Cluster Computing (2019) 22:161–174
https://doi.org/10.1007/s10586-018-2828-1

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

Hanul Sung[1] · Jeesoo Min[1] · Sujin Ha[1] · Hyeonsang Eom[1]

Received: 7 January 2018 / Revised: 23 May 2018 / Accepted: 17 July 2018 / Published online: 2 August 2018

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 ones. However, because the memory bandwidth cannot be provided in isolation unlike the cores and cache memory, the latency-critical workloads experience poor performance isolation even though the core and 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 the memory bandwidth to the latency-critical workload and co-executing batch processing ones, firstly, our proposed approach performs few pre-profilings under the assumption that memory bandwidth contention is the worst with a divide and conquer method. Second, we predict the memory bandwidth to meet the SLO for all queries per seconds (QPSs) based on results of the pre-profilings. 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 ones. It is experimentally found that our proposed approach can achieve up to 99% SLO assurance and improve the server utilization up to 6.5×.

Keywords Latency-critical workload · QoS · SLO · Server utilization · Batch processing workload

1 Introduction

Latency-critical workloads, such as web search engines, social networks and finance market applications, rely on data centers consisting of a large number of server nodes. Since requests of the workloads are issued across the server nodes with network interactions, some requests take a long time to respond and it is called a tail latency [1, 2]. The latency is the most intuitive indicator for clients to assess the quality of the services [3, 4]. If they experience the unexpected tail latencies, not stable or high latencies, it devours revenue directly [5, 6]. According to experiments at Amazon online shop, sales actually decreased by 1% whenever the page load time increased by 100 ms [7]. Therefore, the latency-critical workloads are sensitive to the tail latencies for meeting the SLOs between a service provider and clients [8]. Unexpected tail latencies are caused by sharing hardware resources, CPUs, a cache memory and a memory controller with other workloads [9–11]. Thus, the service providers for the latency-critical workloads use the entire data centers only for them to meet the SLOs [12]. The level of hardware resource utilization for the data center is 10–45% and it is wasted with most of the resources unused [13, 14].

✉ Hyeonsang Eom
hseom@cse.snu.ac.kr

Hanul Sung
husung@dcslab.snu.ac.kr

Jeesoo Min
jsmin@dcslab.snu.ac.kr

Sujin Ha
sjha@dcslab.snu.ac.kr

[1] Department of Computer Science and Engineering, Seoul National University, Seoul, Republic of Korea

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 [12, 14–16]. 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 [17].

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 [18, 19]. 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, or *OMBM*, for strict QoS and high hardware resource utilization without any runtime monitoring and profiling. The proposed approach increases the hardware resource utilization by co-locating the latency-critical workload and the batch processing ones. And it also meets the 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.

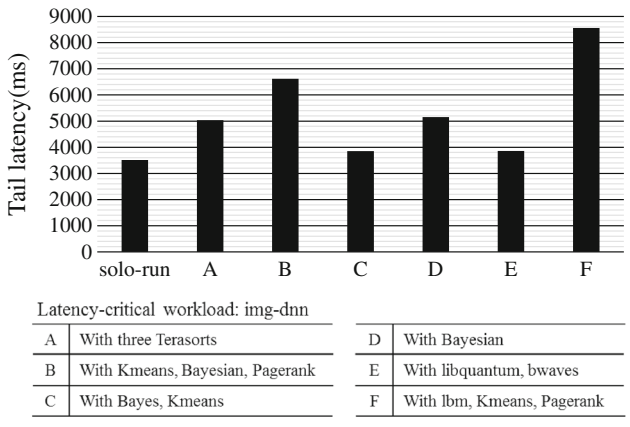


Fig. 1 Performance isolation result with isolated core and cache memory for a latency-critical workload

First, as you can see in Fig. 1, the tail latencies are changed sensitively depending on the memory bandwidth usage characteristics of the co-executing batch processing workloads. Thus, whenever the co-executing batch processing workloads are changed, we need to pre-profile newly for meeting the SLOs. This is impossible in practice.

Second, the QPSs for the latency-critical workloads are diverse [20] and the SLOs that could be met by the service providers also vary. Thus, it is not possible to detect the memory bandwidth which guarantees various SLOs for all QPSs through pre-profilings, either.

To address the problems, we performed the pre-profilings on the assumption that the contention for the shared memory bandwidth is the worst with a divide and conquer method. Based on the results of the minimal pre-profilings, we created two prediction graphs and predicted the amount of the memory bandwidth to meet all SLOs for all QPSs.

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 up to 6.5×.

In the rest of this paper, Sect. 2 introduces the background and motivation. Section 3 describes the *OMBM* approach with a flow chart, algorithms and figures. Section 4 demonstrates the effectiveness of our proposed approach by showing the results of prediction accuracy, SLOs guarantee and server utilization. Section 5 discusses some related research. Finally Sect. 6 concludes the paper.

2 Background and motivation

This section introduces the overall architecture of our proposed approach, *OMBM*. To help the readers better understand it, we explain an existing technique for memory bandwidth isolation in the Background subsection. Then, we show examples which motivate our proposed approach.

2.1 Memory bandwidth isolation

Memguard [17] is a memory bandwidth reservation system which is implemented on the operating systems in attempt to provide isolated bandwidth. The system cannot provide physically isolated memory bandwidth, but it just adjusts the amount of the memory bandwidth used using an operating system scheduler.

For example, there are two workloads, A and B, running on a machine with 10 GB/s of memory bandwidth. 2 MB/s is reserved for A and 8 MB/s for B. When a new time slice which is provided by the operationg system starts, they share the memory controller at the same time. If A spends all of the own reserved memory bandwidth, 2 MB/s, A cannot use the memory controller anymore until the next time slice. And B can use the memory controller alone before the new time slice. But if both of them cannot fill their reservations in a time slice, then they continue to share the memory controller. That is, they can be guaranteed the amounts of memory bandwidth reserved, but they might be disturbed by the shared memory controller.

To provide memory bandwidth for temporal isolation, memguard measures the memory bandwidth usage for each workload by monitoring the LLC miss performance counters. If the core/workload overuses the memory bandwidth than the reserved amount, the system asks the operating system scheduler to help the core not use the memory bandwidth anymore. The operating system scheduler dequeues all of the workloads on the core and prevents them from using the memory bandwidth until the next time slice. The system cannot make each workload use the physically isolated memory bandwidth, but it produces similar effects.

2.2 Motivation

Unlike the cores and cache memory, there is no technique which divides the memory bandwidth physically. For this reason, even if the latency-critical workload uses isolated core and cache memory, it cannot experience the expected tail latencies due to the contention for the memory bandwidth.

