MLB: A Memory-aware Load Balancing for Mitigating Memory Contention

Dongyou Seo, Hyeonsang Eom and Heon Y. Yeom
Department of Computer Science and Engineering, Seoul National University

Abstract

Most of the current CPUs have not single cores, but multicores integrated in the Symmetric MultiProcessing (SMP) architecture, which share the resources such as Last Level Cache (LLC) and Integrated Memory Controller (IMC). On the SMP platforms, the contention for the resources may lead to huge performance degradation. To mitigate the contention, various methods were developed and used; most of these methods focus on finding which tasks share the same resource assuming that a task is the sole owner of a CPU core. However, less attention has been paid to the methods considering the multitasking case. To mitigate contention for memory subsystems, we have devised a new load balancing method, Memory-aware Load Balancing (MLB). MLB dynamically recognizes contention by using simple contention models and performs inter-core task migration. We have evaluated MLB on an Intel i7-2600 and a Xeon E5-2690, and found that our approach can be effectively taken in an adaptive manner, leading to noticeable performance improvements of memory intensive tasks on the different CPU platforms. Also, we have compared MLB with the state of the art method in the multitasking case, Vector Balancing & Sorted Co-scheduling (VBSC), finding out that MLB can lead to performance improvements compared with VBSC without the modification of timeslice mechanism and is more effective in allowing I/O bound applications to be performed. Also, it can effectively handle the worst case where many memory intensive tasks are co-scheduled when non memory intensive ones terminate in contrast to VBSC. In addition, MLB can achieve performance improvements in CPU-GPU communication in discrete GPU systems.

1 Introduction

The current OS scheduler such as Linux balances the load of the runnable tasks across the available cores for balanced utilization among multiple CPU cores. For this purpose, the scheduler migrates tasks from the busiest runqueue to the idlest runqueue just by using the CPU load metric for task. However, the load balancing method ignores the fact that a core is not an independent processor but rather a part of an on-chip system and therefore shares some resources such as Last Level Cache (LLC) and memory controller with other cores [1]. The contention for shared resources is a very important issue in resource management. However, there is no mechanism in the scheduler for mitigating the contention.

Among the shared resources, the contention for memory subsystems makes a big impact on the overall system performance [1][2]. In an article, it is anticipated that the off-chip memory bandwidth will often be the constraining resource for system performance [3]. Even though it is expected that the memory bandwidth is a major bottleneck in multicore CPUs, the number of cores continues to increase. For example, 8 cores on a Xeon E5-2690 introduced in 2012 share 51.2GB/sec memory bandwidth, where 12 cores on a Xeon E5-2695v2 introduced in 2013 share 59.7GB/sec memory bandwidth [4]. Although the number of cores increased by 50%, the memory bandwidth increased only by about 16%. The more physical cores are integrated, the more intensive contention for memory subsystems can occur.

The more serious contention is assumed to happen especially in the server environment. The evolution of hardware and virtualization technology enables many tasks to be consolidated in a CPU platform. Task consolidation is considered as an effective method to increase resource utilization. But, the concurrent execution of many tasks can cause serious contention for shared resources, and hence lead to performance degradation, counteracting the benefit of task consolidation [1][2][5].

Also, the dynamicity of resource demand leads to difficulty in efficient resource management in the server environment, intensifying the contention for shared resources. The server environment is characterized by a

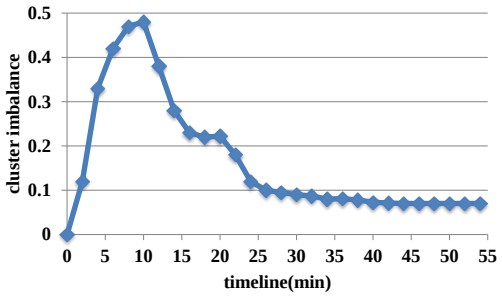


Figure 1: VMware cluster imbalance

large dynamic range of resource demands with high variations over short time intervals [6][7]. It increases the complexity of resource assignment and complicates task migration. Thus, the physical nodes are not always balanced in such an environment. Figure 1 presents the timeline of VMware cluster imbalance [8], which is defined as the standard deviation over all N_h values (N_h = *Resource demand*$_{host\ node}$ / *Resource capacity*$_{host\ node}$). The high imbalance value indicates that resource demands are not evenly distributed among physical nodes. The level of cluster imbalance multiplies ten times ($0.05 \rightarrow 0.5$) from 5 to 20 minutes. That means specific host nodes are seriously overloaded. The imbalance is irregular and prediction is very difficult [7][8].

Most of the studies on mitigating the contention for shared resources have been conducted on the assumption that a task is the sole owner of a CPU core. These studies concentrate on defining the task classification and deciding which tasks share the same resource considering the singletasking case [1][5]. However, we should also consider the multitasking case. The server environment is characterized by a large dynamic range of resource demands with high variations over short time intervals [6]. It increases the complexity of resource assignment and complicates task migration. and hence the physical nodes are not always balanced in the server environment. This imbalance can prevent running tasks fromg being the sole owners of CPU cores. In order to mitigate contention for shared resources in the multitasking, we have developed an inter-core load balancing method called Memory-aware Load Balancing (MLB). This method uses simple models which dynamically recognize memory contention, and mitigates it. MLB periodically monitors the level of memory contention, and migrates memory intensive tasks into specific cores and hence decreases simultaneous memory requests because memory intensive tasks in the same core would never be scheduled at the same time. The focus of our method lies on long-running and compute-bound workload in common with other previous approaches.

We have evaluated MLB on existing SMP platforms which have Uniform Memory Access (UMA) architecture and compared MLB with the state of the art method considering the multitasking case, Vector Balancing & Sorted Co-scheduling (VBSC) [2]. Through this comparision, we conclude that MLB can lead to throughput improvements for I/O bound applications compared with VBSC (for the TPC-C benchmark by about 300% and for the TPC-B one by about 27%). Also, it can more effectively hadle the worst case by improving the performance of a memory intensive task, soplex, by about 30% compared with VBSC.

The primary contributions of this paper are the following:

- We have devised simple models effectively by indicating the level of memory contention and predicting the maximum number of the memory intensive tasks spreading on multiple cores, each on a core in order to decide the appropriate moment when MLB should be triggered.
- We have developed the MLB algorithm which fulfills dynamic inter-core task migration and integrated our algorithm into in the CFS scheduler which is the current Linux default scheduler and ported VBSC in the CFS scheduler because VBSC is based on an old CPU scheduler, the O(1) scheduler. Lastly, we have compared MLB with the ported VBSC. Our attempt is made for the first time to compare the methods for mitigating memory contention in the multitasking case.
- We have evaluated MLB on existing SMP platforms, which are widely being used in practice. We have shown that MLB can be applied to two different CPU platforms, i7-2700 and Xeon E5-2690, leading to noticeable improvements in the overall system performance.
- In addition, we have shown that MLB can also achieve performance improvements in heterogeneous systems consisting of CPU and GPU cores executing CPU and GPU applications simultaneously where the GPU applications do not utilize the full memory bandwidth, by enhancing CPU-GPU communication.

The rest of the paper is organized as follows: Section 2 discusses the related work. Section 3 presents the result of quantifying the effect of memory contention, our memory contention model, and the correlation between the model and performance. Section 4 describes our load balancing algorithm. Section 5 shows the experimental results obtained by comparing our method with the previous approach. Section 6 concludes this paper.

2 Related Work

Zhuravlev et al.[1] proposed a pain classification, and Distributed Intensity (DI) which is considered as the best method in terms of practicality and accuracy, and compared DI with other methods for mitigating the contention for memory subsystems in the singletasking case for the first time. In their method, a task has two scores, sensitivity and intensity. The higher locality the task has, the higher sensitivity score does the task get. The locality of shared cache is measured by using the stack distance profile and miss rate heuristic. The intensity is defined by the number of LLC references per one million instructions. Their method avoids co-locating a high sensitive task with a high intensive task.

