M.S. THESIS

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

메모리 경쟁을 완화 시킬 수 있는 메모리 인지 로드 밸런스 기법

FEBRUARY 2014

DEPARTMENT OF COMPUTER SCIENCE ENGINEERING
COLLEGE OF ENGINEERING
SEOUL NATIONAL UNIVERSITY

Seo Dongyou

M.S. THESIS

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

메모리 경쟁을 완화 시킬 수 있는 메모리 인지 로드 밸런스 기법

FEBRUARY 2014

DEPARTMENT OF COMPUTER SCIENCE ENGINEERING
COLLEGE OF ENGINEERING
SEOUL NATIONAL UNIVERSITY

Seo Dongyou

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

메모리 경쟁을 완화 시킬 수 있는 메모리 인지 로드 밸런스 기법

지도교수 엄 현상

이 논문을 공학석사 학위논문으로 제출함

2013 년 11 월

서울대학교 대학원

전기.컴퓨터 공학부

서 동유

Seo Dongyou의 공학석사 학위논문을 인준함

2014 년 1 월

위 원 장	신 영길	(인)
부위원장	엄 현상	(인)
위 원	염 헌영	(인)

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; 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, task arrival patterns and the demand for resources in the current server environment are highly dynamic; hence, tasks, the number of which is larger than that of CPU cores, can be executed simultaneously on a CPU. In order to mitigate contention for memory subsystems in such multitasking cases, dealing with the dynamicity of resource demand, we have devised a Memory-aware Load Balancing (MLB) method. MLB dynamically recogrnizes contention by using simple contention models and performs inter-core task migration to mitigate the contention. We have evaluated MLB on an Intel i7-2600 (desktop-level CPU) and a Xeon E5-2690 (server-level CPU), and found that our approach can be effectively taken in an adaptive manner, leading to noticeable performance improvements of memory intensive tasks (about 15% in best case) on the different CPU platforms. Also, MLB can achieve performance improvements in CPU-GPU communication in discrete GPU systems.

Keywords: Multicore processors, shared resource contention, load balancing, dynamicity, SMP platform

Student Number: 2012-20784

Contents

List of Figures

List of Tables

0.1 Introduction

The current OS scheduler balances the runnable tasks across the available cores for balanced utilization among multiple CPU cores. For this, 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 and so on with other cores[Zhuravlev et al]. 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 overall system performance[Zhuravlev et al][Merkel et al]. In some articles, it is anticipated that off-chip memory bandwidth will often be the constraining resource in system performance[Patterson][Williams et al]. Even though it is expected that memory bandwidth is a major bottleneck in multi-core CPUs, the number of cores continues to increase. Intel plans to have an architecture that can scale up to 1,000 cores on a single chip[1000 scale chip]. 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[Natalie et al]. 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[Zhuravlev et al] [Kim et al][Hood et al][Merkel et al].

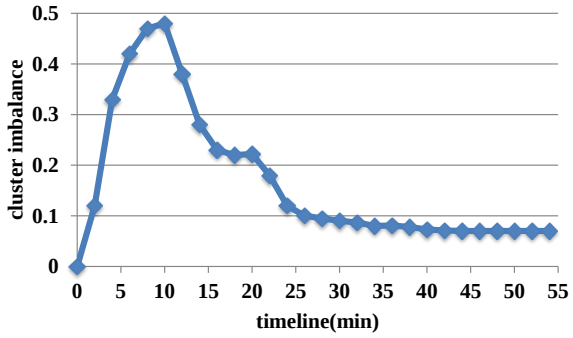


Figure 1 VMware cluster imbalance

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 large dynamic range of resource demands with high variations over short time intervals [Reiss et al] [Gupta et al]. 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[Gulati et al], 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[Gupta et al] [Gulati et al].

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[Zhuravlev et al] [Jiang et al][Chandra et al][Kim et al]. However, we should also consider the multitasking case because it is very difficult to predict the cluster imbalance which may prevent a task from being the sole owner of a core, leading to the case. In order to mitigate contention for shared resources, especially memory subsystems in the multitasking case, we have developed an inter-core load balancing method called Memory-aware Load Balancing (MLB). This method can be easily applied in harmony with the existing method for the singletasking case because it is designed to be triggered only in the multitasking case. We have evaluated MLB on existing SMP platforms and shown how effectively MLB can mitigate the contention for memory subsystems and hence improve system performance, demonstrating its strength in the case with dynamic resource demand.

The primary contributions of this paper are the following:

- MLB periodically monitors the level of memory contention, and gets triggered on some condition. We have devised simple models effectively by indicating the level of memory contention and predicting the 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.

- MLB 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. We have developed the MLB algorithm which fulfills dynamic inter-core task migration.

- We have evaluated MLB on existing SMP platforms, which are widely being used in practice. We have shown that MLB can be applied to both server-level (Xeon E5-2690) & desktop-level (i7-2600) CPUs 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 focus of our method lies on long-running, compute-bound and independent tasks. Also, we assume that the tasks hardly perform I/O operations and never communicate with one another as assumed in other related studies. Thus, we have used the SPEC CPU 2006 benchmark suite, the applications of which are single-threading programs and use the internal CPU resources aggressively. In addition, we have evaluated our method on the machines which have plenty of memory to minimize the effect of the additional I/O operations caused by the lack of memory.

The rest of the paper is organized as follows: Section 2 discusses the related work. Section 3 presents our motivations. Section 4 presents the result of quantifying the effect of memory contention, our memory contention model, and the correlation between the model and performance. Section 5 describes our load balancing algorithm and addresses several implementation issues. Section 6 shows the experimental results. Section 7 discusses advantages. Section 8 concludes this paper and presents our future work.

0.2 Related Work

Various solutions have been developed to mitigate the contention for shared resources via scheduling. [Jiang et al] presented the methology regarded as a perfect scheduling policy. Their method constructs a graph where tasks are

depicted as nodes connected by edges, the weights of which are the sums of the levels in performance degradation due to their resource contention between the two tasks. The methodology analyzes which tasks should share the same resource to minimize performance degradation caused by resource contention. However, it is feasible only for offline evaluation in contrast to ours. The overhead in graph construction is $O\left(n^2\right)$(n is the number of tasks). It is not a practical method if the number of tasks is considerably large. [Xie et al] introduced the animalistic classification, where each application can belong to one of the four different classes (turtle, sheep, rabbit and devil). Basically, it is hard to classify each application which has various usage patterns for sharing resources with only four classes. Moreover, some applications may belong to multiple classes such as both devil and rabbit classes. Also, the application is sensitive to the usage patterns of co-located tasks by polluting cache lines seriously. Thus, Xie's methodology may lack accuracy.

[Zhuravlev et al] proposed pain classification and Distributed Intensity (DI) which remedies the shortcomings of above mentioned methodologies in terms of practicality and accuracy. In their method, 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 [Chandra et al] and miss rate heuristic. Intensity is defined by the number of LLC references per one million instructions. Their method avoids co-locating the high sensitive task with the high intensive task. Also, they presented a detailed analysis to identify which shared resource in a CPU platform is a major factor causing performance degradation. The methology proposed by [Kim et al] 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, the 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] and [Knauerhase et al] 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 (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 on core A. Merkel's method is better than Knauerhase's method in various aspects. [Merkel et al] motivated our research, which is explained in detail in the next section.

Out of internal CPU resources, [Novakovic et al] introduces the first end-to-end system which transparently and efficiently handles interference on any major server resource in virtualized environments, including Network Interface Card(NIC) and disk I/O. But, I/O contention is out of our scope and their system requires sandbox nodes to analyze the contention for shared resources and fulfills inter-node migration. Our method does not need additional physical nodes to fulfill intra-node migration. [Ahn et al] presents method using live migration in virtualized environments to avoid remote accesses on Non-Uniform Memory Access(NUMA) architecture and mitigates the contention for memory subsystems. We do not consider NUMA architecture and virtualization but Ahn's method also does not consider multitasking case. Besides, there are harware apporaches to mitigate the contention for memory channel[Muralidhara et al]

and a methlogy using synthetic benchmark to mimic the inteference situation and predict the level of performance degradation[Mar et al] and many more.

