Bubble Task: A Dynamic Execution Throttling Method for Multi-core Resource Management

Dongyou Seo, Hyeonsang Eom, and Heon Y. Yeom

School of Computer Science and Engineering,
Seoul National University, Seoul, Korea
{dyseo,hseom,yeom}@dcslab.snu.ac.kr

Abstract. Memory bandwidth is a major resource which is shared among all CPU cores. The development speed of memory bandwidth cannot catch up with the increasing number of CPU cores. Thus, the contention for occupying more memory bandwidth among concurrently executing tasks occurs. In this paper, we have presented Bubble Task method which mitigates memory contention via throttling technique. We made a memory contention modeling for dynamically deciding throttling ratio and implemented both software and hardware versions to present trade-off between fine-grained adjustment and stable fairness. Bubble Task can lead to performance improvement in STREAM benchmark suite which is one of the most memory hungry benchmark by 21% and fairness in memory bandwidth sharing among SPEC CPU 2006 applications which have different memory access patterns.

Keywords: Multicore processor, SMP platform, CPU execution throttling, resource contention, bandwidth fairness

1 Introduction

The number of CPU cores on a chip has been increasing rapidly. Intel plans to have an architecture that can scale up to 1,000 cores on a single processor [1]. However, the improvement speed of memory bandwidth, an important shared resource, cannot keep up with the increasing number of cores. For example, 8 cores on Xeon E5-2690 in 2012 share 51.2GB/sec memory bandwidth, where 12 cores on Xeon E5-2695v2 introduced in 2013 share 59.7GB/sec memory bandwidth [2]. Although the number of CPU cores increased by 50%, the memory bandwidth only increases by about 16%. Considering this trend, Patterson anticipated that off-chip memory bandwidth will often be the constraining resource in system performance [3].

In this environment, numerous tasks can be concurrently executed on increasing number of cores. The tasks share memory subsystems such as Last Level Cache (LLC) and memory bandwidth while the sharing can lead to memory contention which makes system performance unpredictable and degraded [4] [5] [6] [7].

Several methods have been presented for mitigating the contention. Among these methods, task classification is the most well known example. This method decides which tasks share same LLC and memory bandwidth and avoids the worst cases by minimizing performance interference [4] [6] [8] [9]. However, task classification cannot mitigate the contention when most of running tasks are memory intensive tasks on a physical node as Ahn et al. [10] pointed out. In this paper, we present Bubble Task which can dynamically recognize and mitigate unavoidable memory contention via throttling approach. It can reduce concurrent memory access and thus improve resource efficiency. Also, memory bandwidth can be more fairly distributed by our throttling policy.

The primary contributions of this paper are the following:

- We presented a simple model, memory contention model, to show the level of contention. Bubble Task Scheduler can efficiently decide per-core throttling ratio with respect to current contention level.
- We evaluated Bubble Task by using STREAM and lmbench benchmark suites which are the most well known of memory stressors. Bubble Task can lead to performance improvement in both stress tests.
- We implemented both software and hardware versions Bubble Task. We compared the versions in terms of trade-off between fine-grained adjustment and stable fairness.

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 while a task is the sole owner of a CPU core. The rest of the paper is organized as follows: Section 2 discusses the related work. Section 3 presents our contention model and describes our Bubble Task Scheduler. Section 4 shows the experimental results. Section 6 concludes this paper.

2 Related Work

Various solutions have been developed to mitigate the contention for shared resources via scheduling. Jiang et al. [8] presented the methodology 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. [9] 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. [4] 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. 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 [11] 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 an major factor causing performance degradation.

The methodology proposed by Kim et al. [6] 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. However, this classification methods cannot make effect in cases where there are so many memory intensive tasks. Bubble Task can throttle specific cores and can lead to the mitigation of memory subsystems.

Ahn et al. [10] presented a migration method among physical nodes in virtualized environments to deal with the unavoidable cases and their method also avoids remote accesses on Non-Uniform Memory Access architecture via VM live migration. However, Bubble Tasks is an intra-node method which does not consider virtualization and migration method among physical nodes.

Throttling method was presented by Zhang et al. [12] to control the execution speed or resource usage efficiency before us. They proposed hardware execution throttling method using Intel's duty-cycle modulation mechanism [13]. Their hardware approach can lead to more stable fairness due to its fine-grained execution speed regulation than previous software approach presented by Fedorova et al. [14]. However, they did not present dynamic throttling policy. Bubble Task method adapts per-core throttling by using our memory contention model which can dynamically decide how much a CPU core should be throttled. We refered to their approach and implemented both software and hardware versions of Bubble Task.