The methology proposed by Kim et al.[5] is similar to Zhuravlev's classification. But, their classification and scheduling algorithm are much simpler to classify many tasks and stronger to deal with them. Although the task classification methodologies are effective to the management of shared resources, most of previous methodologies are not applicable in the multitasking case in contrast to ours.

A few methodologies have been devised to mitigate the contention for shared resources in the multitasking case. Merkel et al.[2] and Knauerhase et al.[9] introduced vector balancing & sorted co-scheduling and OBS-M & OBS-L, respectively, to deal with the scheduling issue in the multitasking case. Their methodologies decrease the number of simultaneous memory requests by deciding the order of tasks per runqueue. Knauerhase's method distributes the tasks according to cache weights (#s of LLC misses per cycle) and keeps the sum of the cache weights of all the running tasks close to a medium value. When a new task is ready to be allocated to a core, their scheduler checks the weights of tasks on other cores and co-schedules a task whose weight best complements those of the co-running tasks. Merkel's method is better than Knauerhase's method in various aspects. It motivated our research, which is explained in detail and compared with our method in Section 5.4.

Blagodurov et al.[10] considered the NUMA architecture by mitigating memory contention and presented Distributed Intensity NUMA Online (DINO). Their method pointed out that NUMA-agnostic methods fail to eliminate memory-contention and create additional interconnect contention, introducing remote latency overhead. DINO migrates the thread's memory along with the thread to avoid remote accesses, and prevents superfluous thread migrations which incur high cost overhead. Also, Dashti et al.[11] described an algorithm that addresses traffic congestion on NUMA systems. Note that our method is desinged only for UMA systems at this point of time. We plan to extend our mehtod for the bigger systems which have NUMA architectures in future.

3 Memory Contention Modeling

MLB can mitigate the memory contention by migrating memory intensive tasks to specific cores. However, MLB should not migrate such tasks if there is no contention. Thus, MLB should first check whether or not the memory contention occurs. In the case of memory contention, MLB should measure how intensive the contention is to decide whether or not to migrate memory intensive tasks. In this section, we present a simple and effective memory contention model, and also show the correlation between our model and the performance degradation caused by memory contention to demonstrate the usefulness of the model.

3.1 Memory contention level

There is no single dominant component contributing to the contention for memory subsystems and several components play an important role [12]. Figure 2 shows the average memory bandwidth and memory request buffer full rate with respect to the number of stream applications [13] on an E5-2690. Intel provides off-core response events which can permit measuring memory bandwidth and memory request buffer full rate [14]. The retired memory traffic is the number of memory accesses generated by LLC miss events and prefetcher requests. The memory bandwidth is the amount of retired memory traffic multiplied by 64bytes (size of a cacheline) [15]. The retired memory traffic is a good metric to monitor the overall utilization of memory subsystems including LLC, prefetcher and memory controller and all of those components play important roles for memory contention. The memory bandwidth of stream application is invariable. This means that the memory subsystems can be steadily stressed during the execution time of stream application and the stress degree can be adjusted with respect to the number of running stream applications concurrently accessing memory subsystems.

Memory bandwidth does not increase linearly as the number of stream applications increases. The bandwidth is saturated at a constant level, which is near the maximum memory bandwidth of the E5-2690 (about 50GB/sec). In contrast to the saturated memory bandwidth, the memory request buffer full rate increases quadratically as the number of stream applications grows. A memory request buffer full event indicates a wasted cycle due to the failure in enqueuing a memory request when the memory request buffer is full. As the number of stream applications increases, more memory requests are simultaneously generated while the number of failures increases because the capacity of mem-

ory request buffer is limited. The memory bandwidth and memory request buffer full rate shown in Figure 2 are roughly symmetric with respect to the y=$a \times x$ + b line. The more gentle the inclination of memory bandwidth curve gets, the more quadratic does the inclination of memory request buffer full rate become.

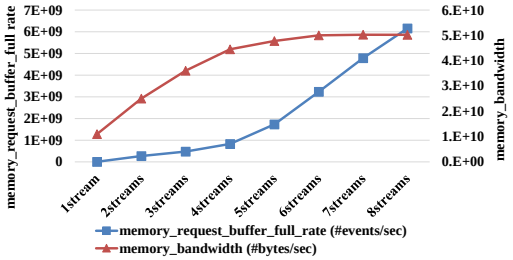


Figure 2: The correlation between memory request buffer full rate and memory bandwidth with respect to the number of stream applications

We constructed our memory contention model based on the correlation between memory bandwidth and memory request buffer full rate as seen in Figure 2. The correlation can be used in general, independent of the application, because the same sorts of memory events happen leading to the correlation regadless of the application.

Equation (1) shows our model. The memory contention level is proportional to the number of retries to make a memory request retire. A high memory contention level indicates a lot of retries because many tasks compete in enqueuing their memory requests into the buffer and hence the memory request buffer is often full. Also, many retries imply the high contention for overall memory subsystems because the retired memory traffic is closely related to LLC, prefetcher and integrated memory controller. The number of memory request buffer full events per memory request retirement event reflects the number of retries to make a memory request retired. We thus divide the request buffer full rate by the retired memory traffic rate. The utilized memory traffic should not be considered as an indicator of memory contention. Our model represents the efficency in memory access to indicate the degree of memory contention.

$$Memory\ Contention\ Level_{system\_wide} = \frac{Memory\ Request\ Buffer\ full\ rate}{Retired\ memory\ traffic\ rate} \quad (1)$$

3.2 Correlation between memory contention level and performance

To evaluate the correlation between the memory contention level and performance, we performed stress tests for memory subsystems similar to those described in [16]. We designated a stream application as a stressor because a stream application has high memory intensity and the memory intensity of a stream is invariable from start to end. A stream application can impose a certain amount of pressure to the memory subsystems at runtime. Thus, we measured the memory contention level and performance of each target application while increasing the number of stressors. We used all SPEC CPU applications [17]. The memory bandwidth for each target application is different (see the value in the x-axis for the point labeled solo in Figure 3). We did the stress tests on both an i7-2600 and an E5-2690. The specifications of our SMP CPU platforms are shown in Table 1. Due to the lack of space, we present the results of six applications per CPU platform in this paper. The reason why we selected those applications is that the application set can show the apparent degradation diffrences between memory intensive applications and non intensive applications due to memory contention. To stress memory subsystems during the entire execution time of each target application, the stressors continued to run until the target application terminated.

The memory contention level is a system-wide metric. It indicates the level of overall contention in the CPU platform. We need to precisely figure out the correlation between the level and performance of target application because the sensitivity of each application to the contention for memory subsystems is different. We thus use the sensitivity model presented in Kim et al.[5]. This sensitivity model is very simple, but it is effective and powerful. Equation (2) shows the sensitivity model. The sensitivity model considers the reuse ratio of LLC ($LLC_{hit}\ ratio$) and the stall cycle ratio affected by the usage of memory subsystems. We calculated the predicted degradation of target application multiplying the sensitivity of each application by the memory contention level increased by both the target application and stressors as seen in Equation (3).

The results are shown in Figure 3. The blue vertical line (left y-axis) of each graph indicates the runtime normalized to the sole execution of the target application and the red line (right y-axis) shows the predicted degradation calculated by Equation (3). The x-axis indicates the system-wide memory bandwidth. The predicted degradation is fairly proportional to the measured degradation (normalized runtime). As the predicted degradation increases, the measured degradation accordingly increases on all CPU platforms. The memory bandwidth of the E5-2690 increases as the number of stressors grows. The highest memory bandwidth reaches the maximum memory bandwidth (about 50GB/sec) [18] when the number of stressors is 5 or 6.

