core and the cache memory. To address this problem, we have proposed an optimized memory bandwidth management approach for ensuring QoS and high server utilization. The approach is composed of two components, a *predictor* and a *allocator*. The *predictor* creates two the prediction graphs which predict tail latencies according to the memory bandwidth for all QPSs and the *allocator* provides isolated shared resources to the latency-critical workload and the batch processing ones. By utilizing this approach, the service provider can meet SLOs of the latency-critical workloads and improve the server hardware resource utilization.

Acknowledgment

This research was supported by 1)Institute for the Information & communications Technology Promotion (IITP) Grant funded by the Korea government (MSIP) (R0190-16-2012, High Performance Big Data Analytics Platform Performance Acceleration Technologies Development). And it was also partly supported by 2)the National Research Foundation (NRF) grant (NRF-2016M3C4A7952587, PF Class Heterogeneous High Performance Computer Development), 3)the National Research Foundation of Korea(NRF) grant funded by the Korea government(MSIP) (NRF-2017R1A2B4004513, Optimizing GPGPU virtualization in multi GPGPU environments through kernels' concurrent execution-aware scheduling) and 4)BK21 Plus for Pioneers in Innovative Computing (Dept. of Computer Science and Engineering, SNU) funded by National Research Foundation of Korea(NRF) (21A20151113068).

References

[1] T. Zhu, A. Tumanov, M. A. Kozuch, M. Harchol-Balter, and G. R. Ganger, "Prioritymeister: Tail latency qos for shared networked storage," in *Proceedings of the ACM Symposium on Cloud Computing*, ser. SOCC '14. New York, NY, USA: ACM, 2014, pp. 29:1–29:14. [Online]. Available: http://doi.acm.org/10.1145/2670979.2671008

[2] S. Govindan, J. Liu, A. Kansal, and A. Sivasubramaniam, "Cuanta: Quantifying effects of shared on-chip resource interference for consolidated virtual machines," in *Proceedings of the 2Nd ACM Symposium on Cloud Computing*, ser. SOCC '11. New York, NY, USA: ACM, 2011, pp. 22:1–22:14. [Online]. Available: http://doi.acm.org/10.1145/2038916.2038938

[3] J. Mars, L. Tang, R. Hundt, K. Skadron, and M. L. Soffa, "Bubble-up: Increasing utilization in modern warehouse scale computers via sensible co-locations," in *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-44. New York, NY, USA: ACM, 2011, pp. 248–259. [Online]. Available: http://doi.acm.org/10.1145/2155620.2155650

[4] R. Nathuji, A. Kansal, and A. Ghaffarkhah, "Q-clouds: Managing performance interference effects for qos-aware clouds," in *Proceedings of the 5th European Conference on Computer Systems*, ser. EuroSys '10. New York, NY, USA: ACM, 2010, pp. 237–250. [Online]. Available: http://doi.acm.org/10.1145/1755913.1755938

[5] H. Kasture and D. Sanchez, "Ubik: Efficient cache sharing with strict qos for latency-critical workloads," in *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '14. New York, NY, USA: ACM, 2014, pp. 729–742. [Online]. Available: http://doi.acm.org/10.1145/2541940.2541944

[6] X. Yang, S. M. Blackburn, and K. S. McKinley, "Elfen scheduling: Fine-grain principled borrowing from latency-critical workloads using simultaneous multithreading," in *2016 USENIX Annual Technical Conference (USENIX ATC 16)*. Denver, CO: USENIX Association, 2016, pp. 309–322. [Online]. Available: https://www.usenix.org/conference/atc16/technical-sessions/presentation/yang

[7] L. A. Barroso and U. Hoelzle, *The Datacenter As a Computer: An Introduction to the Design of Warehouse-Scale Machines*, 1st ed. Morgan and Claypool Publishers, 2009.

[8] H. Zhu and M. Erez, "Dirigent: Enforcing qos for latency-critical tasks on shared multicore systems," in *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '16. New York, NY, USA: ACM, 2016, pp. 33–47. [Online]. Available: http://doi.acm.org/10.1145/2872362.2872394

[9] D. Lo, L. Cheng, R. Govindaraju, P. Ranganathan, and C. Kozyrakis, "Heracles: Improving resource efficiency at scale," in *Proceedings of the 42Nd Annual International Symposium on Computer Architecture*, ser. ISCA '15. New York, NY, USA: ACM, 2015, pp. 450–462. [Online]. Available: http://doi.acm.org/10.1145/2749469.2749475

[10] H. Yun, G. Yao, R. Pellizzoni, M. Caccamo, and L. Sha, "Memguard: Memory bandwidth reservation system for efficient performance isolation in multi-core platforms," in *2013 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, April 2013, pp. 55–64.