3 Bubble Task

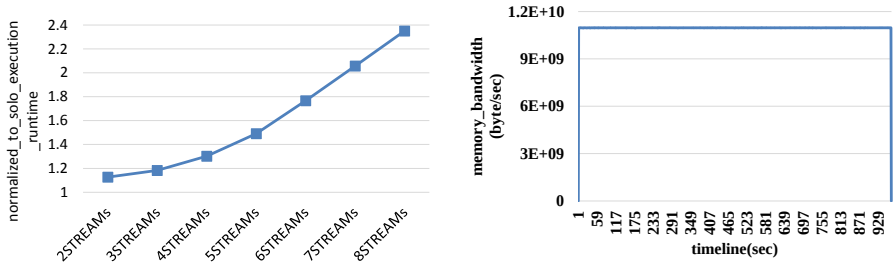
Bubble Task dynamically recognizes memory contention and decides the degree of per-core throttling ratio with respect to current contention level. In this section, we will introduce a new memory contention model and also show the correlation between our model and the performance to demonstrate the usefulness of the model. Lastly, we present our dynamic Bubble Task policy.

3.1 Memory Contention Modeling

The contention for memory subsystems degrades performance. Figure 1 (a) presents the runtime results with respect to the number of concurrently executing STREAM applications on 8cores Xeon E5-2690. STREAM benchmark is the most of the memory hungry application and the memory access patterns is invariable from start to end as seen in Figure 1 (b). 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.

The more the number of STREAMs, the more exponentially the performance of STREAM applications degrades due to the high memory contention. In this section, we will propose a new memory contention model which efficiently indicates the degree of the contention for memory subsystems and evaluate the correlation between our contention model and performance.

Figure 2 shows the average memory bandwidth and memory request buffer full rate with respect to the number of STREAM applications [16] on 8cores Xeon E5-2690. The memory bandwidth is the amount of retired memory traffic multiplied by 64bytes(size of a cacheline) [18], and Intel provides off-core response events which can permit measuring retired memory traffics [19]. 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 for memory subsystems and several components play an important role [5]. Retired memory traffic is thus a good metric to monitor the overall utilization of memory subsystems including LLC, prefetcher and memory controller.



(a) **The runtime results with respect to the number of co-executing applications**

(b) **Memory bandwidth of solo execution**

Fig. 1. Runtime comparision and memory bandwidth of STREAM benchmark

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. 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. To measure it, we monitored SQ_FULL event [19]. 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 2 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 2. 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.

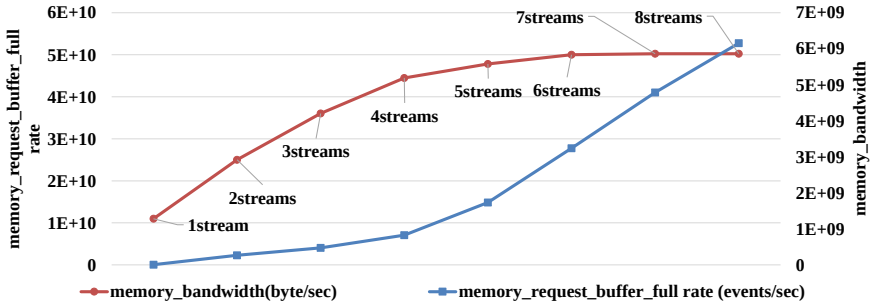


Fig. 2. The correlation between memory request buffer full rate and memory bandwidth with respect to the number of STREAM applications on 8core Xeon E5-2690

To evaluate the correlation between our memory contention model and performance, we did the stress tests for memory subsystems similar to [15] on 2 different CPU platforms. The specifications of our CPU platforms are organized in Table 1. We designated a STREAM application as a stressor because a stream application has high memory intensity and the memory intensity of a STREAM application is invariable from start to end. 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 3 and 4). We do the stress tests on both an i7-2600 (with four cores) and a Xeon E5-2690 (with eight cores). 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 [6]. 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 results are shown in Figure 3(for E5-2690) and 4(for i7-2600). 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 results demonstrate that memory contention level effectively indicates the contention degree closely correlated with the performance of target application.