In contrast, the highest memory bandwidth of the i7-

Descriptions	Xeon E5-2690	i7-2600
# of cores	8	4
Clock speed	2.9GHz (Turbo boost: 3.8GHz)	3.4GHz (Turbo boost: 3.8GHz)
LLC Capacity	20MB	8MB
Max memory bandwidth	51.2GB/sec	21GB/sec
CPU Category	Server level CPU	Desktop level CPU
Microarchitecture	Sandy-bridge	Sandy-bridge

Table 1: Specifications of our SMP CPU platforms

2600 reaches the maximum memory bandwidth (about 21GB/s) [19] when the number of stressors is 1 or 2. In the cases of executing lbm, libquantum, soplex and GemsFDTD, the memory bandwidth decreases when each target application is executed with three stressors. The memory contention levels of the applications with three stressors are much higher than that of the non-memory intensive applicationa which are omnetpp and tonto (lbm:11.6; libquantum:10.1; soplex:10.88; GemsFDTD:9.55; omnetpp:7.66; tonto:7.68). The results imply that the system-wide memory bandwidth can be decreased by too many retries in enqueuing memory requests into the buffer. The contention in an i7-2600 can be drastically mitigated because the memory bandwidth of the i7-2600 is low, thus more easily saturated than in the case of the E5-2690. The results demonstrate that memory contention level effectively indicates the contention degree closely correlated with the performance of target application.

$$Sensitivity = (1 - \frac{LLC_{miss}}{LLC_{reference}}) \times \frac{Cycle_{stall}}{Cycle_{retired}} \quad (2)$$

$$\begin{aligned} &Predicted\ Degradation_{target\_application} \\ &= Memory\ Contention\ Level_{system\_wide} \\ &\times Sensitivity_{target\_application} \end{aligned} \quad (3)$$

4 Design and Implementation of Memory-aware Load Balancing (MLB)

To implement MLB, we have to solve the follwing questions: Which task should MLB consider as a memory intensive task? When should MLB be triggered? How many memory intensive tasks should MLB migrates? To which cores should MLB migrate memory intensive tasks? In this section, we address these design and implementation issues in detail.

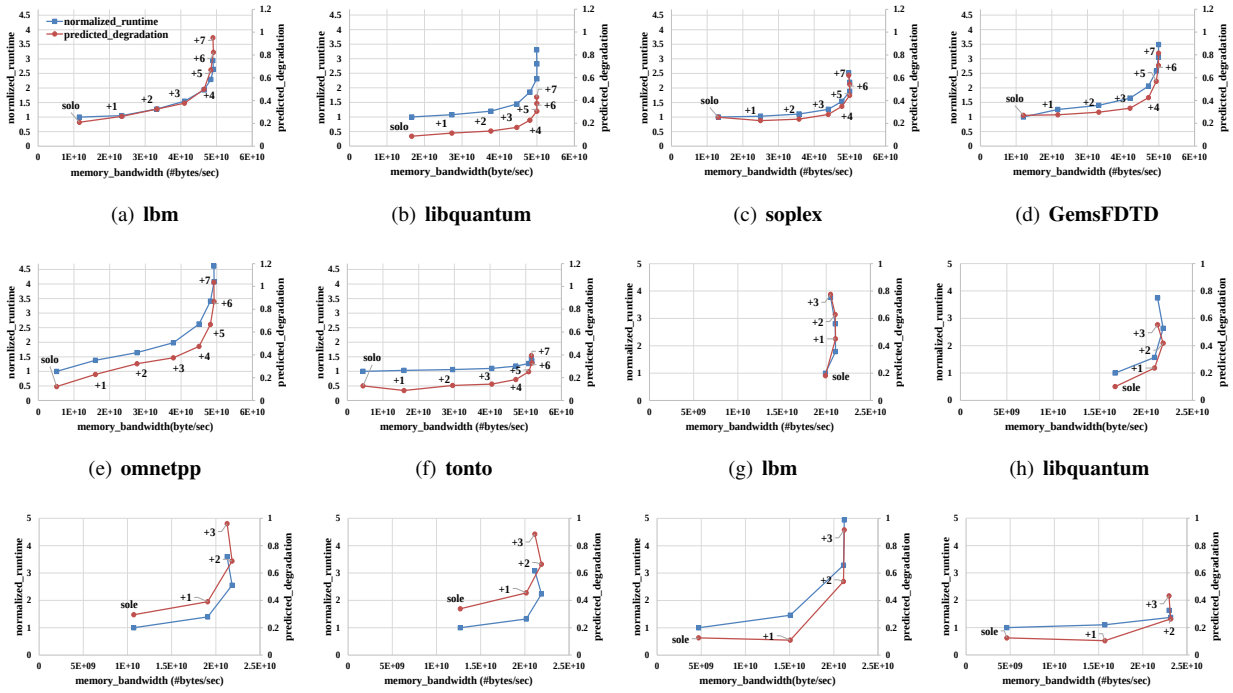
4.1 Intensive runqueue and non intensive runqueue lists

MLB should dynamically divide tasks into memory intensive task and non memory intensive task sets. MLB monitors the retired memory traffic (off-core response events [14]) generated by each task and accumulates the memory intensity into the corresponding task structure whenever the cputime of the task expires (at context switch). MLB predicts future usage patterns of memory subsystems from the behavior history. To check the accuracy of the profiled intensity value with MLB, we compared the results of Intel Performance Counter Monitor (PCM) with the traffic values monitored by MLB. We found that the difference in result between PCM and MLB is negligible (under 5%). MLB considers the tasks whose memory intensity is higher than the average memory intensity of all tasks as memory intensive tasks when the memory contention is higher considering the number of intensive runqueues (or when the memory contention is intensive).

Also, MLB should determine which core memory intensive tasks are migrated to. For this purpose, MLB manages two runqueue lists, intensive runqueue list and non intensive one. There is one runqueue per core and a memory intensive task can be scheduled only on a core whose runqueue is designated as an intensive one. In contrast, a non memory intensive task can be scheduled only on a core whose runqueue is designated as a non intensive one. But, MLB can migrate the non memory intensive task that has the highest intensity to an intensive runqueue when the memory contention is intensive. As a result, there will be eventually no memory intensive tasks in the non intensive runqueues. In this way, MLB performs the task migration possibly not based on the features of intensive runqueue and non intensive runqueue; MLB can dynamically select a core and mark its runqueue as intensive one or non intensive one depending on that memory contention level.

4.2 Predicted number of intensive runqueues

MLB should find out when memory intensive tasks are migrated to intensive runqueues. We have constructed a model for the predicted number, which is the anticipated number of memory intensive tasks which are to be scheduled on each core at the same time. MLB can dynamically calculate the memory contention level, but what MLB also needs to figure out is the number of untreated memory intensive tasks to be migrated. If the number of intensive runqueues is smaller than the predicted number, MLB decides that there is a memory intensive task out of intensive runqueues. To build a model for the predicted



(a) **lbm** (b) **libquantum** (c) **soplex** (d) **GemsFDTD**

(e) **omnetpp** (f) **tonto** (g) **lbm** (h) **libquantum**

(i) **soplex** (j) **GemsFDTD** (k) **omnetpp** (l) **tonto**

Figure 3: Correlation between memory contention level and performance with respect to memory bandwidth. Figures (a) - (f) show the results for a Xeon E5-2690 and (g) - (l) those for an i7-2600, respectively

number, we used the fact that memory contention increases quadratically. Figure 4 presents the rising curve of memory contention level as the number of stream applications increases. We computed memory contention levels for four different CPU platforms. All rising curves of memory contention level become steeper as the number of stream application increases. Each curve is similar to that of $y = a \times x^2 + b$ where the x-axis shows the number of stream applications and the y-axis indicates the memory contention level. We have constructed our model based on the curve of $y = a \times x^2 + b$. Equation (4) shows the model with respect to the memory contention level.