[11] M. Ferdman, A. Adileh, O. Kocberber, S. Volos, M. Alisafaee, D. Jevdjic, C. Kaynak, A. D. Popescu, A. Ailamaki, and B. Falsafi, "Clearing the clouds: A study of emerging scale-out workloads on modern hardware," in *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XVII. New York, NY, USA: ACM, 2012, pp. 37–48. [Online]. Available: http://doi.acm.org/10.1145/2150976.2150982

[12] H. Cook, M. Moreto, S. Bird, K. Dao, D. A. Patterson, and K. Asanovic, "A hardware evaluation of cache partitioning to improve utilization and energy-efficiency while preserving responsiveness," in *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ser. ISCA '13. New York, NY, USA: ACM, 2013, pp. 308–319. [Online]. Available: http://doi.acm.org/10.1145/2485922.2485949

[13] H. Kasture and D. Sanchez, "Tailbench: a benchmark suite and evaluation methodology for latency-critical applications," in *2016 IEEE International Symposium on Workload Characterization (IISWC)*, Sept 2016, pp. 1–10.

[14] "STREAM Benchmark," http://www.cs.virginia.edu/stream/ref.html.

[15] S. Huang, J. Huang, J. Dai, T. Xie, and B. Huang, "The hibench benchmark suite: Characterization of the mapreduce-based data analysis," in *2010 IEEE 26th International Conference on Data Engineering Workshops (ICDEW 2010)*, March 2010, pp. 41–51.

[16] K. Hurt and E. John, "Analysis of memory sensitive spec cpu2006 integer benchmarks for big data benchmarking," in *Proceedings of the 1st Workshop on Performance Analysis of Big Data Systems*, ser. PABS '15. New York, NY, USA: ACM, 2015, pp. 11–16. [Online]. Available: http://doi.acm.org/10.1145/2694730.2694732

[17] R. Mian, P. Martin, and J. L. Vazquez-Poletti, "Provisioning data analytic workloads in a cloud," *Future Gener. Comput. Syst.*, vol. 29, no. 6, pp. 1452–1458, Aug. 2013. [Online]. Available: http://dx.doi.org/10.1016/j.future.2012.01.008

[18] C. jean Wu and M. Martonosi, "A comparison of capacity management schemes for shared cmp caches."

[19] Y. Xie and G. H. Loh, "Pipp: Promotion/insertion pseudo-partitioning of multi-core shared caches," in *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ser. ISCA '09. New York, NY, USA: ACM, 2009, pp. 174–183. [Online]. Available: http://doi.acm.org/10.1145/1555754.1555778

[20] P. Ranganathan, S. Adve, and N. P. Jouppi, "Reconfigurable caches and their application to media processing," in *Proceedings of the 27th Annual International Symposium on Computer Architecture*, ser. ISCA '00. New York, NY, USA: ACM, 2000, pp. 214–224. [Online]. Available: http://doi.acm.org/10.1145/339647.339685

[21] 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 39th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO 39. Washington, DC, USA: IEEE Computer Society, 2006, pp. 423–432. [Online]. Available: http://dx.doi.org/10.1109/MICRO.2006.49

[22] D. Sanchez and C. Kozyrakis, "Vantage: Scalable and efficient fine-grain cache partitioning," in *Proceedings of the 38th Annual International Symposium on Computer Architecture*, ser. ISCA '11. New York, NY, USA: ACM, 2011, pp. 57–68. [Online]. Available: http://doi.acm.org/10.1145/2000064.2000073

[23] R. Manikantan, K. Rajan, and R. Govindarajan, "Probabilistic shared cache management (prism)," in *Proceedings of the 39th Annual International Symposium on Computer Architecture*, ser. ISCA '12. Washington, DC, USA: IEEE Computer Society, 2012, pp. 428–439. [Online]. Available: http://dl.acm.org/citation.cfm?id=2337159.2337208

