On Mitigating Memory Bandwidth Contention through Bandwidth-Aware Scheduling

Di Xu[1][2], Chenggang Wu[1]*
[1]Key Laboratory of Computer System and Architecture, Institute of Computing Technology, Chinese Academy of Sciences, Beijing, China
[2]Graduate University of Chinese Academy of Sciences, Beijing, China
{xudi, wucg}@ict.ac.cn

Pen-Chung Yew
Department of Computer Science and Engineering, University of Minnesota at Twin-Cities, Minnesota, USA
Institute of Information Science, Academia Sinica, Taiwan
yew@cs.umn.edu

ABSTRACT

Shared-memory multiprocessors have dominated all platforms from high-end to desktop computers. On such platforms, it is well known that the interconnect between the processors and the main memory has become a major bottleneck. The bandwidth-aware job scheduling is an effective and relatively easy-to-implement way to relieve the bandwidth contention. Previous policies understood that bandwidth saturation hurt the throughput of parallel jobs so they scheduled the jobs to let the total bandwidth requirement equal to the system peak bandwidth. However, we found that intra-quantum fine-grained bandwidth contention still happened due to a program's irregular fluctuation in memory access intensity, which is mostly ignored in previous policies.

In this paper, we quantify the impact of bandwidth contention on overall performance. We found that concurrent jobs could achieve a higher memory bandwidth utilization at the expense of super-linear performance degradation. Based on such an observation, we proposed a new workload scheduling policy. Its basic idea is that interference due to bandwidth contention could be minimized when bandwidth utilization is maintained at the level of average bandwidth requirement of the workload. Our evaluation is based on both SPEC 2006 and NPB workloads. The evaluation results on randomly generated workloads show that our policy could improve the system throughput by 4.1% on average over the native OS scheduler, and up to 11.7% improvement has been observed.

Categories and Subject Descriptors

D.4.1 [**Operating Systems**]: Process Management—*Scheduling*

*To whom correspondence should be addressed.

General Terms

Algorithms, Management, Measurement, Performance

Keywords

Process scheduling, memory bandwidth, bus contention

1. INTRODUCTION

Shared-memory multiprocessors, such as recent multi-core processors, are widely used as a cost-effective way to build high performance servers. However, it is well known that the interconnect bandwidth between processors and main memory has become a major bottleneck that hindered their scalability [6]. As the trend is to put dozens, or even hundreds of cores on one chip [11], the interconnect bandwidth is being pushed to its practical limit [14]. Consequently, the available bandwidth to access memory (called *memory bandwidth* for the rest of the paper) will continue to be a critical issue for such systems.

Numerous approaches have been proposed to reduce the requirement of a program to access memory, thus mitigating the memory bandwidth bottleneck, for example, improving data locality and eliminating useless prefetches [8] in a program. However, as long as the working-set size of a job exceeds the size of the on-chip cache, the effectiveness of such approaches is quite limited.

Bandwidth-aware job scheduling has been proposed as an effective strategy to the above approaches when the number of concurrent jobs exceeds the number of processors on a multi-core system [12][4][5]. It attempts to predict the bandwidth requirement of a job, and selects synergistic concurrent jobs to avoid bandwidth saturation while keeping full utilization of the available bandwidth. A bandwidth-aware scheduling policy can thus improve the system throughput without the expensive bandwidth increase. It has been proposed as a stand-alone technique, or has been used in processors with hardware multi-threading [15] and clusters [10].

In most bandwidth-aware scheduling schemes, *job segments*, i.e., segments of code executed during a *scheduling quantum*, are usually used as scheduling units. To the best of our knowledge, all existing policies try to select and schedule job segments that could maximize their total bandwidth requirement to as close to the peak memory bandwidth provided on the system as possible, denoted as PBW (stands for *peak bandwidth*) for the rest of the paper. It is based on the premise that co-scheduled job segments could fully utilize the available bandwidth on the system, and experience little performance degradation before the peak system bandwidth is reached.

However, when we examined the above premise in more detail, we found it was hardly the case. Concurrent job segments actually suffer significant slowdown way before their combined bandwidth requirement reaches the peak bandwidth. Further analysis also shows that the memory accesses are issued rather irregularly during program execution. A typical scheduler uses the *average* bandwidth requirement in a scheduling quantum to guide its scheduling. Hence, it is only aware of the changes in bandwidth requirement *between* scheduling quanta, but is totally unaware of the potential huge bandwidth fluctuation *within* a scheduling quantum. Consequently, severe bandwidth contention could still happen *within* a quantum even if their combined *average* bandwidth is below the peak bandwidth.