To address this problem, our approach utilizes memguard to support isolated memory bandwidth to workloads. However, since the system is implemented with CPU scheduling, it cannot make the memory controller be completely physically isolated. Thus there is still a limitation in that the service provider guarantees strict QoS and provides high server utilization.

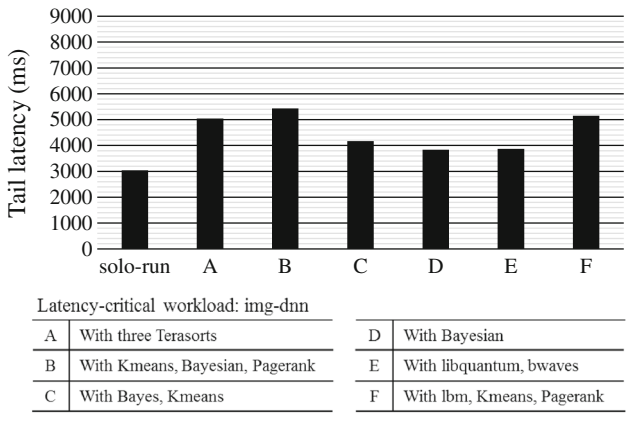


Fig. 2 Performance isolation result for a latency-critical workload with the memguard system

Figure 2 shows tail latencies of the latency-critical workload with applying the memguard system. Unlike Fig. 1, all of the shared resources including memory bandwidth are provided being isolated in Fig. 2. Despite all of the shared resources are provided being isolated, the performance of executing alone is different from that of executing with other batch processing workloads. In addition, the performance results of the latency-critical workload are different depending on the co-executing batch processing ones. That is because the memguard system does not make the workloads use the physically separate memory controller, but only considers the memory controller usage of each workload. Thus they share the memory controller at the same time and the tail latencies are changed depending on the memory bandwidth usage characteristic of co-executing batch processing workloads. As a result, we cannot expect that the latency-critical workload experiences stable tail latencies, even if it is provided with the same amount of memory bandwidth.

Because of this limitation, a huge amount of pre-profiling is needed to prepare for all concurrent executions with any batch processing workloads. In addition, since tail latencies have effects different from the contention for the memory controller corresponding to QPSs, we should also perform the pre-profiling at various QPSs. However, this is difficult in practice.

3 OMBM design

In order to solve the problem mentioned above, we propose an Optimized Memory Bandwidth Management approach, or *OMBM*, for meeting SLOs and improving the hardware resource utilization. The main idea of *OMBM* is (1) assuming that the contention for the memory bandwidth is the worst for handling every situation and (2) utilizing a divide and conquer method for minimal profiling. By using them, we predict the amount of memory bandwidth for the latency-critical workload without numerous pre-profiling. This section gives details of our proposed approach, *OMBM*.

3.1 OMBM overview

Figure 3 illustrates the architecture of *OMBM*. Our proposed approach consists of two components, (1) *predictor* and (2) *allocator*. The *predictor* performs pre-profilings and decides the memory bandwidth to be allocated to the latency-critical workload with the few sampled QPSs.

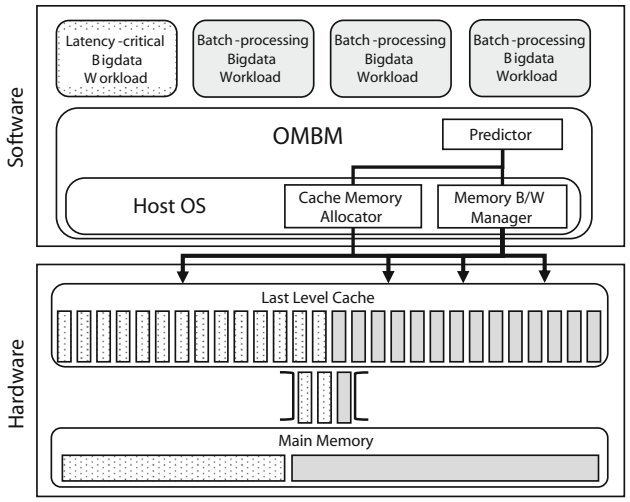


Fig. 3 Proposed approach architecture

Based on the results of pre-profiling, the *predictor* creates two prediction graphs. These graphs are used to 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 of the batch processing ones.

Figure 4 shows the processing flows of *OMBM*. As shown in the figure, the *predictor* is divided into two parts, a *online predictor* and a *offline predictor* depending on when it is executed.

Before the actual execution, the *offline predictor* starts pre-profilings of the latency-critical workload and creates the first graph. The *online predictor* makes the second graph based on the first graph and predicts the memory bandwidth of the latency-critical workloads for meeting the SLO with the second graph at runtime.

For reducing pre-profiling overhead, the service providers select the memory bandwidth to pre-profile in advance and the QPSs are chosen by using the divide and conquer method. Most of all, we assume that contention for the memory bandwidth is the worst for strict QoS no matter what batch processing workloads are executed.

First, the *offline predictor* initializes the first sampled memory bandwidth and the first QPS range, MIN, MAX, and MED, for pre-profilings. And it makes the contention for the memory bandwidth the worst by executing the STREAM benchmark [21] which makes tremendous memory contention and allocates as much isolated memory bandwidth to the latency-critical workload as the sampled memory bandwidth using memguard. Then it starts measuring the tail latencies at MIN, MAX and MED QPSs. After finishing the measurement, it decides the new range