0.3 Motivations

Two facts motivated our research. First, most of the recent studies on resource contention have assumed that only one task is being executed on a CPU core. The client using computing resources in the server environment expects his/her application is the sole owner of at least one CPU core. Thus, most of the existing contention-aware schedulers space-shares the machine rather than time-sharing it[Zhuravlev et al]. In fact, it is very difficult to predict the load imbalance among physical nodes with the dynamicity of resource demands. The dynamicity is one of the most important features of the current server environment[Reiss et al][Gupta et al]. However, the schedulers do not assume that multiple tasks might share a CPU core. The dynamicity also means dynamic task creation and termination. We have proposed a new method to mitigate the contention for memory subsystems in the multitasking case and shown its strength in dealing with dynamic task creation and termination.

Second, [Merkel et al] claim that migrating memory intensive tasks to specific cores pollutes cache lines and that it is not always possible to divide tasks into CPU or memory intensive task sets because there may be tasks which have medium memory intensity, concluding that it is not a good approach. It is not true that migrating memory intensive tasks to specific cores is also useless on the current SMP platforms which have bigger LLCs and provide the higher maximum memory bandwidth. Their CPU platforms were Intel Core2 Quad Q6600 and AMD Opteron 2354. They concentrated upon the evaluation with Q6600 whose memory controller exists out of die. The latency of the controller

is larger than that of integrated memory controller[Lin et al]. Two cores on a die share an L2 cache and there are two dies on the CPU platform while all cores on the current SMP platforms share an L3 cache. Also, they evaluated Opteron 2354, which is a SMP platform. Although Opteron 2354 is a server level CPU, the LLC capacity of Opteron 2354 is four times smaller than our desktop level CPU. As a result, their platforms are very different from the current SMP platforms which are widely being used, in that Q6600 has an old-fashioned architecture and the LLC capacity of Opteron 2354 is very small.

Table 1 shows the specifications of our SMP platforms. Intel i7-2600 is a desktop-level CPU, but it has an LLC of 8MB that is the same as the LLC capacity of sever-level Q6600 ($2 \times 4MB$), with higher CPU frequencies and memory bandwidth than Q6600. The performance of i7-2600 is thus a bit better than Q6600. Xeon E5-2690 is a high performance server-level CPU platform. We conducted static experiments with our CPU platforms. We evaluated two placements. In Placement 1, memory intensive tasks (lbm, libquantum, GemsFDTD and soplex) are migrated to specific cores. In Placement 2, memory intensive tasks are evenly distributed on all CPU cores. The placements are precisely shown in Table 2. We executed the task sets and measured performance metrics such as memory bandwidth (byte/sec), memory request buffer full rate (events/sec), LLC miss rate (events/sec) and LLC reference rate (events/sec), normalizing the results for Placement 1 to those for Placements 2. After any one task terminated, we stopped monitoring to analyze the situation where all memory intensive tasks were stressing memory subsystems. On Xeon E5-2690, soplex was the first terminating task in both Placement 1 (834 secs) and Placement 2 (877 secs). On i7-2600, lbm and namd were the first terminating tasks in Placement 1 (767 secs) and Placement 2 (773 secs), respectively.

As seen in Figure 2, LLC miss rate (E5-2690: 5.8%; i7-2600: 8.2%) and

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/s	21GB/s
CPU Category	Server level CPU	Desktop level CPU
Microacrcitecture	Sandy-bridge	Sandy-bridge

Table 1 The specifications of our SMP CPU platforms

LLC reference rate (E5-2690: 6.1%; i7-2600: 8.1%) increase a little bit in Placement 1 on all our platforms. On the other hand, memory bandwidth increases proportionally (E5-2690: 5.8%; i7-2600: 8.5%). Placement 1 truly leads to the pollution of cache lines, but the point which should be made is the decrease of memory request buffer full rate (E5-2690: 7.8%; i7-2600: 4.5%). Memory request buffer full event indicates the failure in inserting a memory request into memory request buffer due to the full condition of the buffer. If the request buffer in the memory controller is full, the task cannot enqueue its request right away. Thus, the task should retry to enqueue the memory request, thus wasting several cycles. The more frequently memory request buffer full events happen, the more CPU cycles are wasted in retrial. To measure memory request buffer full events, we use Intel's OFFCORE_REQUEST_BUFFER.SQ_FULL event [IntelGuide]. AMD also provides the performance counter related with the requst buffer events[AMDGuide]. [Zhuravlev et al] and [Merkel et al] showed that the contention for memory controller is more fatal in performance than LLC.

CPU platforms	Placement 1	Placement 2
Xeon E5-2690 (8cores CPU)	2lbm(M)[0,2], 2libquantum(M)[1,3], 2GemsFDTD(M)[1,3], 2soplex(M)[0,2], 2namd(C)[4,6], 2sjeng(C)[4,6], 2gamess(C)[5,7], 2gobmk(C) [5,7]	2lbm(M)[0,4], 2libquantum(M)[1,5], 2GemsFDTD(M)[2,6], 2soplex(M)[3,7], 2namd(C)[0,4], 2sjeng(C)[1,5], 2gamess(C)[2,6], 2gobmk(C) [3,7]
i7-2600 (4cores CPU)	lbm(M)[0], libquantum(M)[1], GemsFDTD(M)[1], soplex(M)[0], namd(C)[2], sjeng(C)[2], gamess(C)[3], gobmk(C)[3]	lbm(M)[0], libquantum(M)[1], GemsFDTD(M)[2], soplex(M)[3], namd(C)[0], sjeng(C)[1], gamess(C)[2], gobmk(C)[3]
Workload_name (M : memory intensive workload / C : CPU intensive workload) [core-id]		

Table 2 Workload placements

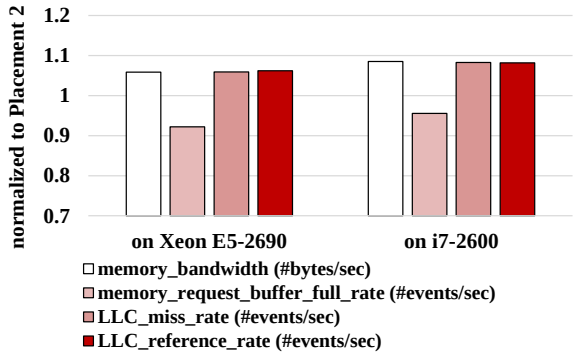


Figure 2 Average memory bandwidth, memory request buffer full rate, LLC miss rate and LLC reference rate measured when memory intensive tasks are statically migrated to specific cores

The increase of memory bandwidth and the decrease of memory request buffer full rate can sufficiently offset the increase of LLC miss rate and LLC reference rate in terms of performance.

According to our result, the statement made by [Merkel et al] is half right and half wrong for the current SMP platforms. On the current SMP platforms, the statement is true because LLC miss actually increases when memory intensive tasks are migrated to specific cores. On the other hand, the memory contention is mitigated by migrating memory intensive tasks to specific cores because memory request buffer full rate decreases. The mitigation of memory contention can offset the increase of LLC miss. We name the method driving memory intensive tasks into specific cores like placement 1 as Memory-aware Load Balancing(MLB). MLB is very simple and strong at dealing with the dynamicity of resource demand. However, simultaneously the tasks causing memory contention are created in a physical node. This is not a big problem to MLB because it simply drives the tasks into selected cores step by step. The

simplicity of MLB is advantageous over Merkel's method. We will explain the stronger points in more detail in section 9(advantages) after our experimental results are shown.

0.4 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 or if the corresponding actions are already taken. 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 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 to demonstrate the usefulness of the model.

0.4.1 Memory contention level

Figure 3 shows the average memory bandwidth and memory request buffer full rate with respect to the number of stream applications[Stream] on E5-2690. The memory bandwidth is the amount of retired memory traffic multiplied by 64 (size of a cacheline)[Memory bandwidth], and Intel provides Off-core response events which can permit measuring retired memory traffics[IntelGuide]. Retired memory traffic is the number of LLC miss events and prefetcher requests, and the traffic eventually flows into integrated memory controller. There is no single dominant component contributing to the contention of memory subsystems and several components play an important role[Zhuravlev et al]. Retired memory traffic is thus a good metric to monitor the overall utilization of mem-

ory subsystems including LLC, prefetcher and memory controller. 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.