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

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

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
Microacrcitecture	Sandy-bridge	Sandy-bridge

Table 1. The specifications of our SMP CPU platforms

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

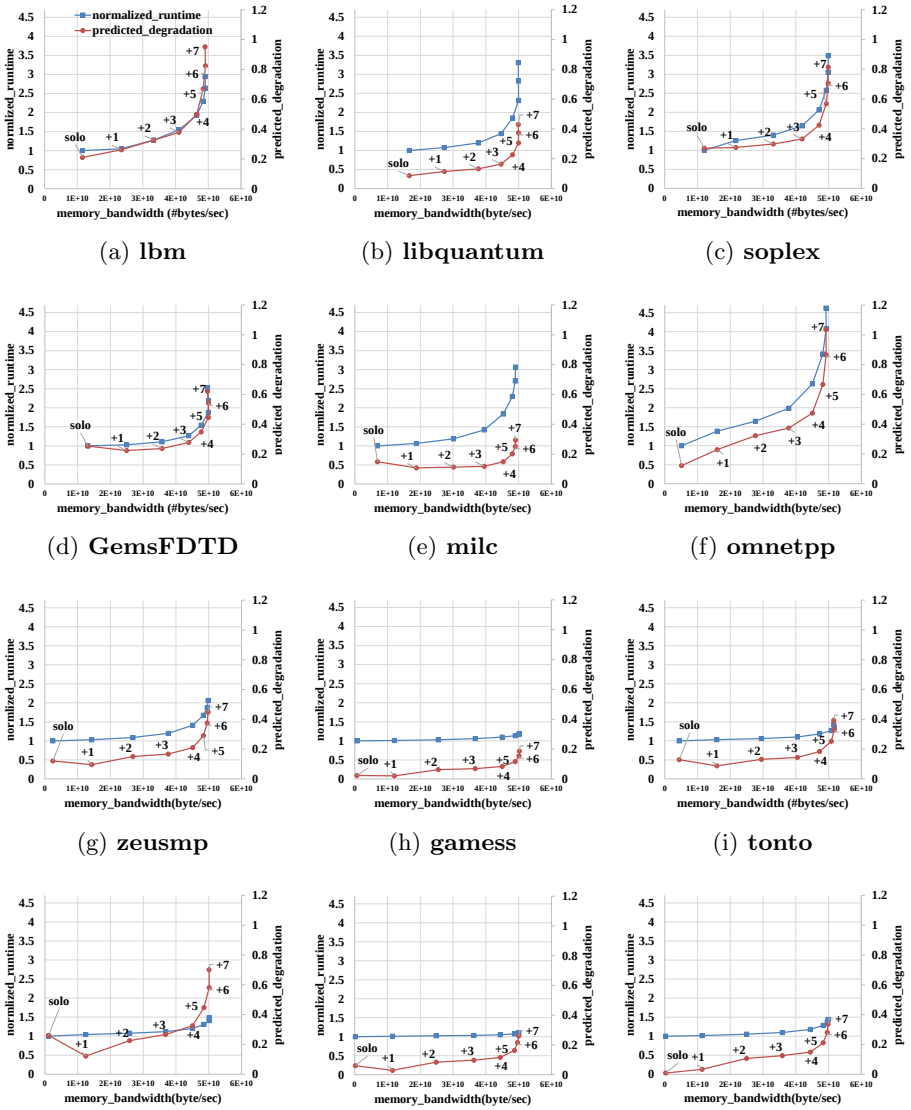
3.2 Dynamic Bubble Task Policy

In Bubble Task policy, current throttling load is determined with respect to current contention level and the load is distributed between CPU cores in proportional to the per-core intensity ratio during the interval(1 second). Dynamic Bubble Task policy is organized in Algorithm 1 and Bubble Task architecture is described in Figure 5.

Current contention load is dynamically calculated with respect to current contention level(Line 3 and 4). We adapted 21 and 200 for E5-2690 CPU in maximum contention level and maximum throttling load, respectively. Average intensity is the average traffic among CPU cores during the interval(Line 5) and the CPU cores generating more memory traffics than average intensity are selected as antogonists which should be throttled in next step. Current throttling load is distributed between the cores in proportional to per-core memory traffic ratio(From Line 6 to Line 15).

We implemented both software and hardware versions of Bubble Task. In software version, Bubble Task Scheduler forks artificial Bubble Tasks which execute infinite while loop and dynamically adjusts the nice value of Bubble Task to throttle target application on the same core. Bubble Task Scheduler sends new nice value with respect to per-core throttling ratio to corresponding Bubble Task and it adapts the nice value by using set priority system call. Figure 6 presents throttling ratio of target application with respect to the nice value of Bubble Task for CPU-bound application. Software version of Bubble Task can fulfill fine-grained throttling control. However, it cannot throttle target application without variation because software version requires context-switch between target application and corresponding Bubble Task (context-switch overhead) [12].