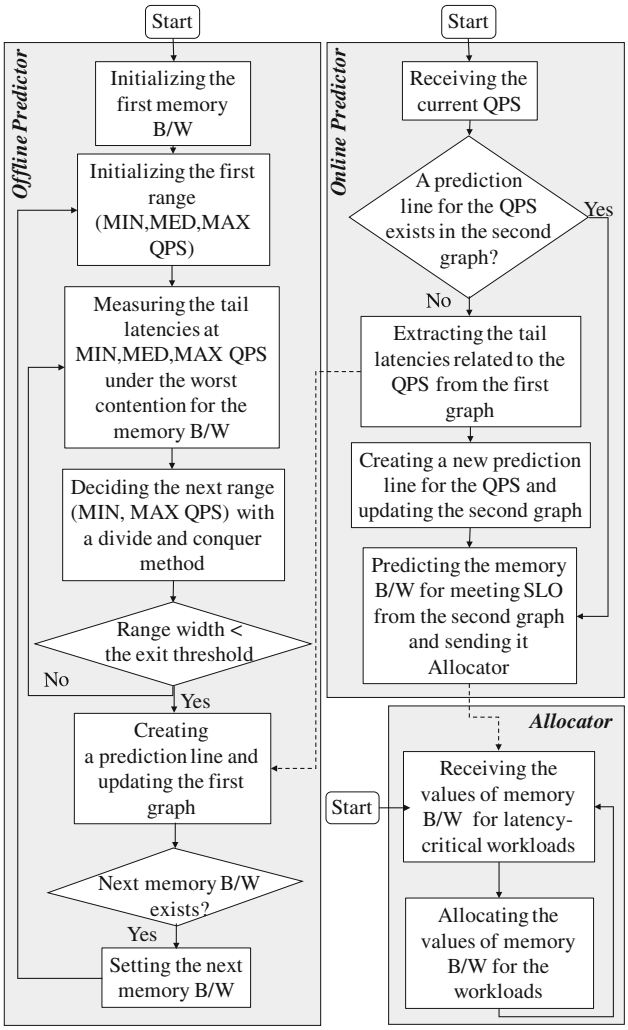


Fig. 4 Flow chart of the proposed OMBM

for the next pre-profilings using the divide and conquer method. And then it checks whether the new range width is smaller than the pre-defined exit threshold. (The pre-defined exit threshold is also set by the service providers.) If it does not, the *offline predictor* continues to pre-profile with the new range. Otherwise, the pre-profiling for that sampled memory bandwidth is over. And the *offline predictor* creates a prediction line for the memory bandwidth and adds it to the first graph. Next, it checks whether a next sampled memory bandwidth exists or not. If the sampled memory bandwidth remains, the *offline predictor* starts measuring the tail latencies corresponding to the new memory bandwidth. but, if there is not any sampled memory bandwidth, it finishes the pre-profilings.

After performing the pre-profilings, the *online predictor* is started. When it receives the current QPS of the latency critical workload, it checks that a prediction line for the QPS already exists in the second graph. If the QPS occurred before, the prediction line is in the second graph.

But it did not, the *online predictor* creates a prediction line by extracting the tail latencies related to the QPS from the first graph. And then it updates the second graph by inserting the prediction line. With the prediction line, it predicts the amount of the memory bandwidth for the latency-critical workload and sends it to the *allocator*.

Finally, the *allocator* assigns as much isolated memory bandwidth as the *online predictor* reported to the latency-critical workload and the rest is allocated to the batch processing ones.

The following subsections give more detailed technical information about the *predictor* and the *allocator*.

3.2 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 limitation on the memory bandwidth isolation as mentioned above, the latency-critical workload cannot experience the expected tail latencies. For these reasons, we need to execute a lot of pre-profilings. However, it is realistically impossible.

To reduce numerous pre-profilings, we propose two ideas. First, in order to meet the SLOs strictly, we assume the worst case where the contention for the memory bandwidth is the highest and predict the amount of memory bandwidth for the latency-critical workload for all QPSs. To create the extreme contention for the memory bandwidth, the STREAM benchmarks causing the tremendous contention for the 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 be pre-profiled with the divide and conquer method and performs the 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-profilings. The graphs show the predicted amount of memory bandwidth to meet the SLOs for all QPSs. The first graph is created by the *offline predictor* in advance and the second one is created by the *online predictor* 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-profilings 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 the SLOs. The second graph is continually updated whenever the 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 Pseudo-code for the 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-criticalworkload
IC_{XD}, IC_{DN} : Margins of increase values from MAX to MED and from MED to MIN
$TL = \{tl_{DQN} \sim tl_{DQX}\}$: Tail latency

```
for i ← 0 to n
    CBW ← bw_i
    MAX ← DQX, MIN ← DQN
    while
        MED ← (MAX − MIN)/2
        tl_MAX ← Tail latency of max QPS generated by pre-profiling
        tl_MIN ← Tail latency of min QPS generated by pre-profiling
        tl_MED ← Tail latency of medium QPS generated by pre-profiling
        TL ← TL ∪ tl_MAX ∪ tl_MIN ∪ tl_MED
        IC_XD ← tl_MAX ÷ tl_MED
        IC_DN ← tl_MED ÷ tl_MIN
        if(IC_XD > IC_DN)
            MIN ← MED
        else
            MAX ← MED
        if(MAX − MIN < ETV)
            break
    Creating fpl_i by connecting all of tail latencies of TL
    FPG ← FPG ∪ fpl_i
```

Algorithm 1 takes four inputs: (1) memory bandwidth for the 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}. First, the *predictor* starts to create a prediction line constituting the first prediction graph from the smallest amount of memory bandwidth (Lines 1–2). 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 5–8). In order to identify the rapidly increasing range accurately, the *offline predictor* calculates two margins of increase of tail latencies, from MAX to MED and from MED to MIN (Lines 10–11). In order to prepare for the next pre-profiling step, the *offline 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 12–15). In this way, the *offline predictor* defines the new range and starts the pre-profiling. However, if the interval between the *MAX* and *MIN* is smaller than *ETV*, the *offline predictor* stops the pre-profiling (Lines 16–17). When the progressive pre-profiling in the divide and conquer manner is finished, the *predictor* creates a prediction line by connecting the tail latencies obtained by the pre-profilings (Line 18).

The *offline 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).

Figure 5 shows the first prediction graph. We provide five BWS, 0.8, 3.2, 6.4, 19.2 and 24.8 GB/s, thus n is 5. And we set *DQX* and *DQN* to 10,000 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

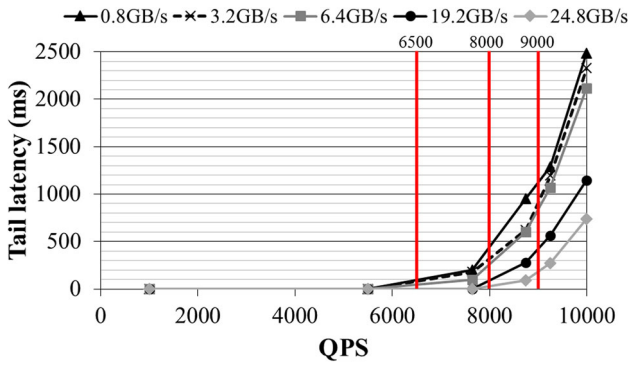


Fig. 5 First prediction graph

shown in the figure, firstly *MAX*, *MIN* and *MED* is set to 10,000, 1000, 5500 and the pre-profiling is executed. Then the *offline predictor* defines the range which changes more increase rapidly as the new range for the next pre-profiling, thus *MIN* is set to 5500 and *MED* is set to 7750. The prediction lines are created by repeating the 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)*

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 *online 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 the memory bandwidth at the current QPS from the first prediction line (Lines 3–5). Then the *online predictor* connects all of the tail latencies in *ETL* for creating the prediction line for the current QPS and merge it into the second prediction graph (Lines 6–7). Finally, the *predictor* predicts the memory bandwidth to meet the SLO using *SPG* and reports the predicted result to *allocator* (Lines 10–11).