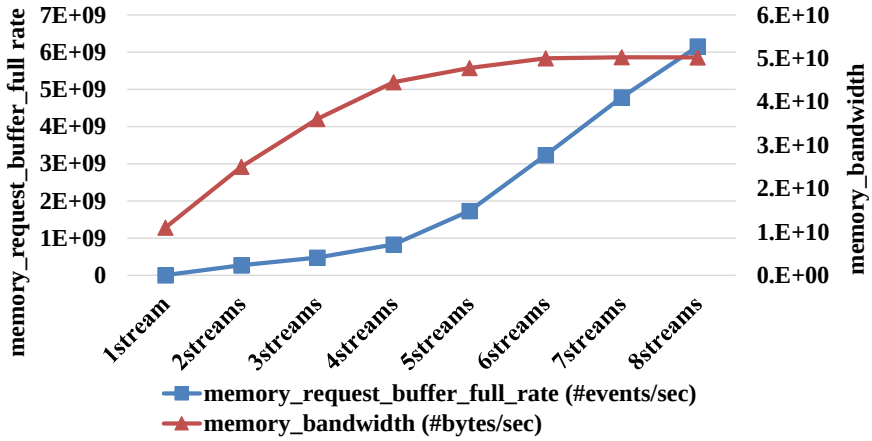


Figure 3 The correlation between memory request buffer full rate and memory bandwidth with respect to the number of stream applications

Memory bandwidth does not increase linearly even though the number of stream applications increases. The bandwidth is saturated at a constant level, which is near the maximum memory bandwidth of E5-2690 (about 50GB/sec). In contrast to the saturated memory bandwidth, memory request buffer full rate increases exponentially as the number of stream applications grows. As mentioned in Section 3, 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 memory request buffer is limited. The memory bandwidth and memory request buffer full rate shown in Figure 3 are symmetric with respect to the y=$a \times x$ line. The more gentle the inclination of memory bandwidth curve gets, the more exponential does the inclination of memory request buffer full rate become.

We constructed our memory contention model based on the correlation between memory bandwidth and memory request buffer full rate as seen in Figure 3. Equation (1) shows our model. Memory contention level is the number of retries to make a memory request retire. 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 connected to LLC, prefetcher and integrated memory controller.

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

0.4.2 The correlation between memory contention level and performance

To evaluate the correlation between memory contention level and performance, we did the stress tests for memory subsystems similar to [Mar et al]. 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 SPEC CPU applications. The memory bandwidth for each target application is different (see the value in the x-axis of the point labeled solo in Figure 4 and 5). We do the stress tests on both an i7-2600 (with four cores) and a Xeon E5-2690 (with eight cores). The results of E5-2690 are in Figure 4 and those of i7-2600 in Figure 5. To stress memory subsystems during the entire execution time of each target application, the stressors continued to run until the target application terminated. 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]. This sensitivity model is a very simple model, but the model 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 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). 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 Xeon E5-2690 increases as the number of stressors grows. The highest memory bandwidth reaches the maximum memory bandwidth (about 50GB/sec) when the number of stressors is 5

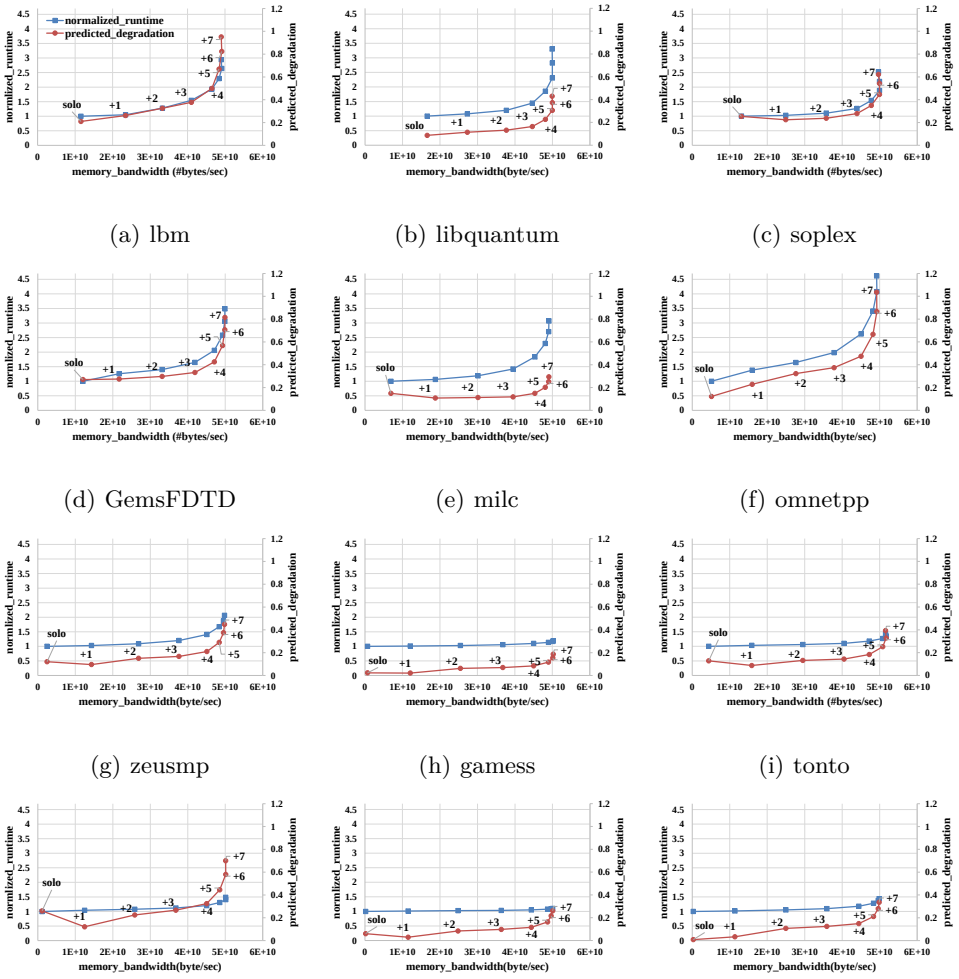
or 6. In contrast, the highest memory bandwidth of i7-2600 reaches the maximum memory bandwidth (about 21GB/s) when the number of stressors is 1 or 2. In the cases of executing lbm, soplex and GemsFDTD, the memory bandwidth decreases when each target application is executed with 3 stressors. The memory contention levels of the applications with 3 stressors are much higher than that of the non-memory intensive application which is tonto (lbm:11.6; soplex:10.88; GemsFDTD:9.55; 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 on i7-2600 can be drastically mitigated because the memory bandwidth of i7-2600 is low, thus more easily saturated than E5-2690. The results demonstrate that memory contention level effectively indicates the contention degree closely correlated with the performance of target application.

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

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

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

To implement MLB, many things should be determined. Which task should MLB consider as a memory intensive task? When should MLB be triggered? How many memory intensive tasks should MLB migrates? Which cores should MLB migrate memory intensive tasks to? In this section, we address these implementation issues in detail.