$$\begin{aligned} &\textit{Predicted Number} \\ &= [\sqrt{\frac{\textit{Memory Contention Level} - \textit{Baseline}}{\textit{Inclination}}}] \end{aligned} \tag{4}$$

$$\left(\begin{array}{c} \textit{Inclination} \propto \frac{1}{\textit{Max Memory Bandwidth}} \\ \textit{Baseline} = \textit{Memory Contention Level Baseline} \end{array} \right)$$

We rotated the curve of $y = a \times x^2 + b$ on the $y = x$ axis of symmetry, and then replace y, a and b with the memory contention level, inclination and baseline, respectively. The inclination is inversely proportional to the maximum memory bandwidth of the CPU platform. The inclination for the CPU platform which has the lower maximum memory bandwidth is steeper than that of the higher CPU platform as shown in Figure 4. The inclination of i7-2600 is the steepest, but that of E5-2690 or 2670 is relatively gentle. The baseline is the default memory contention level. Although there is no running task, the memory contention level is not zero due to the miscount of hardware performance counter and the execution of background tasks. Lastly, we rounded off to transform the value to a discrete number. We empirically calculated inclination (0.77 for an i7-2600 and 0.1 for an E5-2690) and baseline (0.54 for an i7-2600 and 0.2 for an E5-2690) by using the results of the stress test as shown in both Figures 3 and 4. This model outputs the approximated number of memory intensive tasks scheduled on each core at the same time during the interval with an input of memory contention level.

MLB mitigates the contention for memory subsystems by limiting the number of the tasks generating high memory traffic. In the mechanism of MLB, all memory intensive tasks can be executed only on the cores whose runqueuea are intensive ones and hence the selected cores can generate high memory traffic, not all cores. When the predicted number is greater than the number of intensive runqueues, MLB decides that there are untreated

memory intensive tasks which are in non intensive runqueues. Then, MLB checks whether to migrate the memory intensive tasks from the non intensive runqueues to intensive runqueues.

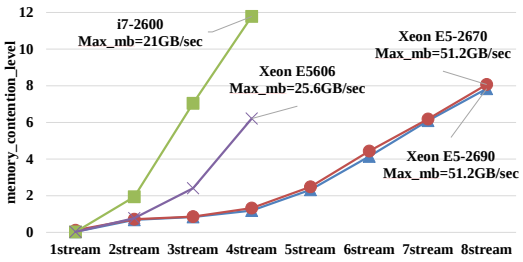


Figure 4: Memory contention levels for four CPU platforms with respect to the number of stream applications

4.3 Memory-aware Load Balacing algorithm

The MLB mechanism is described in Algorithm 1. The key objective of MLB is to keep the number of intensive runqueues no larger than the predicted number. First, two conditions are periodically checked and MLB is triggered when both conditions are met. As described in the previous section, the predicted number is the anticipated number of running memory intensive tasks to be scheduled on each core at the same time. When the predicted number is larger than the number of intensive runqueues (Line 4), MLB is triggered. However, MLB does not immediately perform task-migration. MLB checks the memory intensity (denoted by mem_int) of the highest task (one with the highest memory intensity) in the non intensive runqueues and compares the intensity with that of the lowest task (one with the lowest memory intensity) in the intensive runqueues because the predicted number can be overestimated. The per-task memory intensity is calculated by computing the average value of the retired memory traffic rate generated during the cputime, from each switch-in to the corresponding switch-out. When the intensity of the highest task is no higher than that of the lowest one, MLB decides that all memory intensive tasks are already migrated to intensive runqueues, and hence bypasses (Line 11). This situation is considered as a false positive due to the overestimation (misprediction). But, checking the intensity of the highest task can prevent MLB from performing unnecessary task migration; MLB will be triggered when the intensity of the highest task in the non intensive runqueues is higher. This means that at least the highest task should be migrated to an intensive runqueue.

MLB may swap memory intensive tasks in non intensive runqueues with non memory intensive tasks in intensive runqueues while preserving the total core utilization because it can only swap two tasks whose core utilization are similar. When there is no intensive runqueue including any non memory intensive task in intensive runqueue list (Line 14), MLB inserts the highest runqueue into the intensive runqueue list (a set of intensive runqueues), which has the highest intensity sum for all the tasks in the non intensive runqueue to minimize the number of task swapping (Lines 15 and 24), and resumes swapping. MLB determines the non intensive runqueue including the highest task as the source runqueue (Line 9). The memory intensive tasks in the source runqueue are swapped with non memory intensive tasks in intensive runqeueus (Line 39). As long as a memory intensive task resides in the source runqueue, MLB continues to conduct task swapping and insert a non intensive runqueue to the intensive runqueue list if needed (Lines 24 and 25). MLB also removes the lowest runqueue which has the lowest intensity sum of all the tasks in an intensive runqueue, if the predicted number is smaller than the number of intensive runqueues (Lines 5 and 6). Also, MLB algorithm can be illustrated as in Figure 5.

The number of intensive runqueues can be dynamically adjusted depending on the level of memory contention. MLB essentially migrates the distributed memory intensive tasks among all cores generating simultaneous memory requests to intensive runqueues. As a result, the steps of MLB enforce only the selected cores (intensive runqueues) to generate memory requests via task migration.

5 Evaluation

In this section, we present the results of evaluating MLB. In all our experiments, the memory contention level is calculated every four seconds because four seconds empirically turned out to be the best length of interval for our target workload, compute-bound applications. We present the results of evaluating the ability of MLB to cope with dynamic task creation and termination situations by using our models. Next, we present how much memory contention is mitigated and the performance is thereby improved. Also, we present an additional benefit in CPU-GPU communication. Lastly, we explain the results of comparing our method with VBSC [2].

5.1 Model applicability

Our predicted number model is simple and effective. Based on this model, MLB dynamically adjusts the number of intensive runqueues to control the number of simultaneous memory requests, and therefore MLB can