Figure 6 shows the second prediction graph which is created based on red lines of Fig. 5. Using the graph, we

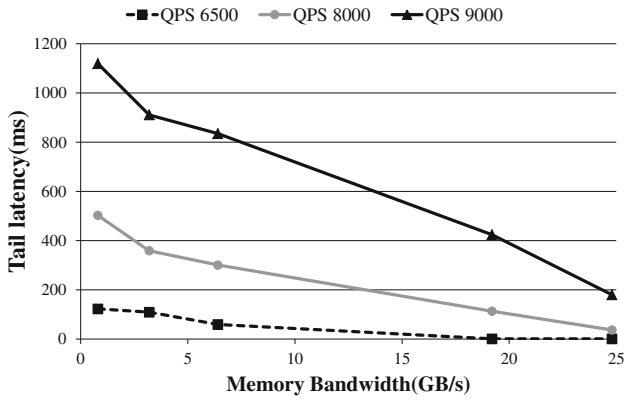


Fig. 6 Second prediction graph

predict the memory bandwidth as 14 GB/s for meeting an SLO, 200 ms at 8000 QPS.

3.3 Allocator

To provide isolated memory bandwidth for the latency-critical workload and the batch processing workloads, the *allocator* divides the memory bandwidth into two based on the memory bandwidth transferred from the *online predictor*. And it provides the amount of the isolated memory bandwidth received from the *online 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 approach, for providing isolated cache memory. The cache *allocator* divides the cache memory in half and provides them to the latency-critical workload and the batch processing ones.

4 Experimental evaluation

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

4.1 Experiment environment and benchmark workloads

In this research, we performed experiments on an Intel Haswell server with a quad-core Intel i7-4770 k 3.5 GHz processor, 8 MB 16-way set associative cache memory and 24.8 GB/s memory bandwidth and used Linux 3.14.15 kernel and Hadoop 2.6. We used a Tailbench benchmark [20] developed by MIT as the latency-critical workloads and a HiBench benchmark [22], a representative of map-reduce workloads and SPECcpu2006 which has characteristics to be a good candidate for big data processing benchmarks [23], as the batch processing workloads. Table 1 shows some details of these workloads.

4.2 Prediction accuracy

In order to predict the amount of memory bandwidth to meet SLOs for all QPSs, our approach creates the prediction graphs with few pre-profilings. We select the amount of memory bandwidth for the pre-profilings and reduces the number of QPSs to be profiled 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 7 shows the prediction lines of the second prediction graph at a specific QPS and the real tail latencies by adjusting the allocated memory bandwidth. To make the same contention for memory bandwidth, we ran the STREAM benchmarks. We used silo, masstree and img-dnn of the Tailbench benchmark as the latency-critical workloads in Fig. 7a–c. We just performed 30 pre-profilings with the sampled memory bandwidth values, 0.8, 3.2, 6.4, 19.2 and 24.8 GB/s and the exit threshold is 90. And we obtained the prediction accuracies, 99% for silo, 76% for masstree and 92% for img-dnn.

Obviously, the prediction accuracies depend on the sampled memory bandwidth and the exit threshold. First, since the workloads have different levels of sensitiveness to the memory bandwidth, although the number of the pre-profilings is the same, the prediction accuracies vary depending on the sampled memory bandwidth. Figure 8 shows the change of prediction accuracy depending on the sampled memory bandwidth. We collected two samples, each of which consists of five memory bandwidth values, 0.8, 3.2, 6.4, 19.2 and 24.8 GB/s for the first sample and 0.8, 6.4, 12.4, 19.2 and 24.8 GB/s for the second one. The first sample is biased towards low and high memory bandwidth values, but the second sample contains equally divided memory bandwidth ones. As shown in Fig. 8a, because the actual latencies of masstree fall near 11 MB/s, prediction is performed quite well for the second sample, but the change cannot be detected for the first sample. On the contrary, img-dnn has actual latencies that change near 3 MB/s. Thus, it is possible to find out the change closely for the first sample, but it is not for the second one.

Second, as the number of the sampled memory bandwidth values for the pre-profilings increases, the prediction accuracies also increase. We made three prediction lines with three samples which consist of three, five and seven memory bandwidth values; The first prediction line (labeled 3 in the figure) has 0.8, 6.4 and 24.8 GB/s, the second one (labeled 5), 0.8, 3.2, 6.4, 19.2 and 24.8 GB/s, and the third one (labeled 7), 0.8, 1.6, 3.2, 6.4, 12.8, 19.2 and 24.8

Table 1 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
SPEC cpu2006	libquantum	Physics/quantum computing
	lbm	Computational fluid dynmaics, Lattice Boltzmann Method
	bwaves	Computational fluid dynamics

GB/s, as shown in Fig. 9. Three prediction lines show similar tendencies, but they are slightly different between 1 and 3 MB/s. Since the prediction lines, 3 and 5, there is not any result of pre-profiling in the range, it is not possible to find out the change of the actual latencies. Unlike them, 7 permits predicting the change due to the result of the pre-profiling at 1.6 GB/s.

Finally, since the number of the pre-profilings varies by the exit threshold, the prediction accuracies are also changed with the exit threshold. The smaller the exit threshold value is, the greater becomes the number of sampled QPSs for the pre-profilings. We set the minimal interval to 2.5, 5, 10, 20 and 40% for the exit threshold values; as shown in Fig. 10, the prediction lines with the exit threshold less than 10% are almost similar. But, two prediction lines with the exit threshold 20 and 40% are far from the actual tail latencies.

4.3 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 utilization measured with our approach or not. Since the prediction lines are created in 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.