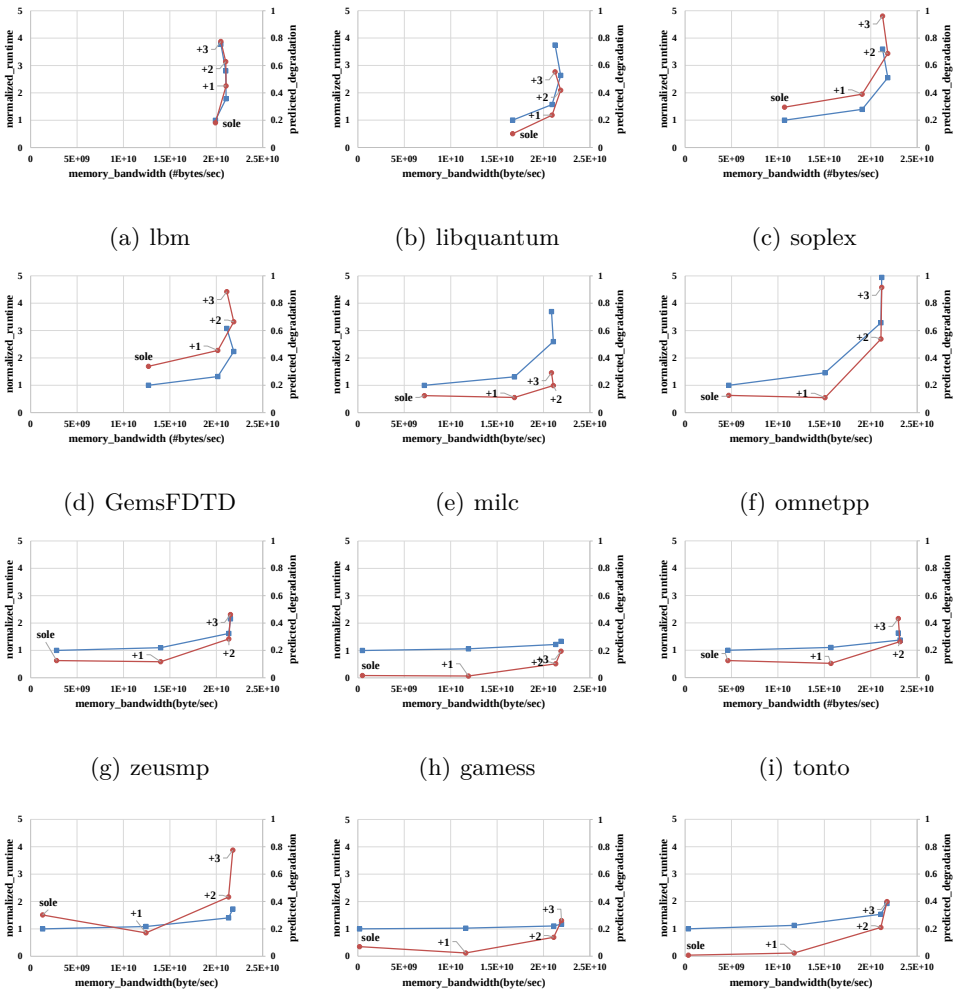
(a) lbm (b) libquantum (c) soplex

(d) GemsFDTD (e) milc (f) omnetpp

(g) zeusmp (h) gamess (i) tonto

(j) gobmk (k) namd (l) sjeng

Figure 4 The correlation between memory bandwidth, memory contention level and performanc on Xeon E5-2690



(a) lbm (b) libquantum (c) soplex

(d) GemsFDTD (e) milc (f) omnetpp

(g) zeusmp (h) gamess (i) tonto

(j) gobmk (k) namd (l) sjeng

Figure 5 The correlation between memory bandwidth, memory contention level and performanc on i7-2600

0.5.1 Target runqueue and non target runqueue lists

MLB should dynamically divide tasks into memory intensive task and non memory intensive task sets although we statically treat lbm, libquantum, soplex and GemsFDTD as memory intensive tasks in the previous sections. Therefor, MLB monitors the retired memory traffic (off-core response events [IntelGuide]) 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 historical behavior. To check the accuracy of the profiled intensity value with MLB, we compared the results of Intel Perfromance Counter Monitor (PCM)[PCM] with the traffic values monitored by MLB. We found that the difference 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 intensive.

Also, MLB should determine which core memory intensive tasks are migrated to. For this purpose, MLB manages two runqueue lists, target runqueue list and non-target one. Memory intensive tasks should be enqueued only to target runqueues. In contrast, non-memory intensive tasks should be enqueued only to non-target runqueues. But, MLB can migrate the non-memory intensive task which has the highest intensity to a target runqueue when the memory contention is intensive, but there are no memory intensive tasks in the non target runqueues. In this way, MLB performs the task migration possibly not based on the features of target runqueue and non-target runqueue. The numbers of target runqueues and non-target runqueues can dynamically increase and decrease depending on memory contention level.

0.5.2 Predicted number

The number of target runqueues indicates the condition where the migration of memory intensive tasks is performed. However, memory contention level is the variable, the value of which is basically the number of the retries to enqueue a memory request. When memory contention happens, MLB should find out which memory intensive tasks to migrate to target runqueues. We have constructed a model for the predicted number, which is the anticipated number of memory intensive tasks which are to be scheduled during the interval. 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. The predicted number indicates how many memory intensive tasks are expected to aggressively access memory subsystems during the interval. If the number of target runqueues is smaller than the predicted number, MLB decides that there is a memory intensive task out of target runqueues. To build a model for the predicted number, we used the fact that memory contention increases exponentially. Figure 6 presents the rising curve of memory contention level as the number of stream applications increases. We computed memory contention levels on 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 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.

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

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

We rotated the curve of $y = a \times x^2 + b$ on $y = x$ axis of symmetry, and then replace x, a and b with 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 lower maximum memory bandwidth is steeper than that of the higher CPU platform as shown in Figure 6. 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, 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 i7-2600 and 0.1 for E5-2690) and baseline (0.54 for i7-2600 and 0.2 for E5-2690) by using the results of the stress test as shown in both Figure 4, 5 and 6. This model outputs the approximated number of memory intensive tasks scheduled during the interval with an input of memory contention level.

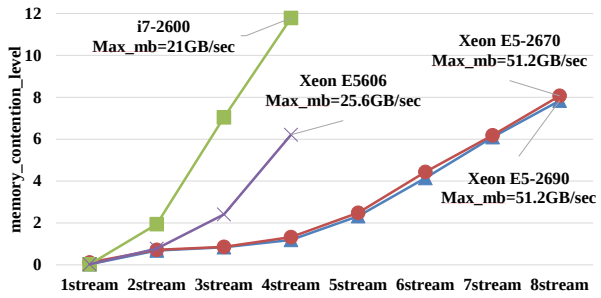


Figure 6 The memory contention levels of 4 CPU platforms with respect to the number of stream applications

MLB mitigates the contention for memory subsystems by changing the number of the tasks generating high memory traffic. In the mechanism of MLB, only the tasks in the target runqueues can generate the high memory traffic. When the predicted number is greater than the number of target runqueues, MLB decides that there are untreated memory intensive tasks which are in non-target runqueues. Then, MLB prepares to migrate the memory intensive tasks from the non-target runqueue to target runqueues.

0.5.3 Memory-aware Load Balacing algorithm

The MLB algorithm is presented as Algorithm 1. When two conditions are true, MLB is triggered. As described in the previous section, the predicted number is the anticipated number of running memory intensive tasks to be scheduled at the same time. When the predicted number is larger than the number of target runqueues (Line 3), MLB is triggered. However, MLB does not immediately perform task-migration. MLB checks the memory intensity (=value) of the highest task in the non target runqueues and compares the intensity with the average intensity of all running tasks because the predicted number can be overestimated (Line 9). When the intensity of the highest task is no higher than the average intensity, MLB decides that all memory intensive tasks are already migrated to target runqueues, and hence bypasses. This situation is considered as a false positive due to the misprediction. But, checking the intensity of the highest task can prevent MLB from performing unnecessary task migration. In contrast, MLB will be triggered when the intensity of the highest task in the non-target runqueues is higher. This means that at least the highest task should be migrated to a target runqueue.

MLB swaps memory intensive tasks in non-target runqueues with non memory intensive tasks in target runqueues. When there is no non-memory intensive

task in target runqueues, MLB inserts the highest runqueue into target runqueue list, which has the highest intensity sum for all tasks in a non target runqueue to minimize the number of task swapping (Lines 13 and 41) and resumes swapping. To prevent the imbalance of CPU load among all cores, MLB does not perform the task swapping if the difference between the CPU load of the tasks to swap is greater than a certain threshold. The threshold is empirically determined and set to 15%. MLB determines the non-target runqueue including the highest task as the source runqueue (Line 8). The memory intensive tasks in the source runqueue are swapped with non-memory intensive tasks in target runqeueus (Line 29). Until only non-memory intensive tasks reside in the source runqueue, MLB continues to conduct task swapping and insert a non-target runqueue to the target runqueue list if needed (Lines 41 and 42). MLB also removes the lowest runqueue which has the lowest intensity sum of all tasks in a target runqueue if the predicted number is no greater than the number of target runqueues (Line 5).

MLB can be performed with exisiting CPU load balancer. To maintain the features of target runqueue and non-target runqueue, we made a slight modification of the CPU load balancer since the CPU load balancer does not consider the memory intensity of a task. When our CPU load balancer selects the busiest runqueue as a target runqueue and the idlest runqueue as a non-target runqueue, it traverses the tasks residing in the target runqueue in ascending order of memory intensity. It preferably migrates the task with the lowest intensity in the target runqueue. In the opposite situation, it preferably migrates the task with the highest intensity in the non-target runqueue. Also, MLB can evict the tasks which are transformed from memory intensive ones to non-memory intensive ones. To deal with such cases, MLB periodically checks the number of transformed tasks in all target runqueues. If the number of transformed tasks

is no smaller than that of memory intensive tasks in one target runqueue, MLB swaps the memory intensive tasks with the transformed tasks and removes the target runqeue filled with the transformed tasks from the target runqueue list.

The number of target 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 target runqueues. As a result, the steps of MLB enforce only the selected cores (target runqueues) to generate memory requests via task migration.