In this paper, we use a job segment extraction and simulation scheme to quantify the effect of the bandwidth contention. We evaluate a large number of random job segment compositions from the SPEC CPU2006 floating point benchmarks. It shows that due to the bandwidth contention, concurrent job segments approach the system peak bandwidth often at the expense of super-linear slowdown. Based on this observation, we propose a new scheduling policy, which uses an *ideal average bandwidth requirement* (IABW) to set the *scheduling bandwidth target*, so that the workload's global bandwidth utilization is kept steady during execution. It results in a better overall performance.

We evaluate the effectiveness of our policy using a user-level scheduler on Linux. The workloads are randomly generated using both the SPEC 2006 and NPB suite. The evaluation results show that, the throughput could be improved by up to 11.7% over the native OS scheduler, and the average is 3.4% and 4.9% for the SPEC workloads and NPB workloads, respectively.

Our contributions are:

- As far as we know, this is the first work that quantifies the effect of bandwidth contention *within* each scheduling time quantum on overall performance. We re-evaluate the performance of concurrent jobs when they share the limited memory bandwidth. We show that the rate of performance degradation is super-linear when the overall bandwidth requirement approaching the system peak bandwidth.

- A scheduling policy that could mitigate such impact is proposed. It controls system-wide bandwidth utilization according to an IABW of the workload. We verify our idea both with a mathematical proof and with real system experiments using random workloads. The scheduling scheme is simple yet quite effective in improving the system throughput.

- We improve runtime bandwidth prediction using several techniques that include filtering out bandwidth utilization when the bandwidth is over-saturated, and a phased-based bandwidth prediction.

The rest of this paper is organized as follows. In section 2, we evaluate the impact of memory bandwidth fluctuation on bandwidth contention. Based on the evaluation and analysis, we introduce our scheduling policy in section 3. Sections 4 and 5 present the scheduler and its evaluation results, respectively. In section 6, we introduce related work. Finally, we conclude the paper in section 7.

2. THE IMPACT OF FLUCTUATION IN MEMORY ACCESSES

2.1 Methodology

Our evaluation and analysis are performed on a typical shared-memory system with 4 dual-cores Itanium-2 (Montecito) processors at 1.6GHz [16]. There is no shared cache among the cores. We also turn off the hardware multi-threading on processors. It is equipped with 16GB PC2100 memory. The *theoretical peak bandwidth* of the front-side bus (FSB) is 6.4GB/s. Each memory access requires a *bus transaction*, and each transaction accesses a cache line. The last-level cache line size is 128 bytes, so the theoretical peak *bus transaction rate* (BTR) of FSB is 50 trans./usec. However, the realistic BTR, as measured by Stream benchmark [3], is 40 trans./usec.

It runs Linux 2.6.9. We use *perfmon2* and its corresponding *libpfm* library [2] to access the performance counters. SPEC 2006 floating point benchmarks are used because there are more memory-intensive. The SPEC benchmarks are compiled by Intel compiler v10.1, with optimization flag -fast. We also created some micro-benchmarks for our evaluation and analysis, whose details are in the next section.

2.2 Performance of Various Memory Access Patterns

The first experiment shows an ideal behavior when concurrent jobs share the limited memory bandwidth. We produce a micro-benchmark called *Hog*, whose kernel loop is shown in Figure 1(a). It builds two data arrays and continuously copies data from one to the other. The access stride equals to the last-level cache line size. Also, the size of each array is twice as large as the last-level cache. As a result, all copy operations will generate cache misses and thus memory accesses. We adjust the number of *nop* bundles between two copy operations to create different bandwidth requirements. In each case, we run two instances of *Hog*s in parallel, and evaluate the system-wide BTR and the weighted speedup.

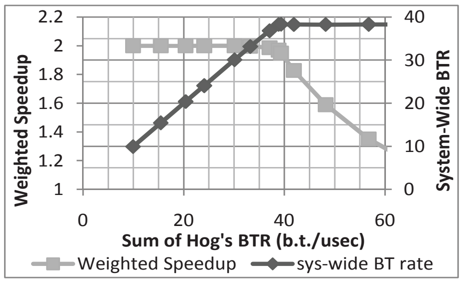
$$WeightedSpeedup = \sum \frac{RUNTIME_{alone}}{RUNTIME_{shared}} \quad (1)$$

The results are shown in Figure 1(b). As the memory bandwidth requirement increases, the system-wide BTR increases linearly in the beginning. An obvious inflection point is observed when the system-wide BTR reaches around 38.2 trans./usec. It then stalls even when the bandwidth requirement of *Hog*s continues to increase because the bus is saturated. The weighted speedup remains the same as before the bus saturation at 2. Afterwards, it decreases significantly. This behavior confirms the premise of existing scheduling policies described above, i.e. parallel jobs nearly don't interference each other until peak bandwidth is reached.