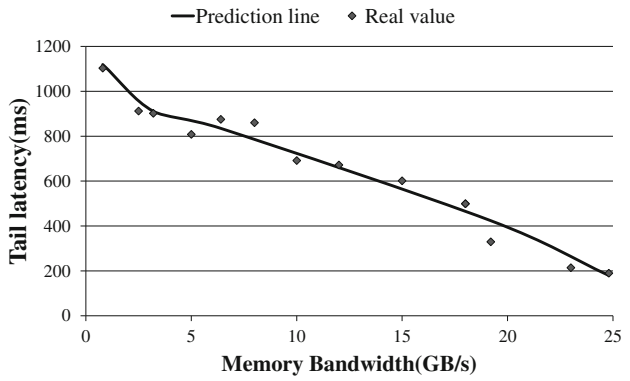
The actual tail latencies and the prediction lines created with the sampled memory bandwidth 0.8, 3.2, 6.4, 19.2 and 24.8 GB/s and the exit threshold is 90 by our proposed approach are shown in Fig. 11. In the figure, the latency-critical workloads, silo, masstree and img-dnn, are executed with co-executing combinations of the various batch processing workloads as described in Table 2 with different QPSs. In Fig. 11a, regardless of the type of the combination, most actual tail latencies are less than or equal to the prediction line. Among them, 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. On the other hand, because *A* and *B* cannot make an interference on the memory controller as much as the worst case, actual latencies are away from the prediction lines.

As shown in Fig. 11b, most latencies are lower the prediction lines and the tail latencies are not changed much even when the memory bandwidth changes. Through this, we can infer that all combination of the batch processing workloads does not affect the silo much. And differences between the prediction values and the actual tail latencies are different according to 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.

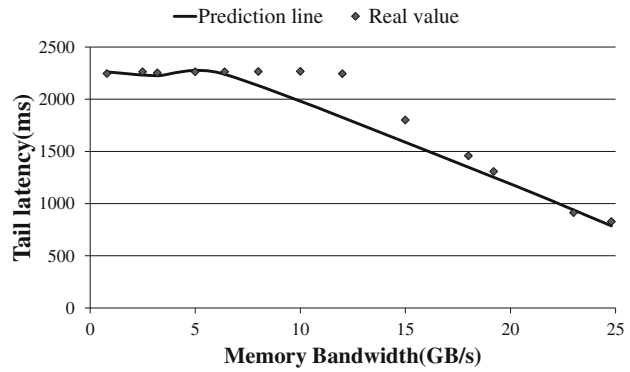
Unlike Fig. 11b, c shows similar values between the actual tail latencies and the prediction line because img-dnn is sensitive to the memory contention. Although the difference between the predicted values and the actual ones is different depending on the type of the co-executing batch processing workloads and QPS, all the actual values are close to and smaller than the prediction line.

In the previous experiments, the latency-critical workloads have the static QPSs. But, most latency-critical workloads have constantly changing QPSs in practice. To demonstrate the effectiveness of our proposed approach, we performed additional experiments with the latency-critical workloads which have dynamic QPSs. In order to do this, we modified the Tailbench benchmark to dynamically produce the QPSs; silo had 1000–10,000 QPSs, masstree, 1000–150,000 QPSs, and img-dnn, 100–1500 QPSs.

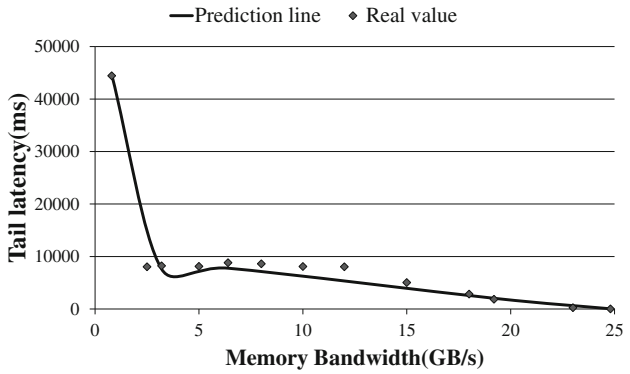
Figure 12 shows the results for the latency-critical workloads with dynamic QPSs. The red lines indicate the SLO in the figure, and we set the SLO to 200, 2000 and 5000 ms to silo, masstree and img-dnn, respectively. As



(a) silo

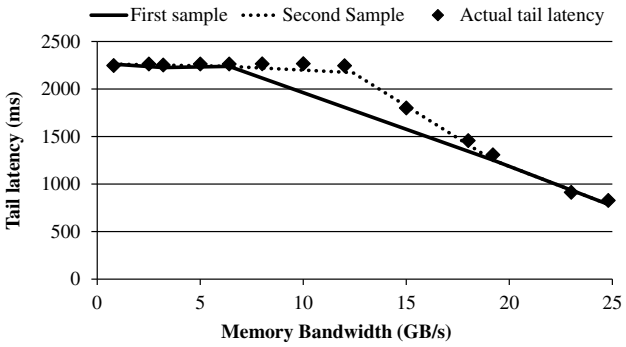


(b) masstree

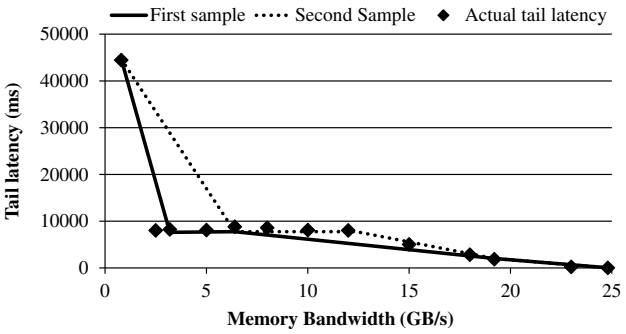


(c) img-dnn

Fig. 7 Prediction accuracy



(a) mastree



(b) img-dnn

Fig. 8 Prediction accuracy depending on the sampled memory bandwidth values

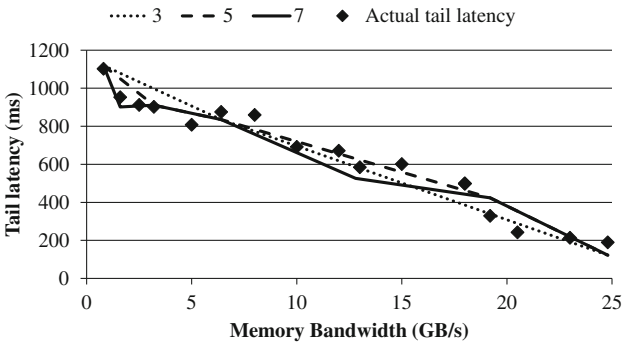


Fig. 9 Prediction accuracy depending on the number of sampled memory bandwidth values

shown in the figures, most tail latencies for them are less than or equal to the red line.

In Fig. 12a, only four results are close to the SLO, about the half of the rest reside in the middle, and the others are close to zero. This means that although the SLO is guaranteed strictly, a lot of memory bandwidth is wasted due to the worst memory contention assumption.