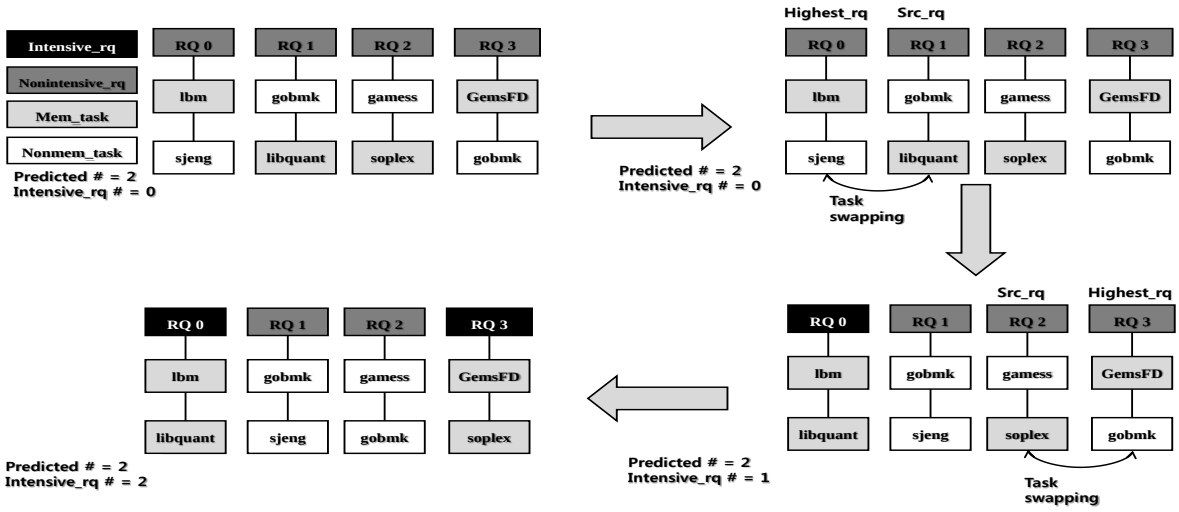


Figure 5: MLB mechanism

Algorithm 1 Memory-aware Load Balancing Algorithm

```
1:  Memory_aware_Load_Balancing() begin
2:  predicted_number = get_predicted_number(memory_contention_level)
3:  avg_int = calculate_average_int(all_tasks)
4:  if predicted_number < intensive_rq_list → count then
5:      lowest_rq = find_the_lowest_rq(intensive_rq_list)
6:      add_rq_to_list(lowest_rq, non_intensive_rq_list)
7:      return
8:  end if
9:  src_rq = find_the_rq_having_highest_task(non_intensive_rq_list)
10: lowest_task = find_the_lowest_task(intensive_rq_list)
11: if src_rq → highest_task → mem_int ≤ lowest_task → mem_int then
12:     return
13: end if
14: if intensive_rq_list → count = 0 then
15:     highest_rq = find_the_highest_rq(non_intensive_rq_list)
16:     add_rq_to_list(highest_rq, intensive_rq_list)
17: end if
18: dest_rq = intensive_rq_list → head
19: for each task in src_rq do
20:     if avg_int ≥ src_rq → highest_task → mem_int then
21:         break
22:     end if
23:     if dest_rq = NULL then
24:         highest_rq = find_the_highest_rq(non_intensive_rq_list)
25:         add_rq_to_list(highest_rq, intensive_rq_list)
26:     end if
27:     if dest_rq = src_rq then
28:         break
29:     end if
30:     i = 0
31:     while i < intensive_rq_list → count do
32:         if avg_int < dest_rq → lowest_task → mem_int then
33:             dest_rq = dest_rq → next
34:             i = i + 1
35:         else
36:             break
37:         end if
38:     end while
39:     swap_two_tasks(src_rq, src_rq → highest_task,
                       dest_rq, dest_rq → lowest_task )
40: end for
```

efficiently deal with the dynamic task creation and termination without performing complex procedures. In this section, we show the result of evaluating the effectiveness of our model and the ability of MLB to deal with the dynamicity. We performed experiments in the scenario where memory intensive tasks are dynamically created and killed to evaluate MLB mechanism. Figure 6 shows the results for an E5-2690. All CPU intensive tasks (namd, sjeng, gamess and gobmk) were executed from the beginning, while the start times for memory intensive tasks (lbm, libquantum, soplex and GemsFDTD) were different.

As shown in Figure 6 (a) with MLB, 2lbms started at 30 secs, 2libquantums at 60 secs, 2soplexs at 90 secs and 2GemsFDTDs at 120 secs. After 2lbms started, MLB detected memory contention so that 2lbms could be migrated to core 5. The predicted number was simultaneously calculated by logging the footprints of memory intensive tasks from the beginning as shown in Figure 6 (b). We used a time window consisting of 19 previous predicted numbers and 1 current predicted number and calculated the average of the 20 numbers as the final predicted number because the memory intensity of task was not always invariable unlike that of stream application. The window has a role in smoothing the predicted number and preveting it from fluctuating, but the overlarge window can make MLB untriggered when memory contention is intensive.

Although only CPU intensive tasks were running before 2lbms started, the memory contention (the predicted number two) due to the CPU intensive tasks occurred and hence there were already two intensive runqueues at the time when 2lbms started. Compared with the singletasking case, each lbm shared time with the co-running CPU intensive task in this case and hence 2lbms were not al-

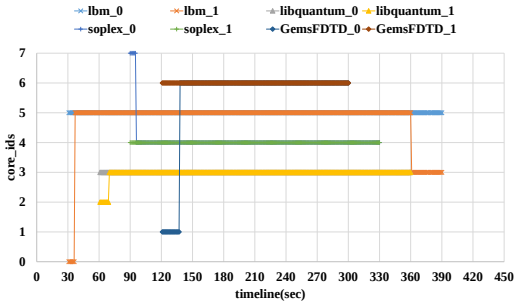
ways scheduled at the same time. Thus, the predicted number was not increased. Likewise in the lbm's case, MLB migrated 2libquantums to core 3 due to the increase of memory contention after 2libquantums started, and the predicted number increased to 3. The number of intensive runqueues increased as the predicted number in accordance with memory contention level increased. After all memory intensive tasks started, MLB managed four intensive runqueues (cores 3, 4, 5 and 6) by changing the number of cores generating high memory traffic to up to four.

If memory intensive tasks dynamically terminated, then MLB decreased the number of intensive runqueues. 2GemsFDTDs ended at 300 secs, 2soplexs at 330 secs, 2libquantums at 360 secs and 2lbms at 390 secs while all CPU intensive tasks terminated at 420 secs. The number of intensive runqueues decreased as the predicted number decreased. In particular, there was no task on core 3 after 2libquantums terminated, and hence MLB randomly migrated a task to core 3 in order to balance the CPU load among all cores. In this scenario, one lbm was selected and migrated from core 5 to core 3. Each lbm was the sole owner of a core from 360 to 390 secs. Although the contention could be further mitigated when 2lbms were migrated to a core, there was no memory intensive task migration from 360 to 390 secs because MLB did not migrate any task that was the sole owner of a core, being applied in harmony with the existing methods considering the singletasking case.

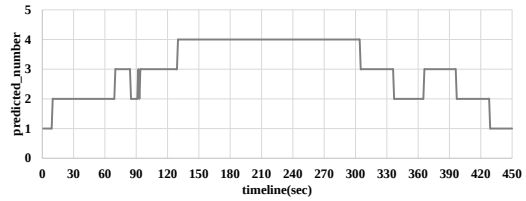
Although the predicted number is overestimated in some interval (from 360 to 390 secs), MLB does not perform wasted migrations due to our prevention mechanism as mentiond in Section 4.3 (Line 11). MLB effectively handles the underestimated memory contention. The worst case is the case where MLB does not migrate memory intensive tasks to intensive runqueues although the memory contention continues to occur. MLB immediately responds to the creation and termination of the tasks which cause the memory contention, based on our simple and applicable model as shown in Figure 6 (a).

5.2 Overall system performance

The performance results are presented in Figure 7. We executed the applications in two mixes for an E5-2690 more than ten times and normalized the average execution times measured with MLB to those with the naive scheduler of Linux kernel 3.2.0. The mixes are organized in Table 2 The results for an i7-2600 will be presented in Section 5.4. All benchmarks in the workload were launched simultaneously and if each benchmark except for the last terminating one terminated it was immediately restarted so that there were all the other benchmarks running when the last one terminated. We sampled the



(a) **Footprints of memory intensive tasks with MLB**



(b) **Timeline of predicted number**

Figure 6: The scenario of dynamic task creation and termination

first runtime of each benchmark to evaluate the performance. The number of memory intensive tasks in workload mix1 is twice larger than that for workload mix2. In other words, the memory contention level in workload mix1 is higher. All memory intensive applications benefited from MLB in both workload mixes.

The memory requests buffer full rate decreased and the memory bandwidth increased similarly in both cases of mix1 and mix2 because the test sets in workload mix1 and mix2 could not saturate the maximum memory bandwidth of E5-2690 as shown in Figure 7 (a). The reduced ratio of memory request buffer full rate and improved ratio of memory bandwidth reflected the performance improvements with MLB. However, the performance degradation for non memory intensive tasks occurs due to the increases in LLC reference and miss, but it is relatively negligible, being under 2%, when compared with the performance gain for memory intensive applications as shown in Figure 7 (b) and (c). MLB could achieve the noticeable improvements in the overall system performance.