However, the reason for such an ideal behavior is due to the stability of memory access intensity. In this case, the average memory access latency is about 350 cycles. The number of *nop*s is 1k at most. Hence, the period of a copy operation and the following *nop*s is no more than 2k CPU cycles. In another word, *Hog*'s BTR is held very steady in any period longer than 2k cycles.

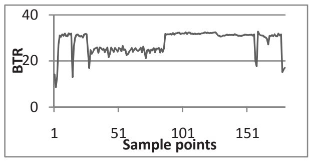
```
/* Hog */
int array_a[N][CACHE_LINE_SIZE/sizeof(int)];
int array_b[N][CACHE_LINE_SIZE/sizeof(int)];
while (1) { //last long enough: at least for 10 secs
  for (j=0;j<N;j++) {
    array_a[j][0] = array_b[j][0];
    for (k=0; k<idle; k++)
      asm("nop");
}}}
```

(a) Kernel loop

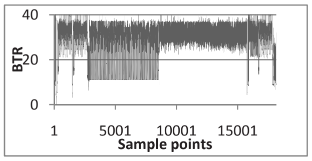


(b) Parallel performance

Figure 1: Micro-benchmark: *Hog*.



(a) Interval=100ms



(b) Interval=1ms

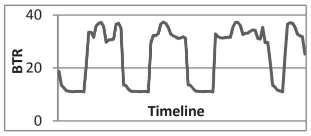
Figure 2: BTR sampling of 433.milc at different intervals.

In comparison, the memory access patterns in real applications could be very bursty and irregular. We take 433.milc as an example. Figure 2 shows its BTR sampled with different sampling rates. From Figure 2(a), we find that even at a relative large sampling interval of 100ms, the BTR of adjacent intervals could vary significantly. As we shorten the sampling interval, the BTR fluctuates even more violently in Figure 2(b). In fact, memory accesses are always irregular and randomly distributed in very small intervals. If the scheduling quantum is 100 ms (same as the default time quantum of Linux kernel version 2.6), and we use the average BTR in that quantum to guide job scheduling as in exiting bandwidth-aware scheduling schemes, we will be made to believe that the program has a rather "steady" behavior as presented in Figure 2(a) instead of violent ones in Figures 2(b).

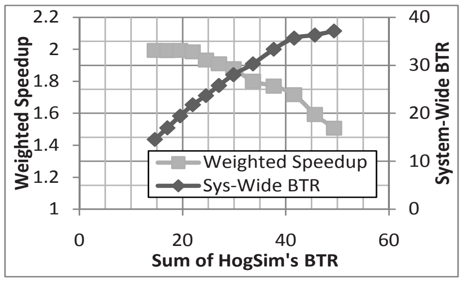
In order to quantify the impact of BTR fluctuations, we set the BTR sampling interval at 1 ms, and refer to it as a fine interval. The scheduling quantum remains as 100 ms. Our first step is to obtain an application's BTR using a fine interval while running alone. We then group the sampled BTRs into segments. Each BTR segment, denoted as α, corresponds to BTR samples collected during each scheduling quantum. There are 100 samples in each segment if the sampling interval is 1 ms and the scheduling quantum is 100ms, i.e., $\alpha =< a_1, a_2, ..., a_{100} >$. In order to simulate α, we modify the kernel loop of *Hog* to produce another micro-benchmark *HogSim*. Its kernel is shown in Figure 3(a).

```
/*HogSim*/
int array_a[N][CACHE_LINE_SIZE/sizeof(int)];
int array_b[N][CACHE_LINE_SIZE/sizeof(int)];
int gs[100], nop[100];
int cur_array_index = 0;
While (1) { // last long enough
  for (j=0; j<100; j++) {
    for (l=0;l<gs[j];l++) {
      a[(cur_array_index+l)%N][0] =\
        b[(cur_array_index+l)%N][0];
      for(i=0;i<nop[j];i++){ nop};
    }
    cur_array_index += gs[j];
}
```

(a) Kernel loop



(b) A input segment from 433.milc

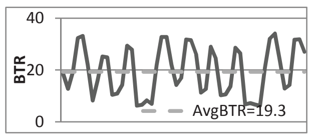


(c) Parallel performance

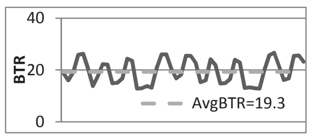
Figure 3: Micro benchmark: *HogSim*.