Algorithm 1 Memory-aware Load Balancing Algorithm

```
1: Memory_aware_Load_Balancing() begin
2: predicted_number = get_predicted_number(memory_contention_level)
3: if predicted_number < target_rq_list → count then
4:     lowest_rq=find_the_lowest_rq(target_rq_list)
5:     add_rq_to_target_rq_list(lowest_rq,non_target_rq_list)
6:     return
7: end if
8: src_rq = find_the_rq_having_highest_task(non_target_rq_list)
9: if src_rq → highest_task → mem_value ≤ avg_value then
10:     return
11: end if
12: if target_rq_list.count = 0 then
13:     highest_rq=find_the_highest_rq(non_target_rq_list)
14:     add_rq_to_target_rq_list(highest_rq,target_rq_list)
15: end if
16: count = src_rq → nr_running
17: dest_rq = target_rq_list → head
18: while count > 0 do
19:     if src_rq = dest_rq then
20:         return
21:     end if
22:     for each task in src_rq do
23:         if count = 0 then
24:             break
25:         end if
26:         if avg_value < dest_rq → lowest_task → mem_value then
27:             break
28:         end if
29:         swap_two_tasks(src_rq, dest_rq → highest_task,
                                dest_rq,dest_rq → lowest_task )
30:         count = count - 1
31:         update_rq(dest_rq)
32:         update_rq(src_rq)
33:         if avg_value > src_rq → highest_task → mem_value then
34:             return
35:         end if
36:     end for
37:     if count > 0 then
38:         dest_rq = dest_rq → next
39:     end if
40:     if dest_rq = NULL then
41:         highest_rq=find_the_highest_rq(non_target_rq_list)
42:         add_rq_to_target_rq_list(highest_rq,target_rq_list)
43:         dest_rq = highest_rq
44:     end if
45: end while
```

0.6 Evaluation

In this section, we present the result of evaluating MLB in various ways. Although the dynamicity of resource demand is very high, we assume that the maximum runqueue length (the number of tasks in a runqueue) is two, considering the fact that cluster load balancers strive to let a CPU bound task become the sole owner of a CPU core. But, MLB can mitigate the memory contention when the maximum runqueue length is larger than two. In all our experiments, memory contention level is calculated every four seconds because four seconds empirically turned out to be the best length of interval. We evaluated the ability of MLB to cope with dynamic task creation and termination situations. Next, we present how much memory contention is mitigated and the performance is thereby improved. Lastly, we discuss the overhead of MLB and an addtional benefit.

0.6.1 Dynamicity

Dynamicity is the main characteristic of current server environments, and various tasks are dynamically created or terminated in such environments. MLB is the method optimized for the dynamicity. Our memory contention and predicted number models are simple and effective. Based on these models, MLB dynamically adjust the number of target runqueues to control the amount of simultaneous memory requests, and therefore MLB can 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 models 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 simulate the dynamicity in the current server environment.

Figure 7 shows the results for Xeon 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 7 (a) and (c) with MLB and the naïve Linux scheduler (naïve), respectively, 2lbms started in 30 secs, 2libquantums in 60 secs, 2soplexs in 90 secs and 2GemsFDTDs in 120 secs. After 2lbms started, MLB detected memory contention so that 2lbms were migrated to core 5 while the migration of 2lbms was slightly delayed due to the lack of profiled data. The predicted number was simultaneously calculated by logging the footprints of memory intensive tasks from the beginning as shown in Figure 7 (b). We used time window(19 old predicted number and 1 current predicted number) because the memory intensity of task was not always invariable unlike that of stream application. The window has a role in smoothing predicted number and preveting it from fluctuating.

Although only CPU intensive tasks were running before 2lbms started, the memory contention (the predicted number two) occurred due to the CPU intensive tasks and hence there were already two target runqueues at the time when 2lbms started. Compared with the single-tasking case, each lbm timeshared the co-running CPU intensive task in this case and hence 2lbms were not always scheduled at the same time. Thus, the predicted number was not increased twice.

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 by 3. The number of target runqueues increased as the predicted number in accordance with memory contention level increased. After all memory intensive tasks started, MLB managed four target runqueues (core 3,

4, 5 and 6) by changing the number of cores generating high memory traffic to up to four.

In contrast to MLB, memory intensive tasks were irregularly placed with the naive Linux scheduler as shown in Figure 7 (c). Also, memory intensive tasks experienced many migrations whenever some tasks were created. The frequent CPU imbalance happened because the CPU load was changed when a task was created. Then, the naive scheduler chose memory intensive tasks as victims to balance the CPU load with. Thus, the footprints of memory intensive tasks are very irregular during task creations (from 30 to 120 secs) and terminations (from 300 to 390 secs). With MLB, memory intensive tasks were the last victims for the CPU balance by our optimizations as explained in Section 5.3. Non-memory intensive tasks are first selected and then memory intensive tasks are checked in the end when the CPU load of all non-memory intensive tasks is similar. The optimization is good for memory intensive tasks, but it can be bad for non-memory intensive tasks because non-memory intensive tasks experience more migration than memory intensive tasks. We explain the impact of frequent migrations of non-memory intensive tasks in Section 6.5 titled Overhead.

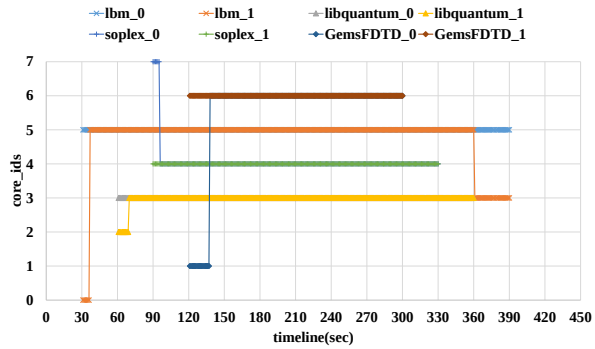
If memory intensive tasks dynamically terminate, then MLB decreases the number of target runqueues. 2GemsFDTDs end in 300 secs, 2soplexs in 330 secs, 2libquantums in 360 secs and 2lbms in 390 secs while all CPU intensive tasks terminated in 420 secs. The number of target runqueues decreased as the predicted 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 is 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 which was the sole owner of a core, being applied in harmony with the existing methods considering the single-tasking case. But the ping-pong migration of memory intensive tasks did not occur as seen from 60 to 90 secs. For clarity, we present only the results for memory intensive tasks, but there was no ping-pong migration of CPU intensive tasks in the case for these tasks, either.

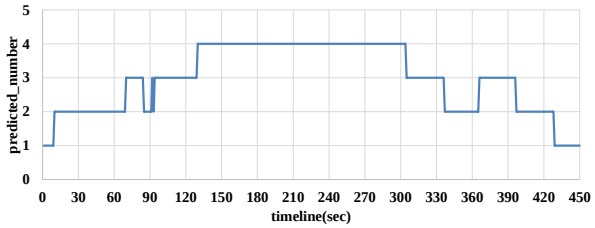
MLB effectively handles the underestimated memory contention. The worst case is the case where MLB does not migrate memory intensive tasks to target runqueues although the memory contention continues to occur. With the naive scheduler, memory intensive tasks can be spreading on multiple cores, each on a core, leading to intensive memory contention in our task creation and termination scenario. In contrast, MLB immediately responds to the creation and termination of the tasks which cause the memory contention, based on our memory contention and predicted number models as shown in Figure 7 (a). The ability to cope with the dynamic task creation and termination is the biggest advantage over the other methods considering the multitasking case like MLB.

0.6.2 Mitigation

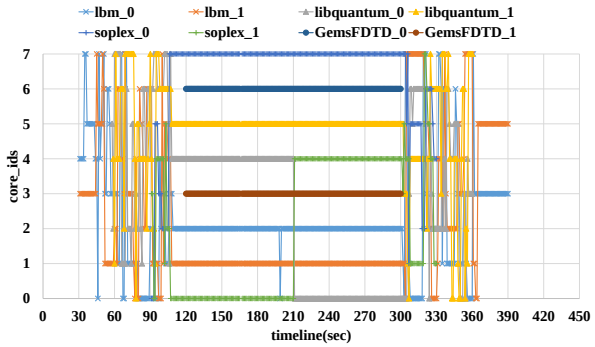
To figure out how much the contention can be mitigated by MLB, we evaluated two workload mixes per CPU platform. The mixes are described in Table 3. The number of memory intensive tasks in workload mix1 is twice larger than that for workload mix2. In other words, memory contention level in workload mix1 is higher. We executed the workload mixes with both MLB and the naive Linux scheduler and measured four performance metrics (memory bandwidth, memory request buffer full rate, LLC reference rate and LLC miss rate). We normalized the values of the metrics with MLB to those of the naive Linux scheduler. To identify the mitigation degree for memory contention during the



(a) The footprints of memory intensive tasks with MLB



(b) The timeline of predicted number



(c) The footprints of memory intensive tasks with naive

Figure 7 The scenario of dynamic task creation and termination

execution of all tasks, we ran all applications at the same time and stopped measuring after any one task terminated in the experiment similar to that for Figure 2. The result is shown in Figure 8. MLB decreased memory request buffer full rate and increased memory bandwidth in all the cases. However, the result for i7-2600 is more drastic. MLB migrated memory intensive tasks to two target runqueues in the case of workload mix1 on i7-2600. But, the amount of mitigated memory contention was slightly reduced with MLB because four memory intensive tasks (lbm, libquantum, GesFDTD and soplex) in two target runqueues could sufficiently saturate the memory subsystems. In contrast to workload mix1, the amount of mitigated memory contention was considerably reduced in workload mix2. Two memory intensive tasks in one target runqueue were never scheduled at the same time. In this situation, lbm and libquantum were alternately scheduled on a core, and thus memory bandwidth increased and memory request buffer full rate decreased.