5.3 Additional benefit in CPU-GPU communication

The CUDA programming model using GPU permits data transfer from host to GPU or vice versa to offload the portion for which the use of GPU is appropriate. In the programming model, CPU-GPU commu-

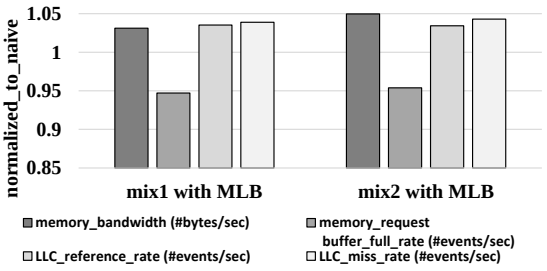
Names of workload mixes	Workloads (C: CPU-bound, M: Memory-bound)
Workload-mix1 (for an E5-2690)	2lbm(M), 2libquantum(M), 2GemsFDTD(M), 2soplex(M), 2namd(C), 2sjeng(C), 2gamess(C), 2gobmk(C)
Workload-mix2 (for an E5-2690)	2lbm(M), 2libquantum(M), 2tonto(C), 2zeusmp(C), 2namd(C), 2sjeng(C), 2gamess(C), 2gobmk(C)
Workload-mix1 (for an i7-2600)	lbm(M), libquantum(M), GemsFDTD(M), soplex(M), namd(C), sjeng(C), gamess(C), gobmk(C)
Workload-mix2 (for an i7-2600)	lbm(M), libquantum(M), tonto(C), zeusmp(C), namd(C), sjeng(C), gamess(C), gobmk(C)

Table 2: Workload mixes (SPEC CPU2006)

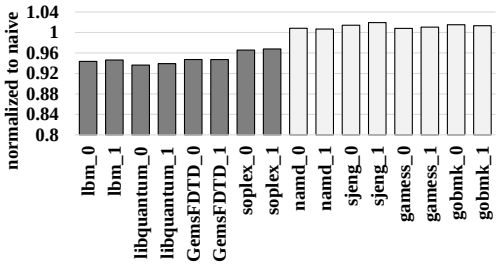
nication is very important for the performance of GPU application [20]. However, the performance of CPU-GPU communication can be degraded by co-located applications because both CPU and GPU applications may share host memory bandwidth in discrete GPU systems [21]. In a high memory contention situation, a GPU application may not obtain sufficient memory bandwidth. MLB can indirectly allow the GPU application to occupy more memory banwidth. We have evaluated MLB with NVIDIA GTX580 and Tesla K20C for an i7-2600 and for an E5-2690, respectively. We executed a GPU application, stremcluster of rodinia benchmark suite [22] and used nvprof to measure the GPU kernel time and the time spent doing communication between CPU and GPU. We replaced a soplex with a streamcluster in workload mix1 for both an i7-2600 and an E5-2690 and compared MLB with the naive scheduler as shown in Figure 8. MLB decreased the CPU-GPU communication time in both CPU platforms more than the naive one while having the kernel times not changed in all cases. The result demonstrates that MLB can lead to improvements in the performance of CPU-GPU communication via the mitigation of memory contention when the memory contenion is intensive.

5.4 Comparison with the VBSC strategy

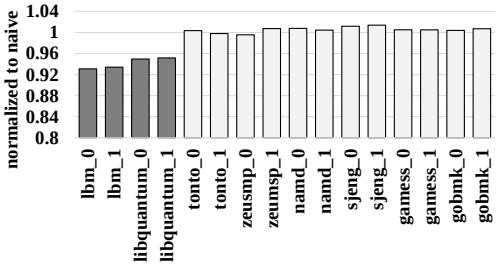
Zhuravlev et al. [1] already made an attempt to compare the methods for mitigating memory subsystems in the singletasking case while they did not compare the



(a) **Performance metrics**



(b) **Execution time for workload-mix1**



(c) **Execution time for workload-mix2**

Figure 7: Comparisons in SPEC CPU2006 applications between MLB and the naive scheduler for an E5-2690

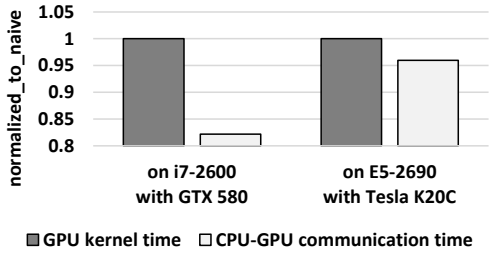


Figure 8: Improvements in CPU-GPU communication with MLB (Streamcluster)

methods in the multitasking case. In this section, we show the result of comparing MLB with the state of the art method, Vector Balancing & Sorted Co-scheduling (VBSC), for mitigating memory contention in the multi-

tasking case. VBSC distributes tasks to maximize the vector deviation of each runqueue (we can regard the vector as the intensity of memory subsystems). Both a memory intensive task and a CPU intensive task co-exist in a runqueue. The task distribution is essentially vector balancing. Sorted co-scheduling determines the order of the tasks to be executed. Their method decides the task order to execute in each runqueue and selected tasks are scheduled in the same order. That means that the number of memory intensive tasks running in the same order can be adjusted and their method fixes the number of simultaneous memory requests. It is similar to MLB that their method limits the number of memory intensive tasks running at the same time. In this section, we explain our VBSC implementation ported to the current version of Linux kernel and compare MLB with the ported VBSC.

5.4.1 VBSC implementation

To compare MLB with VBSC, we ported the VBSC code obtained from the authors to the current default Linux scheduler, Completely Fair Scheduler (CFS), which determines the order of the tasks to be executed by using the per-task vruntime sorted in red-black tree. We tried to replace the red-black tree with two queues, active and expired runqueues, at first because the original VBSC is based on the old Linux scheduler, which is the O(1) scheduler using the two queues. However, the replacement of runqueue structure incurred very high implementation overhead; hence, we designed a new VBSC version which is applicable to CFS. In our implementation, the per-task vruntime is controlled to order running tasks. The task which has the smallest vruntime is first scheduled in an epoch.

As shown in Figure 9, we sampled per-task expired times in epoch 91 to demonstrate the validity of our implementation. Epoch length is about 600ms and each order lasts for 300ms. Through this sampled data, we can notice that our implementation binds a memory intensive task with a non memory intensive task (vector balancing) to maximize the vector deviation of each runqueue and schedules libquantum and soplex, memory intensive tasks, in the first order, and next it schedules lbm and GemsFDTD in the second order to prevent each memory intensive task from being scheduled in the same order by synchronizing scheduling decisions (sorted co-scheduling). The VBSC mechanism is effectively integrated in the CFS scheduler.