Unlike Fig. 12a–c show that most tail latencies are far from zero and they are relatively close to the SLO. In other words, masstree and img-dnn used memory bandwidth efficiently compared to silo. However, because the latency-critical workloads were differently affected by the memory contention depending on the range of QPSs, these results might be shown to be varied.

Table 3 shows the hardware resource utilization with and without 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 5.1× on average and up to 6.5×.

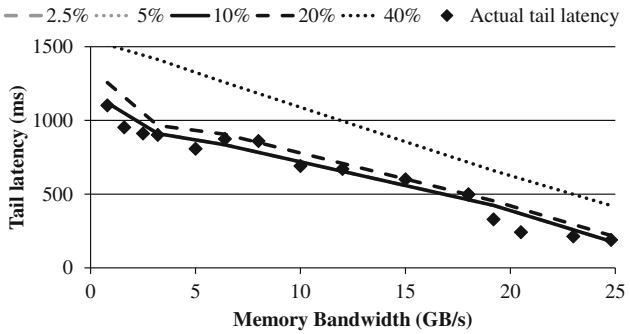


Fig. 10 Prediction accuracy depending on the number of sampled QPSs

5 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 [24]. For this reason, many studies have been conducted such as resource isolation approaches [12, 25–29] and interference-aware management schemes [10, 11, 30–34].

Kasture et al. proposed a cache memory isolation scheme which allocates the isolated cache memory to latency-critical workloads and batch processing ones [12]. 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. [12, 16]. 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 [15]. For meeting SLOs, the system monitors QoSs persistently at runtime. If monitored QoSs are close to the SLO, then the system adjusts the amount of the shared resources by stopping the batch processing workloads. And if tail latencies are far from the SLO, the system restarts the batch processing ones for high server utilization.

Mars et al. investigated performance interferences between a latency-critical workload and a batch processing one by co-locating them [10, 34]. 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. Unlike the system [10], the system [34] performs profilings online. it monitors QoS

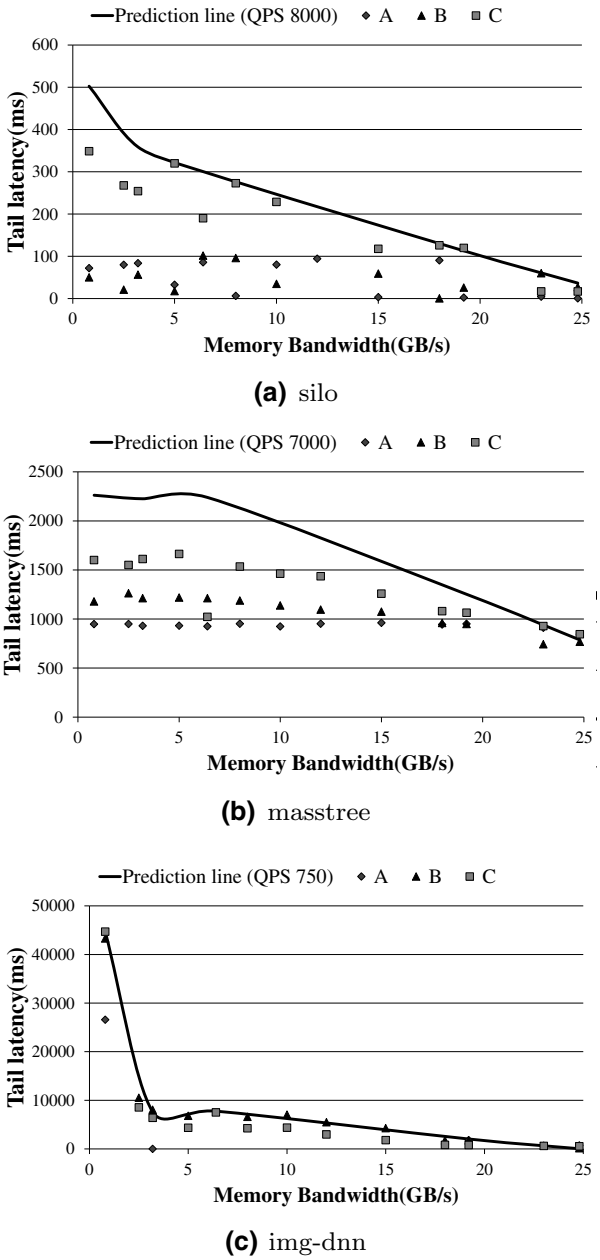


Fig. 11 SLOs guarantee running with batch processing workloads which have static QPS

Table 2 Workload combinations used in experiments

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

at runtime and controls executions of the batch processing workloads based on the results. But, because the system requires all profiled results of the batch processing