Compared with the results for i7-2600, there were no dramstic results for E5-2690 because the test sets in workload mix1 and mix2 could not saturate the maximum memory bandwidth. The memory requests buffer full rate decreased and the memory bandwidth increased similarly in both mix1 and mix2. The reduced ratio of memory request buffer full rate and improved ratio of memory bandwidth reflected the performance improvements with MLB. In all four cases shown in Figure 8, the increased ratio of LLC miss and reference rate were under 5% with MLB.

To see the time-based mitigation with MLB, we measured the temporal change in memory contention level while we started executing sixteen applications in workload mix1 and mix2 on Xeon E5-2690. We ran all applications at the same time. The timeline is shown in Figure 9 (a) and (b). MLB could decrease the memory contention level most of the time compared with the naive

# of workload mixes	Workloads
Workload-mix1 (on E5-2690)	2lbm(M),2libquantum(M),2GemsFDTD(M) ,2soplex(M),2namd(C),2sjeng(C) ,2gamess(C),2gobmk(C)
Workload-mix2 (on E5-2690)	2lbm(M),2libquantum(M),2tonto(C) ,2zeusmp(C),2namd(C),2sjeng(C) ,2gamess(C),2gobmk(C)
Workload-mix1 (on i7-2600)	lbm(M),libquantum(M),GemsFDTD(M) ,soplex(M),namd(C),sjeng(C),gamess(C) ,gobmk(C)
Workload-mix2 (on i7-2600)	lbm(M),libquantum(M),tonto(C),zeusmp(C) ,namd(C),sjeng(C),gamess(C),gobmk(C)

Table 3 Workload mixes

Linux scheduler.

0.6.3 Performance

The performance results are presented in Figure 10. We executed the applications in two mixes per CPU platform more than 10 times and normalized the average execution times with MLB to the naïve scheduler. To figure out the overall effect of MLB to every application, we executed all applications at same time and continued to re-execute the applications until all of them finished at once and the first execution times were sampled. All memory intensive applications (lbm, libquantum, GemsFDTD and soplex) benefited from MLB. In particular, the execution times of memory intensive applications were significantly reduced in workload mix2 on i7-2600. Lbm and libquantum benefited

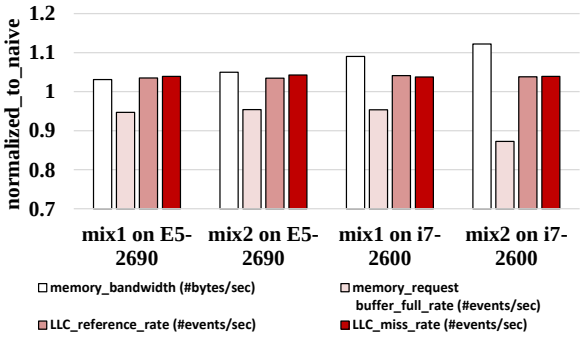
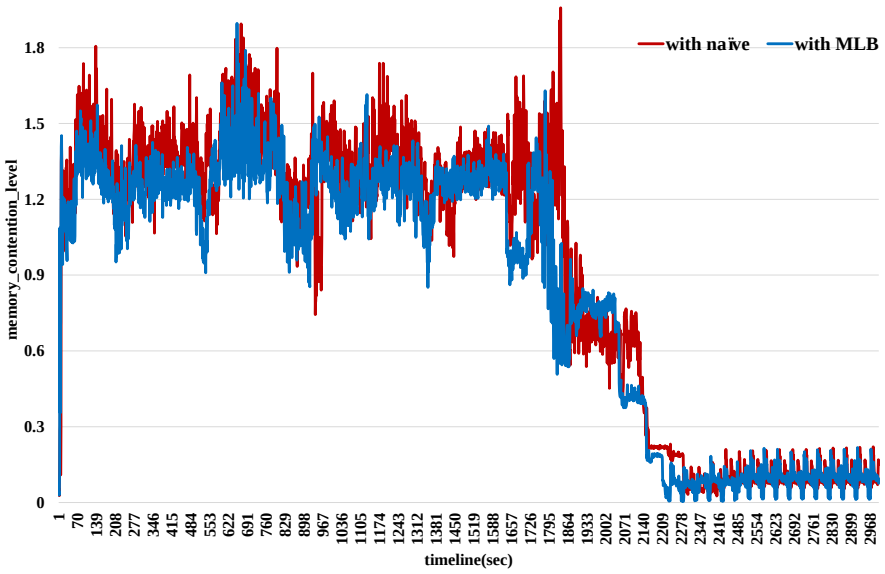


Figure 8 Average metric comparisons between MLB and naive

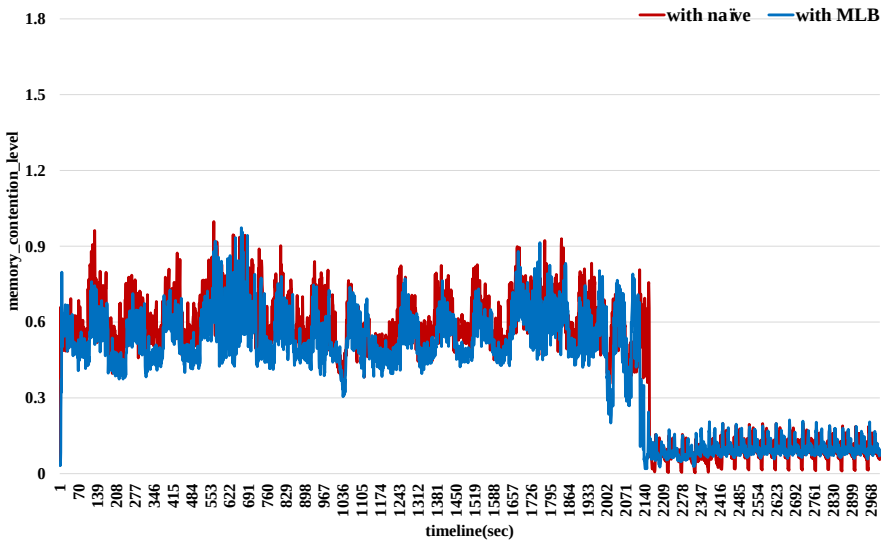
from the highest decrease of memory request buffer full rate as shown in Figure 8. What is interesting is that the performance benefit of memory intensive tasks differs for i7-2600 as shown in Figure 10 (c). The reason is that the distribution of memory bandwidth can be asymmetric depending on the memory access patterns of applications[Moscibroda et al]. However, MLB could achieve the noticeable performance improvements for memory intensive applications in both the server (E5-2690) and desktop level CPU (i7-2600).