In contrast to software version, hardware version can fulfill precise per-core throttling control without variation by using IA32_CLOCK_MODULATION reg-



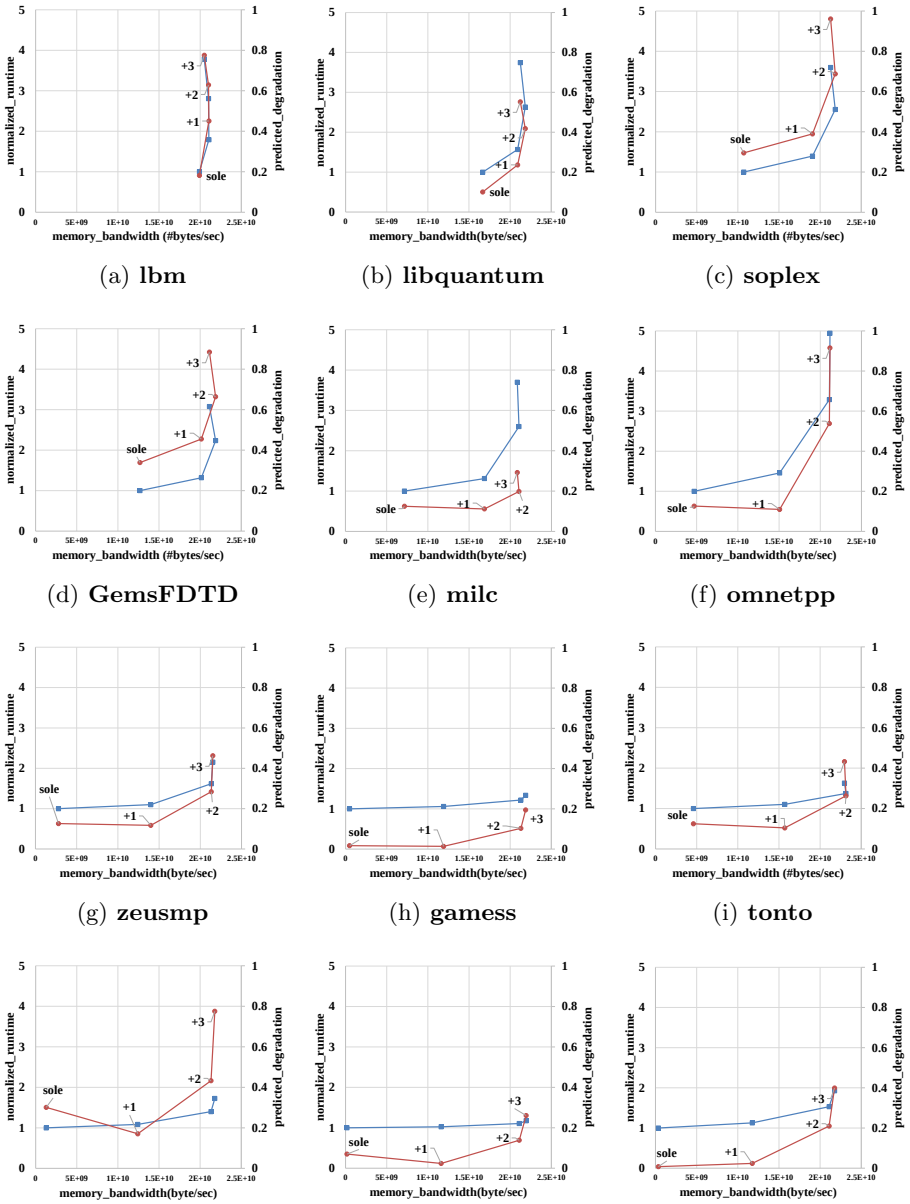
(a) **lbm** (b) **libquantum** (c) **soplex**

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

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

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

Fig. 3. The correlation between memory bandwidth, memory contention level and performance. Figures (a) - (l) show the results for 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**

Fig. 4. The correlation between memory bandwidth, memory contention level and performance. Figures (a) - (l) show the results for i7-2600