5.4.2 Performance of CPU-bound applications

Merkel et al. [2] claimed that the method such as MLB migrating memory intensive tasks to specific cores pollutes cache lines, concluding that it is not a good approach. However, it may not be true that migrating memory intensive tasks to specific cores is useless on the current SMP platforms which have bigger LLCs and provide the higher maximum memory bandwidth. We executed workload mix 1 for an i7-2600 with both MLB and VBSC. The Intel i7-2600 has a specification similar to that of Core2 Quad Q6600 previously evaluated by Merkel et al. and hence we selected an i7-2600 as the platform. Performance results are shown in Figure 10. In these comparison results, we can notice that MLB leads to as much performance improvement as VBSC does.

```
[   79.904381] epoch: 91
[   80.191422] on core 3, namd is expired
[   80.191425] on core 2, libquantum is expired
[   80.195407] on core 0, soplex is expired
[   80.195411] on core 1, gobmk is expired
[   80.478456] on core 1, lbm is expired
[   80.482442] on core 3, GemsFDTD is expired
[   80.490414] on core 0, gamess is expired
[   80.490417] on core 2, sjeng is expired
[   80.502373] epoch: 92
```

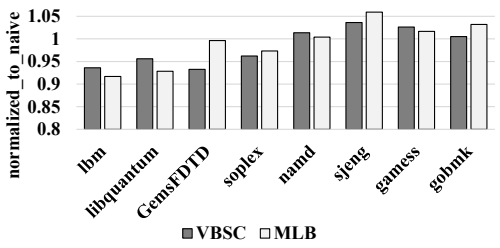
Figure 9: Demonstration of the validity of VBSC implementation

The task group which is scheduled at the same time is different between MLB and VBSC. When MLB schedules two memory intesive tasks, each in an intensive runqueue, we cannot identify which two tasks are scheduled at the same time because MLB does not control allocation of per-task timeslices and does not synchronize the start and expired times. In contrast to MLB, VBSC can decide which two memory intensive tasks should be executed in the same order in an epoch as seen in Figure 9. Thus, the improvement degree of each application is different between MLB and VBSC.

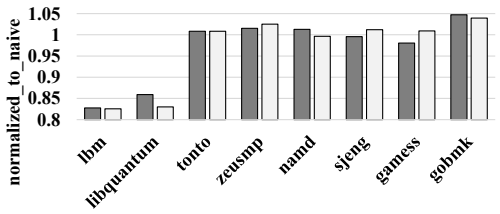
Both MLB and VBSC lead to mitigation of memory contention for workload mix1 and mix2 at a similar level as seen in Figure 10 (c). Based on these results, we can conclude that serious cache pollution concerned by Merkel does not occur on the current CPU platform and MLB also leads to noticeable performance improvements for SPEC CPU 2006 applications, similar to VBSC.

5.4.3 Performance of I/O-bound applications

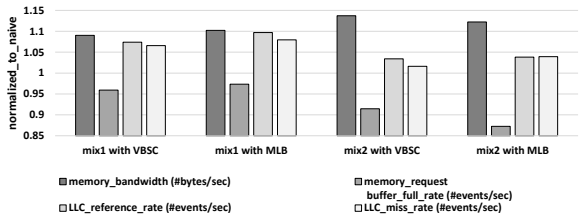
VBSC cannot avoid modifiying the timeslice mechanism for sorting tasks in a runqueue, and hence this scheduling scheme can be in conflict with I/O-bound tasks. I/O-bound tasks typically receive higher priorities than CPU-bound tasks, and are scheduled preferably. However, an I/O-bound task frequently blocks upon an I/O request before the timeslice of the running task expires and unblocks again when the I/O request completes. Thus, the contents of the runqueues constantly



(a) **Excution time for workload-mix1**

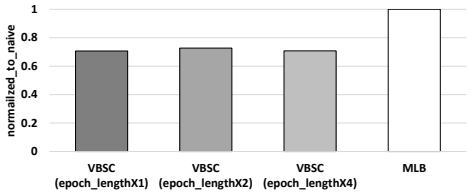


(b) **Excution time for workload-mix2**

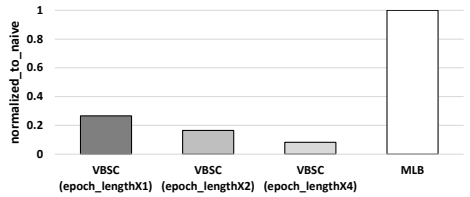


(c) **Performance metrics**

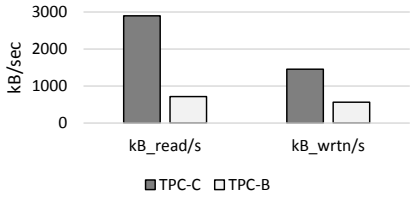
Figure 10: Comparison in performance (SPEC CPU2006)



(a) **TPC-B (TPS)**



(b) **TPC-C (tpmC)**



(c) **I/O intensity (solo execution)**

Figure 11: Comparison in I/O bound applications

change and a sorting approach does not make sense [2]. In contrast to VBSC, MLB can be implemented just by modifying the part of periodical load balancing and hence it can be performed based on the pure timeslice mechanism provided by the CFS scheduler.

We executed workload-mix2 after adding one TPC-B benchmark implemented in the postgresql benchmark (pgbench) or one TPC-C benchmark based on mysql to the mix in order to compare the ability of MLB to deal with I/O bound tasks with that of VBSC. In case of the TPC-B benchmark, 16 clients concurrently made their connections to a postgresql server and Transactions Per Second (TPS) was measured as shown in Figure 11 (a). In case of the TPC-C benchmark, 16 clients connected to a mysql server (warehouse size = 2GB), and Transactions Per Minute (tpmC) was measured as seen in Figure 11 (b).

MLB can allow the I/O bound task to be performed without causing performance degradation compared with the naive Linux scheduler. However, the higher I/O intensity of the application, the more performance degraded with VBSC (The read bandwidth of TPC-C 4 times and write bandwidth 2.6 times than TPC-B). In case of TPC-C benchmark, epoch length (default = 400ms) is a sensitivity parameter. The longer epoch length, the more frequently blocked I/O bound task and thus the performance degradation is more serious. However, sorted co-scheduling cannot be operated with syncronization if epoch length is too smaller. In contrast to TPC-C benchmark, TPC-B one was not relatively effected by epoch length because the I/O intensity of TPC-B is much lower. But, the performance of TPC-B also seriously degraded by about 27% and that of TPC-C by about 300% with VBSC.

5.4.4 Worst case for VBSC

VBSC cannot handle the worst case due to its mechanism. We simulated the dynamic task termination case as the worst case for VBSC. We executed workload-mix1 with both VBSC and MLB and sent SIGKILL signals to the two lowest intensive tasks, namd and sjeng, in 300 secs. As see in Figure 12, the timelines of memory request buffer full rate were monitored to compare in the contention level between VBSC and MLB. With VBSC,

the memory request buffer full rate skyrocketed after the two non intensive tasks terminated, but it did not seriously increase with MLB. This increment was caused by the VBSC mechanism. VBSC binds an intensive task with a non intensive task to maximize the vector deviation of each runqueue. The co-located high intensive tasks became the sole owners of CPU cores when the two non intensive tasks terminated. Then, the co-located high intensive tasks were executed all the time. When other high intensive tasks on other cores were scheduled, all memory intensive tasks were executed on all cores in the same order. Thus, memory contention became more serious. In contrast to VBSC, the high intensive tasks still remained in two intensive runqueues with MLB after the termination and hence there was not a great change in the memory contention because only two intensive tasks could be scheduled before the termination. In this simulation, MLB can lead to an improvement in soplex which was the first finisher with both VBSC and MLB (1,159 secs with VBSC and 795 secs with MLB).