0.6.4 Additional benefit

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[CUDA]. In the programming model, CPU-GPU communication is very important for the performance of GPU application[Jablin et al]. However, 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[Seo et al]. In high memory contention situation, a GPU application may not obtain sufficient memory bandwidth. MLB can indirectly

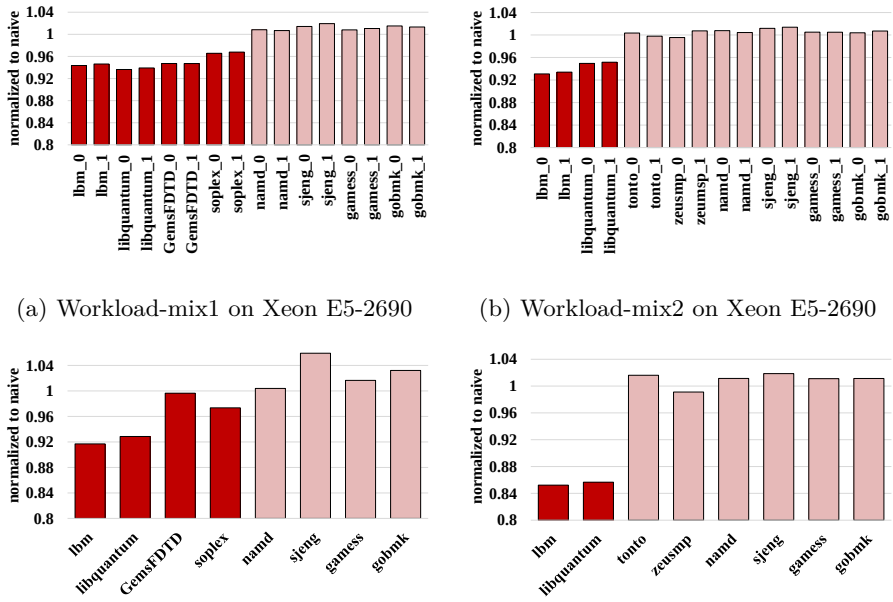


(a) Workload-mix1 on Xeon E5-2690



(b) Workload-mix2 on Xeon E5-2690

Figure 9 The timeline of memory contention level



(a) Workload-mix1 on Xeon E5-2690

(b) Workload-mix2 on Xeon E5-2690

(c) Workload-mix1 on i7-2600

(d) Workload-mix2 on i7-2600

Figure 10 Average runtime comparisons between MLB and naive

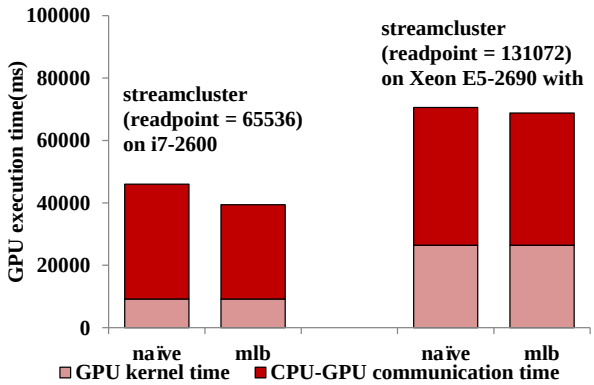


Figure 11 The improvement of CPU-GPU communication with MLB

allow the GPU application to occupy more memory banwidth. We have evaluated MLB with NVIDIA GTX580 and Tesla K20C on i7-2600 and on E5-2690, respectively. We executed a GPU application, stremcluster of rodinia benchmark suite[Rodinia] and used nvprof to measure GPU kernel time and CPU-GPU communication time[Nvprof]. We replaced a soplex with a streamcluster in workload mix1 on both i7-2600 and E5-2690 and compared MLB with naive as shown in Figure 11. MLB decreased the CPU-GPU communication time in both CPU platforms more than naive while having the kernel times in all cases not changed. The result demonstrates that MLB can lead to improvements in the performance of CPU-GPU communication via the mitigation of memory contention when memory contenion is intensive.

0.6.5 Overhead

As can be shown in Figure 10, MLB did not improve the performance of all applications. MLB carried out additional task migrations and preferred to migrate non-memory intensive tasks to balance the CPU load when compared with the naive Linux scheduler. Also, MLB increased LLC miss and reference rate (under 5%). The additional and preferred migration and the increase of LLC miss degraded the performance of non-memory intensive tasks. To analyze in detail, we measured the kernel time increased by MLB when the task set in workload mix1 was executed on Xeon E5-2690 because MLB performed the additional migration in kernel level. The kernel time increased by 1.3% compared with the naive Linux scheduler. The increase ratio of the kernel time is very similar to the performance degradation ratio of non-memory intensive applications as shown in Figure 10 (a). However, the performance degradation for non-memory intensive tasks is relatively negligible, being under 2%, when compared with the performance gain for memory intensive applications.

0.7 Advantages

Merkel's vector balancing & sorted co-scheduling method is the state of the art method to mitigate the memory contention in the multitasking case. We compared our method with Merkel's because the goal of MLB is similar to that of Merkel's. MLB has three advantages over Merkel's. First, they evaluated the vector balancing & co-scheduling method on the outdated CPU platform. Most of the current CPU platforms have the SMP architecture and bigger capacity of LLC, but one of their platforms does not have unified LLC and the memory controller is not integrated on the CPU platform (Q6600) and the other has very small capacity of LLC (Opteron 2354). Their CPU platforms are different in the current CPU platforms. MLB was evaluated on the two current CPU platforms which have more capacity of LLC and higher maximum memory bandwidth having the SMP architecture. In particular, Xeon E5-2690 is a very high performance CPU platform. We showed that MLB could even mitigate the contention for memory subsystems on the high performance CPU platform as shown in Figure 8.

Second, Merkel's method requires manipulating the time slice mechanism of the default CPU scheduler in order to ensure ordered scheduling of tasks. In contrast, MLB does not need to be modified in the time slice mechanism. Their method distributes tasks to maximize the vector deviation of each runqueue (we can regard vector as the intensity of memory subsystems). Both memory intensive task and CPU intensive task co-exist in a runqueue. The task distribution is 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 epoch. That means that the number of memory intensive tasks running in the same epoch is adjusted

Methods	core 0	core 1	core 2	core 3
Vector balancing	(0.9,0.1)	(0.15,0.86)	(0.2,0.8)	(0.79,0.21)
MLB	(0.9,0.86)	(0.8,0.79)	(0.2,0.1)	(0.21,0.15)

Table 4 The result of comparison between MLB and vector balancing

and their method fixes the amount of simultaneous memory requests. It is similar to MLB that their method limits the number of memory intensive tasks running at the same time. But, their method needs to be modified in the time slice mechanism of the default CPU scheduler. There are quite a lot of optimization done for the fairness in the default Linux scheduler, Completely Fair Scheduler(CFS)[CFS]. MLB can be implemented just by modifying the part of periodical load balancing[Love]. Unlike sorted co-scheduling method, MLB can be performed based on the fairness provided by CFS.

Lastly, MLB is more flexible in responding to dynamic task creation and termination rather than vector balancing & sorted co-scheduling. To demonstrate the advantage of MLB, we present the result of comparison as shown in Table 4. Suppose that there are eight tasks executed on a four-core CPU. The vectors of tasks are 0.9, 0.86, 0.8, 0.79, 0.21, 0.2, 0.15 and 0.1. The vector implies the memory intensity. We can regard 4 tasks (0.9, 0.86, 0.8 and 0,79) as memory intensive tasks. With vector balancing, the tasks are distributed as shown in the first row of Table 5. To change the number of simultaneous memory requests, the 0.9 task and 0.79 task are executed on core 0 and core 3 in the odd epoch. In the even epoch, the 0.86 task and 0.8 task are respectively executed on core 1 and core 2 to prevent all memory intensive tasks from being scheduled in the same epoch. Their method effectively changes the number of simultaneous

memory requests. But, what if the 0.21 task terminats? The task re-distribution is not needed because the sum of task vectors is already the maximum. In the odd epoch, it is no harm done because the number of memory intensive tasks running at the same epoch is two and the vector sum is 2.04 (0.9 task on core 0, 0.15 on core 1, 0.2 on core 2 and 0.79 task on core 3). However, the number is three and the sum is 2.55 in the even epoch (0.1 task on core 0, 0.86 task on core 1, 0.8 task on core 2 and 0.79 on core 3). With MLB, the number of memory intensive tasks scheduled at the same time is two and the worst vector sum is 2.11 whichever task terminates.

What if the 0.05 task is created? It is not the maximum vector deviation of each runqueue whichever the task is placed on. The 0.9 and 0.05 tasks should be located on core 0, the 0.86 and 0.1 tasks on core 1, the 0.8 and 0.15 tasks on core 2 and the 0.79, 0.21 and 0.2 tasks on core 3. Then, five task migrations are needed to maximize the vector deviation. Merkel's method can be effective when the vector deviation of each runqueue is the maximum. In contrast, there is no additional procedure whichever the task is placed on in the case of MLB. MLB just evicts a non-memory intensive task from the target runqueues and takes in a memory intensive task when the "treated" state is under the contention degree.

In the server environment, task creation and termination can frequently happen due to the dynamicity of resource demand. As seen in above examples, Merkel's method cannot be flexible in responding to dynamic task termination and creation. The MLB mechanism is simpler than that of Merkel's so that MLB can deal with the dynamicity better than Merkel's. MLB can immediately react to mitigate memory contention as shown in Figure 7 (a).