[24] S. Srikantaiah, M. Kandemir, and Q. Wang, "Sharp control: Controlled shared cache management in chip multiprocessors," in *Proceedings of the 42Nd Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO 42. New York, NY, USA: ACM, 2009, pp. 517–528. [Online]. Available: http://doi.acm.org/10.1145/1669112.1669177

[25] R. Liu, K. Klues, S. Bird, S. Hofmeyr, K. Asanović, and J. Kubiatowicz, "Tessellation: Space-time partitioning in a manycore client os," in *Proceedings of the First USENIX Conference on Hot Topics in Parallelism*, ser. HotPar'09. Berkeley, CA, USA: USENIX Association, 2009, pp. 10–10. [Online]. Available: http://dl.acm.org/citation.cfm?id=1855591.1855601

[26] F. Guo, H. Kannan, L. Zhao, R. Illikkal, R. Iyer, D. Newell, Y. Solihin, and C. Kozyrakis, "From chaos to qos: Case studies in cmp resource management," *SIGARCH Comput. Archit. News*, vol. 35, no. 1, pp. 21–30, Mar. 2007. [Online]. Available: http://doi.acm.org/10.1145/1241601.1241608

[27] F. Guo, Y. Solihin, L. Zhao, and R. Iyer, "A framework for providing quality of service in chip multi-processors," in *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO 40. Washington, DC, USA: IEEE Computer Society, 2007, pp. 343–355. [Online]. Available: http://dx.doi.org/10.1109/MICRO.2007.6

[28] R. Iyer, "Cqos: A framework for enabling qos in shared caches of cmp platforms," in *Proceedings of the 18th Annual International Conference on Supercomputing*, ser. ICS '04. New York, NY, USA: ACM, 2004, pp. 257–266. [Online]. Available: http://doi.acm.org/10.1145/1006209.1006246

[29] R. Iyer, L. Zhao, F. Guo, R. Illikkal, S. Makineni, D. Newell, Y. Solihin, L. Hsu, and S. Reinhardt, "Qos policies and architecture for cache/memory in cmp platforms," in *Proceedings of the 2007 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, ser. SIGMETRICS '07. New York, NY, USA: ACM, 2007, pp. 25–36. [Online]. Available: http://doi.acm.org/10.1145/1254882.1254886

[30] N. Vasić, D. Novaković, S. Miučin, D. Kostić, and R. Bianchini, "Dejavu: Accelerating resource allocation in virtualized environments," *SIGARCH Comput. Archit. News*, vol. 40, no. 1, pp. 423–436, Mar. 2012. [Online]. Available: http://doi.acm.org/10.1145/2189750.2151021

[31] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. Katz, S. Shenker, and I. Stoica, "Mesos: A platform for fine-grained resource sharing in the data center," in *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI'11. Berkeley, CA, USA: USENIX Association, 2011, pp. 295–308. [Online]. Available: http://dl.acm.org/citation.cfm?id=1972457.1972488

[32] 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 40th Annual International Symposium on Computer Architecture*, ser. ISCA '13. New York, NY, USA: ACM, 2013, pp. 607–618. [Online]. Available: http://doi.acm.org/10.1145/2485922.2485974

[33] D. Novaković, N. Vasić, S. Novaković, D. Kostić, and R. Bianchini, "Deepdive: Transparently identifying and managing performance interference in virtualized environments," in *Presented as part of the 2013 USENIX Annual Technical Conference (USENIX ATC 13)*. San Jose, CA: USENIX, 2013, pp. 219–230. [Online]. Available: https://www.usenix.org/conference/atc13/technical-sessions/presentation/novaković

[34] C. Delimitrou and C. Kozyrakis, "Paragon: Qos-aware scheduling for heterogeneous datacenters," in *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '13. New York, NY, USA: ACM, 2013, pp. 77–88. [Online]. Available: http://doi.acm.org/10.1145/2451116.2451125

[35] ——, "Quasar: Resource-efficient and qos-aware cluster management," in *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '14. New York, NY, USA: ACM, 2014, pp. 127–144. [Online]. Available: http://doi.acm.org/10.1145/2541940.2541941

[36] X. Zhang, E. Tune, R. Hagmann, R. Jnagal, V. Gokhale, and J. Wilkes, "Cpi2: Cpu performance isolation for shared compute clusters," in *Proceedings of the 8th ACM European Conference on Computer Systems*, ser. EuroSys '13. New York, NY, USA: ACM, 2013, pp. 379–391. [Online]. Available: http://doi.acm.org/10.1145/2465351.2465388