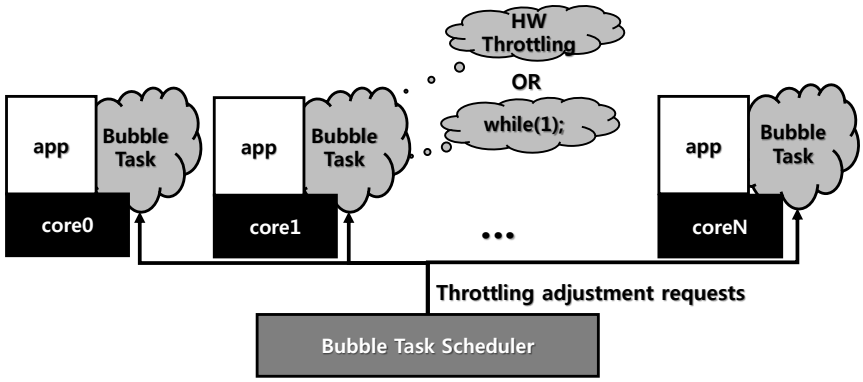


Fig. 5. Bubble Task architecture

ister [19]. However, this method can throttle the CPU utilization of target application by 12.5%(87.5%, 75%, 63.5%, 50%, 37.5%, 25% and 12.5%) and hence it can be considered as more coarse-grained throttling control than software version. There is trade-off between software and hardware versions. We will evaluate and compare two versions in Section 4.2.

Algorithm 1: Dynamic Bubble Task Scheduling

```
1:  Dynamic_Bubble_Task_Scheduling() begin
2:  current_contention_level = get_contention_level()
3:  current_throttling_ratio = current_contention_level / max_contention_level
4:  current_throttling_load =
    current_throttling_ratio × max_throttling_load
5:  avg_intensity = get_avg_intensity()
6:  for each c in all CPU cores do
7:      if avg_intensity ≤ c.mem_traffic then
8:          intensity_sum += c.mem_traffic
9:          throttling_list.insert(c)
10:     end if
11: end for
12: for each c in throttling_list do
13:     per_core_throttling_ratio = c.mem_traffic / intensity_sum
14:     c.throttle(per_core_throttling_ratio × current_throttling_load)
15: end for
```

4 Evaluation

First, we did stress tests by executing STREAM [16] and lmbench [17] applications on 8cores Xeon E5-2690. Identical eight stressors, STREAM or lmbench, are pinned on each core and Bubble Task Scheduler dynamically throttle the CPU utilization of the stressors during the runtime. We monitored five perfor-

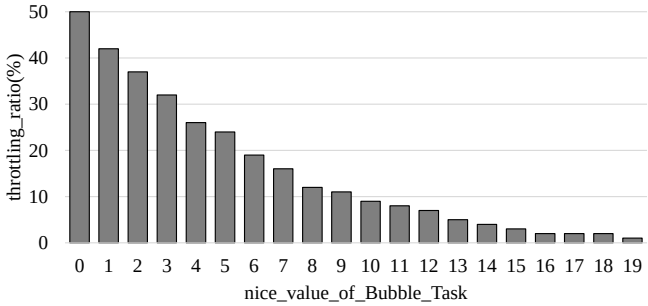


Fig. 6. Per core throttling ratio with respect to nice value of software version Bubble Task

mance metrics, retired_memory_traffic rate, LLC_miss rate, LLC_reference rate, memory_request _buffer_full rate and runtime and next compared Bubble Task method with naive Linux scheduler. We also executed four pairs of SPEC CPU 2006 applications, 2lbms, 2libquantums, 2GemsFDTDs and 2soplexs and compared the bandwidth fairness between software version, naive and hardware version.

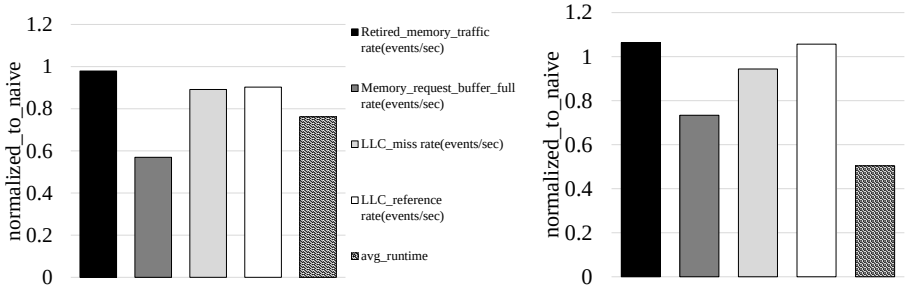
4.1 Stress Test

We executed STREAM and lmbench sets which are known as the best of memory stressor to identify how Bubble Task method can mitigate the contention for memory subsystem and hence improve system performance. We used software version Bubble Task method in this section because software version can fulfill fine-grained throttling control. The major difference between software and hardware versions will be handled in Section 4.2. We monitored 5 performance metrics on E5-2690 and the results are presented in Figure 7. We normalized the results of Bubble Task method to naive.