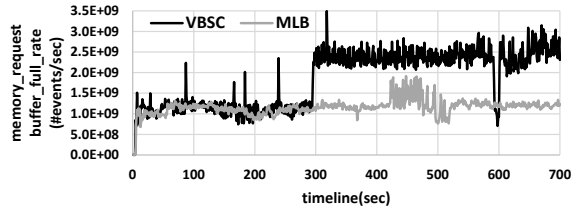
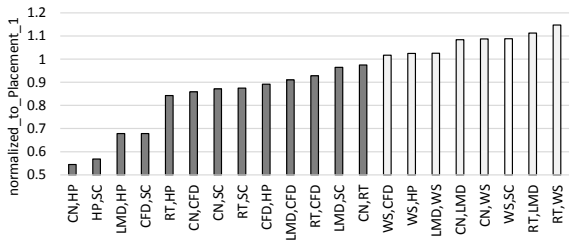


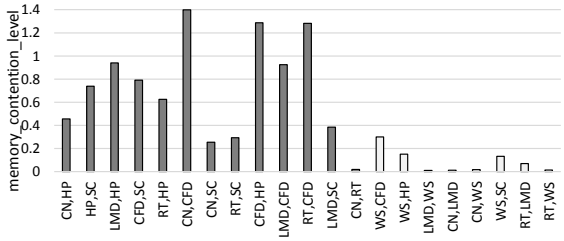
Figure 12: The worst case for VBSC

5.5 Extention of MLB for multi-threaded applications

The approach in which the highly intensive tasks are migrated to specific cores as in MLB can be extended to multi-threaded applications. We used the taskset command and statically executed two multi-threaded applications that respectively created eight threads on eight cores of E5-2690. We made a pair for seven applications (total 21 pairs), canneal and ratrace in PARSEC [23], streamcluster, hotspot, lavaMD and cfd in rodinia [22] and water_spatial in SPLASH2x [23]. All the applications are CPU-bound and long running. For clarity, we abbreviated canneal, raytrace, streamcluster, hotspot, lavaMD, cfd and water spatial to CN, RT, SC, HP, LMD, CFD and WS, respectively. In Placement 1, two applications shared all CPU cores. In Placement 2, each application was run on four private cores. We measured the memory contention level of each application pair in Placement 1 and normalized the average runtime of two applications in Placement 2 to that in Placement 1 as seen



(a) **Average runtime comparisions in multi-threaded applications between Placement 1 and Placement 2**



(b) **Memory contention level in Placement 1**

Figure 13: Performance and memory contention level results with repsect to different co-located workloads in different placements

in Figure 13. Performance in more than half of application pairs improved in Placement 2. Notable is the fact that performance improved in Placement 2 when memory contention level is high (over 0.2) except in the case of water_spacial and cfd. In case of canneal and hotspot, average runtime of the applications can decrease by 46% in Placement 2.

MLB mechanism can reduce the level of the parallelism, but the placement by MLB can decrease the memory contention and increase the cache hit ratio as verified in [24]. The threads of memory intensive application can be only scheduled on four cores in Placement 2. This placement can mitigate the contention for memory subsystems by limiting the number of threads which generates high memory traffic as MLB mechanism does and increase the cache hit ratio, hence leading to performance improvements for multi-threaded applications when the memory contention level is high.

6 Conclusions and Future Work

In this paper, we presented a new load balancing method, Memory-aware Load Balancing (MLB), to mitigate the memory contention in the multitasking case. MLB has the three stronger points than the state of the art method in the multitasking case, Vector Balancing and Sorted Co-scheduling (VBSC). First, MLB mechanism

can be more easily implemented in both user and kernel levels than VBSC without the modification of timeslice mechanism. Second, MLB can allow I/O bound applications to be executed without causing performance degradation unlike VBSC. Lastly, MLB can handle the worst case which can be shown by the VBSC mechanism.

However, there are currently some issues regarding MLB. First, MLB must receive assistance from the global load balancer, which allocates tasks among physical nodes. When there are too many memory intensive tasks in a physical node, all methods including MLB to mitigate the memory contention cannot be used effectively. We plan to design a global load balancer which can be performed in harmony with MLB. Also, MLB should be extended in terms of the fairness for the performance improved by MLB. In figure 10 (a), GemsFDTD rarely benefits from MLB compared with other memory intensive tasks. There is no abstraction to deal with the fairness of the mitigation for memory contention. We plan to improve the fairness of MLB. Lastly, MLB can consider only UMA architecture at this point. We plan to extend MLB on NUMA architecture.

7 Acknowledgement

This work was supported by the National Research Foundation of Korea (NRF) grant NRF-2013R1A1A2064629.

References

[1] Sergey Zhuravlev, Sergey Blagodurov and Alexandra Fedorova. Addressing Shared Resource Contention in Multicore Processors via Scheduling. In ASPLOS, 2010.

[2] Andreas Merkel, Jan Stoess and Frank Bellosa. Resource-conscious Scheduling for Energy Efficiency on Multicore Processors. In EuroSys, 2010.

[3] David Patterson. Latency lags bandwidth. In Communication of the ACM, 2004.

[4] http://ark.intel.com/products/.

[5] Shin-gyu Kim, Hyeonsang Eom and Heon Y. Yeom. Virtual machine consilidation based on interference modeling. In the journal of Supercomputing, April 2013.

[6] Charles Reiss , Alexey Tumanov , Gregory R. Ganger, Randy H. Katz and Michael A. Kozuch. Heterogeneity and Dynamicity of Clouds at Scale: Google Trace Analysis. In SOCC, 2012.

[7] Abhishek Gupta, Osman Sarood, Laxmikant V Kale and Dejan Milojicic. Improving HPC Application Performance in Cloud through Dynamic Load Balancing. In CCGrid, 2013.

[8] Ajay Gulati, Anne Holle, Minwen Ji, anesha Shanmuganathan, Carl Waldspurger and Xiaoyun Zhu. VMware Distributed Resource Management: Design, Implementation, and Lessons Learned. In VMware Academic Program.

[9] Rob Knauerhase, Paul Brett, Barbara Hohlt, Tong Li, and Scott Hahn. Using OS Observations to Improve Performance in Multicore Systems. In Micro, May 2008.

[10] Sergey Blagodurov, Sergey Zhuravlev, Mohammad Dashti and Alexandra Fedorova. A Case for NUMA-aware Contention Management on Multicore Systems. In ATC, 2011.

[11] Mohammad Dashti, Alexandra Fedorova, Justin Funston, Fabien Gaud, Renaud Lachaize, Baptiste Lepers, Vivien Quema and Mark Roth. Traffic Management: A Holistic Approach to Memory Placement on NUMA Systems. In ASPLOS, 2013.

[12] Sergey Zhuravlev, Juan Carlos Saez ,Sergey Blagodurov and Alexandra Fedorova. Survey of Scheduling Techniques for Addressing Shared Resources in Multicore Processors. In ACM Computing Surveys, September 2011.

[13] http://www.cs.virginia.edu/stream/.

[14] Intel(R) 64 and IA-32 Arhcitectures Software Develper's Manual, Volume 3B. System Programming Guide, Part 2.

[15] http://http://software.intel.com/en-us/articles/detecting-memory-bandwidth-saturation-in-threaded-applications.

[16] Jason Mars, Lingjia Tang, Rober Hundt, Kevin Skdron and Mary Lou Soffa. Bubble-Up: Increasing Utilization in Modern Warehouse Scale Computers via Sensible Co-locations. In MICRO, 2011.

[17] http://www.spec.org/cpu2006/.

[18] http://ark.intel.com/products/64596/IntelXeonProcessorE52690-20MCache2_90GHz8_00GTsIntelQPI

[19] http://ark.intel.com/products/52213/IntelCorei72600Processor-8MCacheupto3_80GHz

[20] Thomas B. Jablin, Prakash Prabhu, James A. Jablin, Nick P. Johnson, Stephen R. Beard and David I. August. Automatic CPU-GPU Communication Management and Optimization. In PLDI, 2011.

[21] Dongyou Seo, Shin-gyu Kim, Hyeonsang Eom and Heon Y. Yeom. Performance Evaluation of CPU-GPU communication Depending on the Characteristic of Co-Located Workloads. International Journal on Computer Science and Engineering, May 2013.

[22] https://www.cs.virginia.edu/ skadron/wiki/rodinia/index.php/.

[23] https://parsec.cs.princeton.edu/index.htm.

[24] Tanima Dey, Wei Wang, Jack W. Davidson and Mary Lou Soffa. Characterizing Multi-threaded Applications based on Shared-Resource Contention. In ISPASS, 2011.