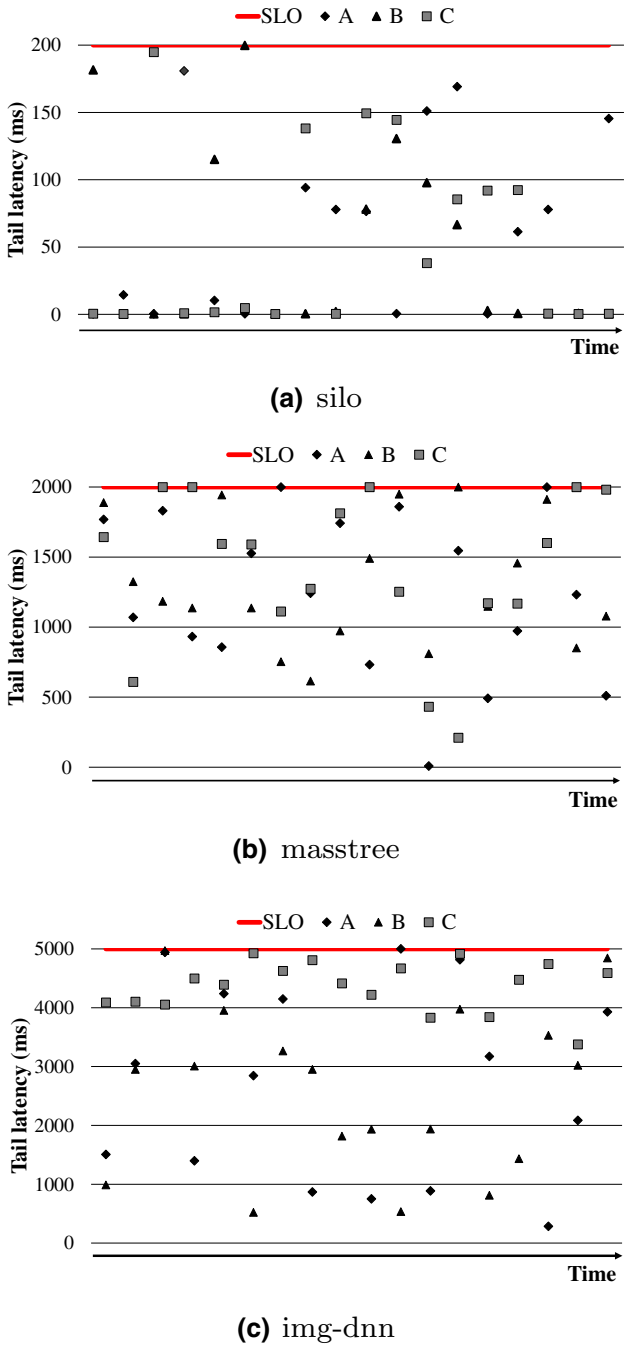


Fig. 12 SLOs guarantee running with batch processing workloads which have dynamic QPSs

workloads, whenever new batch processing ones start executing, the system also starts profiling. Delimitrou et al. performed pre-profilings with various server configurations for finding a server to meet SLOs [30]. The system can find 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.

Table 3 CPU utilization

Combination	Latency-critical workload alone (%)	With three batch workloads (%)
silo		
A	17.77	98.57
B	17.90	73.75
C	15.07	98.90
masstree		
A	24.82	99.57
B	24.15	98.00
C	15.52	96.70
img-dnn		
A	17.77	98.57
B	13.45	68.47
C	18.92	92.00

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

6 Conclusion and future work

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 core and 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* is divided into two parts, a *offline predictor* and a *online predictor*. For reducing the pre-profiling overhead, the *offline predictor* assumes the memory contention is the worst and uses a divide and conquer method. Through the pre-profilings, the *offline predictor* creates a first prediction graph which shows the predicted tail latencies corresponding to all QPSs by adjusting the allocated memory bandwidth in advance. And the *online predictor* makes a second prediction graph which shows the predicted tail latencies according to the memory bandwidth for specific QPSs using the first graph at runtime. Then the *online predictor* predicts the amount of the memory bandwidth for strict QoS of the latency-critical workload, and it lets a *allocator* know. Finally, the *allocator* provides isolated memory bandwidth to the latency-critical workload and the batch processing ones with the information obtained from the *online predictor*. By utilizing this approach, the service provider can meet the SLOs of the latency-critical workloads and improve the server hardware resource utilization.

As you can see in the Sect. 4.2, the prediction accuracies are decided depending on two parameters, the memory bandwidth to be pre-profiled and the exit threshold. The effect of memory bandwidth on the latency varies for every QPS, so each QPS requires different values for the parameters to make the prediction graphs precisely. However, since we do not automatically select the values for the parameters differently depending on the QPS, if the service providers want to get high accuracies in every case, they have to execute profilings by manually adjusting the values. Besides, we do not modify the prediction graphs at runtime, even if the actual tail latencies are far from the precition lines. (But, it is rare that the SLO is not guaranteed.) To address these problems, we need a solution that automatically selects the values for the parameters and updates the prediction graphs based on the actual latencies at runtime. Also, in spite of using our system, the hardware resources are still underutilized due to the assumption where the memory bandwidth contention is worst when we create the prediction graph. To handle this problem, we will improve the predictor which does not waste memory bandwidth, while meeting SLO.