In case of STREAM, LLC reference rate and LLC miss rate decrease by about 10% (memory bandwidth also decreases by about 1%). In particular, memory request buffer full rate decreases by 43%. Memory request buffer full is a metric which can indicate the contention level for memory controller. The mitigation for memory controller leads to performance improvement (about 21%) in STREAM case because the contention for memory controller is more fatal in performance than other resources [4] [5].

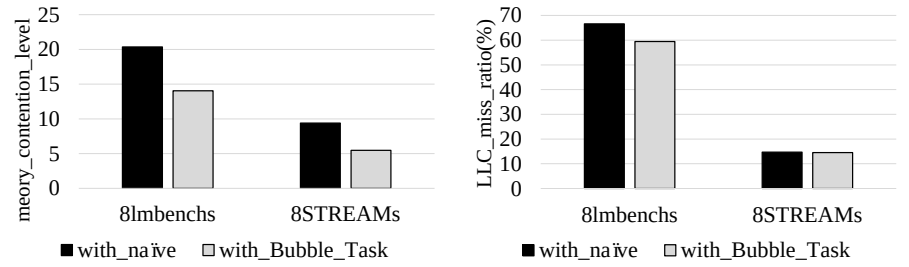
However, the reduction ratio of memory request buffer full rate is higher in STREAM than lmbench but, performance improves more in lmbench case(about 50%). To establish the cause, we compared the absolute results between STREAM and lmbench cases in Figure 8, not normalized results. Fundamentally, average memory contention level presented in Equation 1 is higher in lmbench and absolute mitigation degree of memory contention is higher in lmbench although

relative mitigation degree is lower in lmbench. Also, LLC miss ratio is higher in lmbench and absolute reduction ratio of LLC miss is more noticeable. As a result, we conclude that Bubble Task can take effect more in high memory contention case.



(a) **STREAM** (b) **lmbench**

Fig. 7. Performance metric results of STREAM and lmbench applications



(a) **average memory contention level** (b) **average LLC miss ratio**

Fig. 8. Performance analysis between lmbench and STREAM

4.2 Memory Traffic Fairness

Although we can identify that Bubble Task method can mitigate the contention through stress tests, it can also improve the fairness of memory traffic among executing tasks because Bubble Task Scheduler uses the policy which throttles the tasks generating more memory traffics than average traffic as seen in Algorithm 1(Line 13 and 14). In this section, we evaluated how Bubble Task

method can guarantee the fairness of memory traffic among executing tasks and compared between software and hardware versions.

To evaluate fairness, we executed 4 pairs of SPEC CPU 2006 applications which have different memory access patterns with software version, naive and hardware version and pinned an application on a core to prevent the contention for CPU usage. We monitored per-task memory traffic from start to 1000 seconds. Fairness results are presented in Figure 9.

With hardware version, standard deviation of memory traffic between executing tasks is lower than naive by about 17% as seen in Figure 9 (b) and (c). The victim tasks, soplex_0 and soplex_1, can occupy more memory traffic because Bubble Tasks Scheduler throttles other tasks unfairly occupying memory traffic. Memory traffic can be distributed more fairly between executing tasks with hardware version.

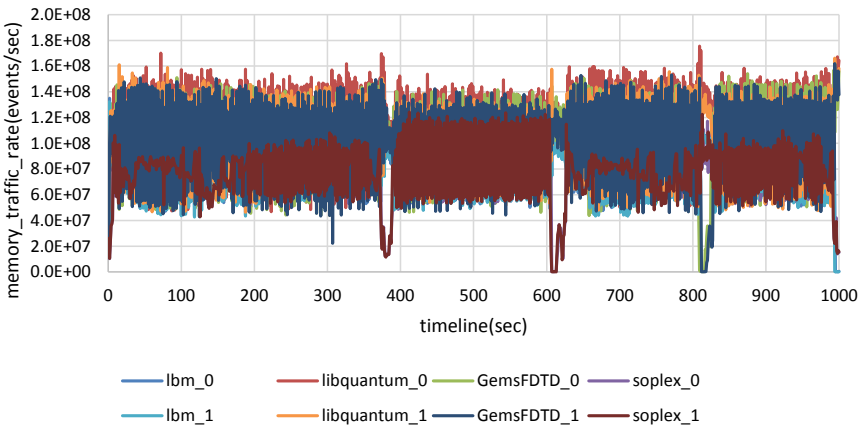
However, memory traffic of executing tasks is very changeable with software version as seen in 9 (a) although victim tasks can occupy more memory traffic. For example, the maximum memory traffic rate of soplex_1 with software version is much higher than hardware version but, the minimum memory traffic rate is lower. The memory traffic of all running tasks seriously fluctuates. Software version can fulfill fine-grained throttling and precisely decrease CPU utilization of high memory traffic tasks by adjusting the nice value of per-core Bubble Task while context-switch overhead makes memory traffic fluctuating.