0.8 Conclusions and Future Work

MLB is a new method to mitigate the contention for memory subsystems and leads to noticeable performance improvements for memory intensive tasks. MLB focuses on multitasking and dynamicity. Most of the existing methods for mitigating the contention concentrate on the single-tasking case. In contrast, MLB targets the cases where the number of running tasks is larger than the number of CPU cores, handling the cluster imbalance. In addition, MLB is stronger in coping with dynamic task creation and termination than other existing methods which consider the multitasking case like MLB. Based on the memory contention and predicted number models, MLB can effectively deal with the dynamicity.

However, there are currently some issues regarding MLB. First, MLB should 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 (c), 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, we plan to optimize MLB in discrete GPU systems. Discrete GPUs share the bandwidth of main memory with CPUs. Thus, the GPU can be affected by the memory contention. We will extend MLB for the bigger systems which have NUMA architectures and are equipped with multiple CPUs/GPUs.

Bibliography

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

[Zhuravlev et al] 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.

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

[Williams et al] Samuel Williams, Andrew Waterman and David Patterson. Roofline: An insightful Visual Performance model for multicore Architectures. In Communication of the ACM, 2009.

[1000 scale chip] http://goo.gl/5lpY8, 2010.

[Natalie et al] Natalie Enright Jerger , Dana Vantrease and Mikko Lipasti. An evaluation of server consolidation workloads for multi-core designs. In IISWC, 2007.

[Jiang et al] Y. Jiang, X. Shen, J. Chen, and R. Tripathi. Analysis and Approximation of Optimal Co-Scheduling on Chip Multiprocessors. In PACT, 2008.

[Xie et al] Y. Xie and G. Loh. Dynamic Classification of Program Memory Behaviors in CMPs. In Proc. of CMP-MSI, held in conjunction with ISCA, 2008.

[Chandra et al] D. Chandra, F. Guo, S. Kim, and Y. Solihin. Predicting InterThread Cache Contention on a Chip Multi-Processor Architecture. In HPCA, 2005.

[Reiss et al] 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.

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

[Gulati et al] 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.

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

[Hood et al] Robert Hood, Haoqiang Jin, Piyush Mehrotra, Johnny Chang, Jahed Djomehri, Sharad Gavali, Dennis Jespersen, Kenichi Taylor, and

Rupak Biswas. Performance Impact of Resource Contention in Multicore Systems. In IPDPS, 2010.

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

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

[Novakovic et al] Dejan Novakovic, Nedeljko Vasic, Stanko Novakovic, Dejan Kostic , and Ricardo Bianchini. DeepDive: Transparently Identifying and Managing Performance Interference in Virtualized Environments. In ATC, 2013.

[Muralidhara et al] Sai Prashanth Muralidhara, Lavanya Subramanian, Onur Mutlu, Mahmut Kandemir and Thomas Moscibroda. Reducing Memory Interference in Multicore Systems via Application-Aware Memory Channel Partitioning. In MICRO, 2011.

[Mar et al] 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.

[Lin et al] Wei-Fen Lin,Reinhardt S.K and Burger D. Reducing DRAM latencies with an integrated memory hierarchy design. In HPCA, 2001.

[Moscibroda et al] T. Moscibroda and O. Mutlu. Memory performance attacks: denial of memory service in multi-core systems. In SS, 2007.

[Ahn et al] Jeongseob Ahn, Changdae Kim and Jaeung Han. Dynamic Virtual Machine Scheduling in Clouds for Architectural Shared Resources. In HotCloud, 2012.

[Seo et al] 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.

[Jablin et al] 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.

[SPEC CPU 2006] http://www.spec.org/cpu2006/.

[CFS] https://www.kernel.org/doc/Documentation/scheduler/sched-design-CFS.txt.

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

[AMDGuide] BIOS and Kernel Developer' s Guide (BKDG) for AMD Family 15h Models 00h-0Fh Processors.

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

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

[CUDA] CUDA Programming Model Overview, NVIDIA techncal report.

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

[Nvprof] http://docs.nvidia.com/cuda/profiler-users-guide/index.html.

[Love] Robert Love, Linux Kernel Development, Third Edition.

[PCM] http://software.intel.com/en-us/articles/intel-performance-counter-monitor-a-better-way-to-measure-cpu-utilization.

요약

현재 대부분의 CPU는 단일 코어가 아닌 Symmetric MultiProcessing(SMP) 구조로 발전하고 있다. 이 SMP구조는 다수의 물리적인 코어에 의해 말단 캐시와 내장형 메모리 컨트롤러가 공유되는 구조를 가진다. 허나 SMP구조에서는 이런한 공유자원에 대한 물리적인 코어간의 경쟁은 상당한 성능 저하를 이끌 수 있다. 다양한 방법들이 공유자원에 대한 경쟁을 완화시키기 위해 고안 되었으며 대부분의 방법들은 하나의 태스크가 하나의 물리적인 코어를 독점한다는 것을 가정하고 태스크의 특성에 따라서 어떤 태스크 끼리 자원을 공유할지 결정하는 방법이다. 하지만 현재의 서버 환경에서는 태스크의 생성 주기와 자원에 대한 요구가 굉장히 역동적(dynamic)이기 때문에 하나의 태스크가 물리적인 코어를 독점할 수 없는 상황이 발생하여 다수의 태스크가 하나의 코어에서 스케줄을 기다리는 멀티 태스킹 상황이 발생한다. 본 논문에서는 기존의 자원 경쟁 완화 기법들과 비교 했을 때 멀티 태스킹 상황을 고려하고 역동적인 자원 요구에 강점을 보이는 Memory-aware Load Balancing(MLB) 기법을 제시하였다. MLB는 단순하면서 효과적인 메모리 경쟁모델을 이용하여 동적으로 공유자원의 경쟁을 감지하고 CPU내부적인 태스크 이주를 통해서 자원 경쟁을 완화시키는 기법이다. MLB를 서버급(Xeon E5-2690) CPU와 데스크탑(i7-2600) CPU에서 평가를 하였으며 MLB가 이 두 CPU 모두에서 자원 경쟁을 완화시켜 메모리 집약적인 프로그램의 성능 향상(최적의 경우 15%)을 이끌 수 있다는 것을 확인하였다. 추가적으로, 메모리 대역폭을 이용하여 CPU와 GPU간의 통신이 이루어 지는데 MLB는 메모리 경쟁을 완화시키기 때문에 이 CPU와 GPU간의 통신의 성능 향상까지 이끌 수 있었다.

주요어: 서울대학교, 전기.컴퓨터공학부, 졸업논문
학번: 2012-20784

Acknowledgements

이 논문을 작성하는데 도움을 주신 모든 분들께 감사의 말씀 드리겠습니다. 언제나 날카로운 지적으로 저의 연구방향을 지도해주신 염헌영 교수님께 감사드립니다. 감히 말씀드리지만 교수님은 제가 만나뵌 엔지니어 중 최고의 엔지니어 이십니다. 연구를 하는데 있어서 아낌없는 지원을 해주신 저의 지도 교수님 엄현상 교수님께 감사의 말씀 드립니다. 2년동안 교수님께서 저에게 주신 신뢰 잊지 못할 것입니다. 제가 이 논문을 작성하는데 가장 영향력을 주신 신규형님께 감사드립니다. 무심한듯 하시면서도 저를 챙겨주시고 연구 하는데 아이디어를 주신 신규형, 형이 있어 제가 이 논문을 완성할 수 있었습니다. 감사합니다. 같은 방에 있으면서 때론 장난도 치면서 같이 열심히 연구실 생활을 한 명원이형, 지웅이, 윤희 당신들이 있어서 연구실에서 웃을 수 있었습니다. 감사합니다. 동기로 들어와서 많은 것들을 함께한 민영이 너가 있어서 혼자 연구실 생활 한다는 느낌을 받지 않은거 같아. 고마워. 그리고 인순누나, 영진이형, 재우형, 찬호, 세훈이형, 신웅형, 성구형, 설웅형, 성재형, 용석이형, 시봉이형, Pablo, 내영이, 계신이, 문봉이, 한울이, 다혜까지 도움을 주신 분들께 감사 말씀 드리겠습니다. 마지막으로, 저를 지금까지 키워준 부모님, 누나 감사드리고 이 논문이 완성된 것을 알게 된다면 기뻐할 사람에게 이 논문을 바칩니다. 감사합니다.