Acknowledgements This research was supported by (1) Institute for 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). It was also partly supported by (2) Institute for Information & communications Technology Promotion (IITP) Grant funded by the Korea government (MSIP) (No. 2017-0-01733, General Purpose Secure Database Platform Using a Private Blockchain), and partly supported by (3) the National Research Foundation (NRF) Grants (NRF-2016M3C4A7952587, NRF-2017M3C4A7083751, PF Class Heterogeneous High Performance Computer Development). In addition, this work was partly supported by (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. Jalaparti, V., Bodik, P., Kandula, S., Menache, I., Rybalkin, M., Yan, C., Jalaparti, V., Bodik, P., Kandula, S., Menache, I., Rybalkin, M., Yan, C.: Speeding up distributed request-response workflows. In: Proceedings of the ACM SIGCOMM 2013 conference on SIGCOMM - SIGCOMM '13, vol. 43, p. 219. ACM Press, New York (2013)
2. Xu, Y., Musgrave, Z., Noble, B., Bailey, M.: Bobtail: avoiding long tails in the cloud (2013)
3. Dabrowski, J.R., Munson, E.V.: Is 100 milliseconds too fast? In: CHI '01 Extended Abstracts on Human Factors in Computing Systems—CHI '01, p. 317. ACM Press, New York (2001)
4. Kapoor, R., Porter, G., Tewari, M., Voelker, G.M., Vahdat, A.: Chronos: predictable low latency for data center applications. In: Proceedings of the Third ACM Symposium on Cloud Computing—SoCC '12, pp. 1–14. ACM Press, New York (2012)
5. Lalith, S., Canini, M., Schmid, S., Feldmann, A.: C3: cutting tail latency in cloud data stores via adaptive replica selection. In: Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation, USENIX Association, p. 296 (2015)
6. Wang, Q., Lai, C.-A., Kanemasa, Y., Zhang, S., Pu, C.: A study of long-tail latency in n-Tier systems: RPC vs. asynchronous invocations. In: Proceedings of the 2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS), pp. 207–217. IEEE (2017)
7. Kohavi, R., Longbotham, R.: Online experiments: lessons learned. Computer **40**(9), 103–105 (2007)
8. Zhu, T., Tumanov, A., Kozuch, M.A., Harchol-Balter, M., Ganger, G.R.: Prioritymeister: tail latency qos for shared networked storage. In: Proceedings of the ACM Symposium on Cloud Computing, SOCC '14, pp. 29:1–29:14. ACM, New York (2014)
9. Govindan, S., Liu, J., Kansal, A., Sivasubramaniam, A.: Cuanta: quantifying effects of shared on-chip resource interference for consolidated virtual machines. In: Proceedings of the 2nd ACM Symposium on Cloud Computing, SOCC '11, pp. 22:1–22:14. ACM, New York (2011)
10. Mars, J., Tang, L., Hundt, R., Skadron, K., Soffa, M.L.: 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, MICRO-44, pp. 248–259. ACM, New York (2011)
11. Nathuji, R., Kansal, A., Ghaffarkhah, A.: Q-clouds: managing performance interference effects for qos-aware clouds. In: Proceedings of the 5th European Conference on Computer Systems, EuroSys '10, pp. 237–250. ACM , New York (2010)
12. Kasture, H., Sanchez, D.: 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, ASPLOS '14, pp. 729–742. ACM, New York (2014)
13. Barroso, L.A., Hoelzle, U.: The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines, 1st edn. Morgan and Claypool Publishers, San Rafael (2009)
14. Yang, X., Blackburn, S.M., McKinley, K.S.: Elfen scheduling: fine-grain principled borrowing from latency-critical workloads using simultaneous multithreading. In: Proceedings of the 2016 USENIX Annual Technical Conference (USENIX ATC 16), pp. 309–322. USENIX Association, Denver (2016)
15. Lo, D., Cheng, L., Govindaraju, R., Ranganathan, P., Kozyrakis, C.: Heracles: improving resource efficiency at scale. In: Proceedings of the 42nd Annual International Symposium on Computer Architecture, ISCA '15, pp. 450–462. ACM, New York (2015)
16. Zhu, H., Erez, M.: 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, ASPLOS '16, pp. 33–47. ACM, New York (2016)
17. Yun, H., Yao, G., Pellizzoni, R., Caccamo, M., Sha, L.: Memguard: memory bandwidth reservation system for efficient performance isolation in multi-core platforms. In: Proceedings of the 2013 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS), pp. 55–64 (2013)
18. Cook, H., Moreto, M., Bird, S., Dao, K., Patterson, D.A., Asanovic, K.: 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, ISCA '13, pp. 308–319. ACM, New York (2013)
19. Ferdman, M., Adileh, A., Kocberber, O., Volos, S., Alisafaee, M., Jevdjic, D., Kaynak, C., Popescu, A.D., Ailamaki, A., Falsafi, B.: 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, ASPLOS XVII, pp. 37–48. ACM, New York (2012)
20. Kasture, H., Sanchez, D.: Tailbench: a benchmark suite and evaluation methodology for latency-critical applications. In: 2016 IEEE International Symposium on Workload Characterization (IISWC), pp. 1–10 (2016)
21. STREAM Benchmark: http://www.cs.virginia.edu/stream/ref.html (2017)
22. Huang, S., Huang, J., Dai, J., Xie, T., Huang, B.: The hibench benchmark suite: characterization of the mapreduce-based data analysis. In: 2010 IEEE 26th International Conference on Data Engineering Workshops (ICDEW 2010), pp. 41–51 (2010)
23. Hurt, K., John, E.: 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, PABS '15, pp. 11–16. ACM, New York (2015)
24. Mian, R., Martin, P., Vazquez-Poletti, J.L.: Provisioning data analytic workloads in a cloud. Future Gener. Comput. Syst. **29**(6), 1452–1458 (2013)
25. Guo, F., Solihin, Y., Zhao, L., Iyer, R.: A framework for providing quality of service in chip multi-processors. In: Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 40, pp. 343–355. IEEE Computer Society, Washington, DC (2007)
26. Iyer, R.: Cqos: a framework for enabling qos in shared caches of cmp platforms. In: Proceedings of the 18th Annual International Conference on Supercomputing, ICS '04, pp. 257–266. ACM, New York (2004)
27. Iyer, R., Zhao, L., Guo, F., Illikkal, R., Makineni, S., Newell, D., Solihin, Y., Hsu, L., Reinhardt, S.: 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, SIGMETRICS '07, pp. 25–36. ACM, New York (2007)
28. Sanchez, D., Kozyrakis, C.: Vantage: scalable and efficient fine-grain cache partitioning. In: Proceedings of the 38th Annual International Symposium on Computer Architecture, ISCA '11, pp. 57–68. ACM, New York (2011)
29. Srikantaiah, S., Kandemir, M., Wang, Q.: Sharp control: controlled shared cache management in chip multiprocessors. In: Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 42, pp. 517–528. ACM, New York (2009)
30. Delimitrou, C., Kozyrakis, C.: Paragon: Qos-aware scheduling for heterogeneous datacenters. In: Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '13, pp. 77–88. ACM, New York (2013)
31. Delimitrou, C., Kozyrakis, C.: Quasar: resource-efficient and qos-aware cluster management. In: Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '14, pp. 127–144. ACM, New York (2014)
32. Novaković, D., Vasić, N., Novaković, S., Kostić, D., Bianchini, R.: Deepdive: transparently identifying and managing performance interference in virtualized environments. In: Presented as Part of the 2013 USENIX Annual Technical Conference (USENIX ATC 13), pp. 219–230. USENIX, San Jose (2013)
33. Vasić, N., Novaković, D., Miučin, S., Kostić, D., Bianchini, R.: Dejavu: accelerating resource allocation in virtualized environments. SIGARCH Comput. Arch. News **40**(1), 423–436 (2012)
34. Yang, H., Breslow, A., Mars, J., Tang, L.: Bubble-flux: precise online qos management for increased utilization in warehouse scale computers. In: Proceedings of the 40th Annual International Symposium on Computer Architecture, ISCA '13, pp. 607–618. ACM, New York (2013)

Hanul Sung is a Ph.D. candidate in computer science and engineering at Seoul National University (SNU), Seoul, Korea, in 2012. She received B.S. degree in computer science from Sangmyung University (SMU), Seoul, Korea, in 2012. Her main research interests are distributed systems, operating systems, high performance storage systems and cloud computing.

Jeesoo Min is a Master Student in computer science and engineering at Seoul National University (SNU), Seoul, Korea. She received B.S. degree in computer science from Handong University, Pohang, Korea, in 2016. Her main research interests are operating systems, cloud computing and storage.

Sujin Ha is a Master Student in computer science and engineering at Seoul National University (SNU), Seoul, Korea. She received B.S. degree in computer science from Sookmyung Women's University, Seoul, Korea. Her main research interests are operating systems, distributed file systems and high performance storage systems.

Hyeonsang Eom received the B.S. degree in computer science and statistics from Seoul National University (SNU), Seoul, Korea, in 1992, and the M.S. and Ph.D. degrees in computer science from the University of Maryland at College Park, Maryland, USA, in 1996 and 2003, respectively. He is currently an associate professor in the Department of Computer Science and Engineering at SNU, where he has been a faculty member since 2005. He was an intern in the data engineering group at Sun Microsystems, California, USA, in 1997, and a senior engineer in the Telecommunication R&D Center at Samsung Electronics, Korea, from 2003 to 2004. His research interests include distributed systems, cloud computing, operating systems, high performance storage systems, energy efficient systems, fault-tolerant systems, security, and information dynamics.