When we compare the runtime results between software and hardware versions, it becomes more clear how software version aggressively fulfills throttling as seen in Figure 10. To figure out the overall effect of Bubble Task 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. Software version can lead to more performance improvement in soplex_0 and soplex_1, the most victims, than hardware version. However, other applications are more degraded with software version because software version aggressively decreases the CPU utilization of high memory intensive tasks via fine-grained throttling control. Average performance is a little bit worse with software version due to context switch overhead and unstable memory traffic.

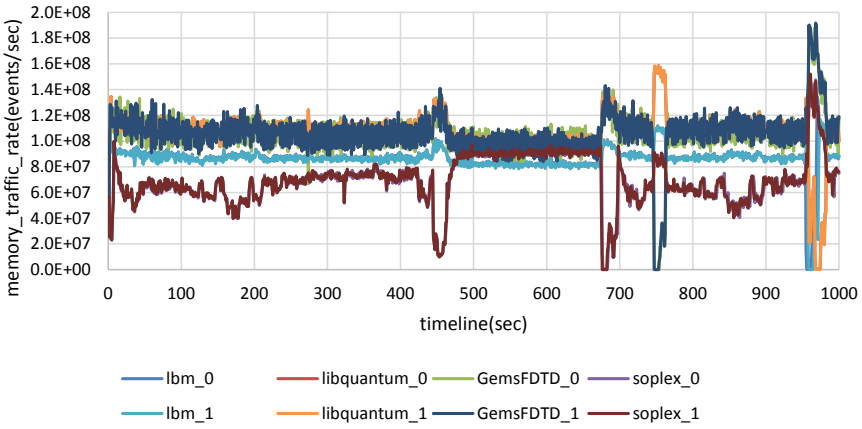
The improvement in system performance did not happen with both Bubble Task versions because the SPEC workload set cannot generate the high memory contention seen in previous stress tests. However, average performance degradation ratio is under 2% while the fairness of memory traffic improves by about 17% with hardware version.

5 Conclusion

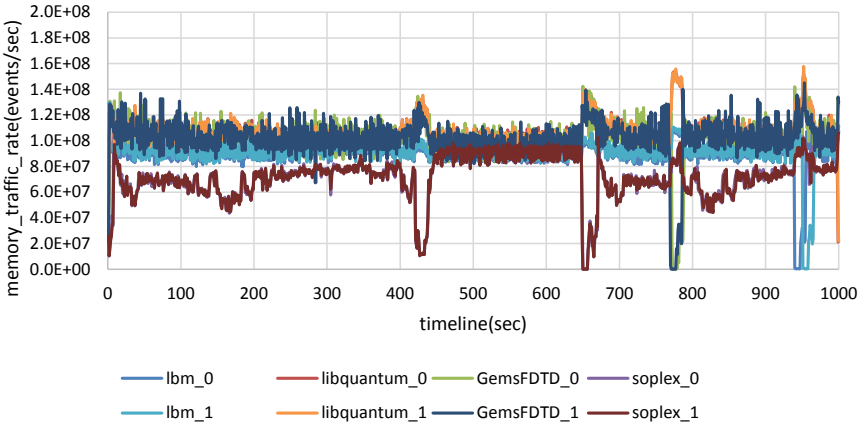
In this paper, we presented memory contention model and dynamic Bubble Task policy. Bubble Task method dynamically decides current throttling load with respect to current contention level and calculates per-core throttling load in proportional to the memory intensity ratio among CPU cores during the interval.



(a) **With software version**



(b) **With naive**



(c) **With hardware version**

Fig. 9. Bandwidth fairness

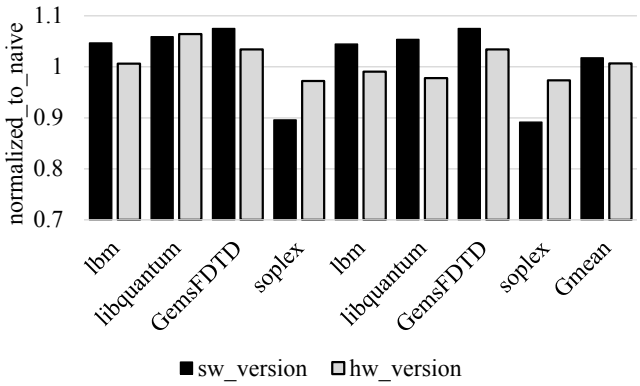


Fig. 10. Performance comparision

We implemented and evaluated both software and hardware versions of Bubble Task. Software version forks Bubble Tasks executing infinite while loop and co-locate them with the tasks unfairly occupying memory bandwidth. Bubble Task shares timeslice of the target task via context-switch mechanism. In contrast, hardware version uses IA32_CLOCK_MODULATION, Intel's model specific register, and can throttle target task without context switch overhead.

The two versions have pros and cons each other. Software version provides fine-grained throttling through the nice value control of Bubble Task and can throttle the CPU utilization of target application more precisely while it cannot guarantee stable fairness of memory traffic due to context switch overhead and unstable memory traffic. In contrast, hardware version can guarantee stable bandwidth fairness through low-overhead hardware throttling control although hardware version provides coarse-grained throttling control.

Through stress test, we can conclude that the more serious the contention for memory subsystems, the more Bubble Task method can take effect as seen in Figure 7 and 8. Future multi-core CPU will be developed with the increasing number of CPU cores but, memory bandwidth cannot catch up with the CPU development speed. Thus, the contention for memory subsystems will occur more and more and then Bubble Task method will contribute to mitigate the contention.

6 Acknowledgement

This research was supported by Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Education (NRF-2013R1A1A2064629). It was also supported by Next-Generation Information Computing Development Program through the National Research Foundation of Korea(NRF) funded by the Ministry of Science, ICT & Future

Planning (No. 2010-0020731). The ICT at Seoul National University provided research facilities for this study.

References

1. http://www.neoseeker.com/news/15313-1000-core-processor-possible-says-intel/.
2. http://ark.intel.com/products/.
3. David Patterson. Latency lags bandwidth. In Communication of the ACM, 2004.
4. Sergey Zhuravlev, Sergey Blagodurov and Alexandra Fedorova. Addressing Shared Resource Contention in Multicore Processors via Scheduling. In ASPLOS, 2010.
5. 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.
6. Shin-gyu Kim, Hyeonsang Eom and Heon Y. Yeom. Virtual machine consilidation based on interference modeling. In the journal of Supercomputing, April 2013.
7. Andreas Merkel, Han Stoess and Frank Bellosa. Resource-conscious Scheduling for Energy Efficiency on Multicore Processors. In EuroSys, 2010.
8. Y. Jiang, X. Shen, J. Chen, and R. Tripathi. Analysis and Approximation of Optimal Co-Scheduling on Chip Multiprocessors. In PACT, 2008.
9. Y. Xie and G. Loh. Dynamic Classification of Program Memory Behaviors in CMPs. In Proc. of CMP-MSI, held in conjunction with ISCA, 2008.
10. Jeongseob Ahn, Changdae Kim and Jaeung Han. Dynamic Virtual Machine Scheduling in Clouds for Architectural Shared Resources. In HotCloud, 2012.
11. D. Chandra, F. Guo, S. Kim, and Y. Solihin. Predicting InterThread Cache Contention on a Chip Multi-Processor Architecture. In HPCA, 2005.
12. Xiao Zhang Sandhya Dwarkadas Kai Shen. Hardware Execution Throttling for Multi-core Resource Management. In ATC, 2009.
13. A. Naveh, E. Rotem, A. Mendelson, S. Gochman, R. Chabukswar, K. Krishnan, and A. Kumar. Power and thermal management in the Intel Core Duo processor. Intel Technology Journal, 10(2):109122, 2006.
14. A. Fedorova, M. Seltzer, and M. Smith. Improving performance isolation on chip multiprocessors via an operating system scheduler. In 16th Intl Conf. on Parallel Architecture and Compilation Techniques, pages 2536, Brasov, Romania, Sept. 2007.
15. 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.
16. http://www.cs.virginia.edu/stream/.
17. http://www.bitmover.com/lmbench/.
18. http://http://software.intel.com/en-us/articles/detecting-memory-bandwidth-saturation-in-threaded-applications.
19. Intel(R) 64 and IA-32 Arhcitectures Software Develper's Manual, Volume 3B. System Programming Guide, Part 2.
20. http://www.spec.org/cpu2006/.