It divides the data copy operations into 100 groups. Each corresponds to a fine interval. The number of copy operations in each group is $1ms * a_i(trans./usec)/2 = 500 * a_i$ (2 bus trans. per copy operation). In order to achieve the desired BTR a_i, an estimated number of *nop*s are inserted after each copy operation. We estimated the number of *nop*s needed to generate a desired BTR in advance, and form a mapping table. For a given BTR a_i, we look up the table, and find its closest value and the corresponding *nop* number. This process is done offline. The results are organized into two arrays: *GS* and *NOP*, and saved in a file. When *HogSim* executes, it reads the file and initializes its own copy of *GS* and *NOP*.

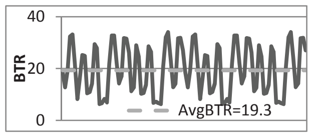
Similar to *Hog*, we measure the performance with two jobs of *HogSim* running concurrently. Figure 3(b) shows BTR variations in the evaluated segment from 433.milc (4k ms to 4.1k ms). The pattern is repeated nearly one third of the total execution of 433.milc. We could adjust its average BTR by shifting the entire curve up or down. Figure 3(c) shows that the behavior of the two concurrent *HogSim* jobs is very different from that of *Hog*s. The weighted speedup of two *HogSim* jobs starts to decrease when their combined bandwidth requirement reaches about 24 trans./usec, i.e. about 2/3 of the realistic peak bandwidth. The figure shows that, even before bus saturates, they cannot achieve a total BTR that equals to the sum of each individual BTR. As the bandwidth requirement increases, the bus utilization gradually approaches the realistic peak bandwidth. However, the



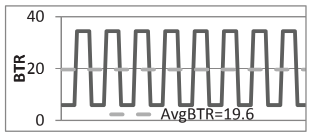
(a) Original segment and average BTR line



(b) Smaller fluctuation *degree* (y=0.5)



(c) Smaller fluctuation *period* (x=0.5)



(d) A derived regular segment

Figure 4: BTR Fluctuation (x-axis is time line).

bus contentions also increase, and the resulting weighted speedup suffers.

2.3 Impact of Fluctuation on System-Wide BTR and Speedup

The shapes of BTR curves vary significantly in the same time segment of different programs or in different segments of the same program. Even if a set of segments have the same average BTR, their BTR fluctuations may still differ in three aspects: *degree* of fluctuation, *period* of fluctuation and *regularity* of fluctuation. In this section, we study how these three factors influence the overall performance respectively. When we evaluate one of the factors, the others should remain unchanged. We design techniques to manipulate these factors for any given job segment $\alpha =< a_1, a_2, ..., a_{100} >$, whose average BTR is denoted as $\bar{\alpha}$.

The first aspect is the *degree* of fluctuation. It characterizes how much each a_i deviates from $\bar{\alpha}$. We manipulate the *degree* of fluctuation as follow: for each a_i, its distance from $\bar{\alpha}$ is scaled by a factor of y. The *degree* of fluctuation would increase if y is larger than 1, and decrease otherwise. As an example, Figure 4(a) illustrates an original segment taken from 437.leslie3d. The segment with its *degree* of fluctuation at 0.5 (y=0.5) is shown in Figure 4(b). The corresponding results measured on the real system are shown in Figure 5(a) and Figure 5(d). They show that, as the *degree* of fluctuation decreases, the weighted speedup and the system-wide BTR will behave more like those of *Hog*s. In another word, the impact of fluctuation becomes less significant.

The second aspect is the *period* of fluctuations. It is defined as the average length of time interval between two adjacent BTRs whose values change across $\bar{\alpha}$. For example, the BTR curve fluctuates across the line of $\bar{\alpha}$ 17 times in the segment shown in Figure 4(a), so the fluctuation *period* of this segment is defined as 100ms/17, or about 6ms. It is an inverse measure on the number of fluctuations occurs within a segment. Given a segment, we can reduce its fluctuation *period* by scaling down each element of array *GS* in *HogSim* by a factor of x (x<1 to maintain the same BTR average). Figure 4(c) shows the behavior of a new segment compared to Figure 4(a) when its fluctuation *period* is halved (i.e. x=0.5). We could see that the original segment is repeated 2 times because the duration of the BTR changes is halved. BTR thus fluctuates more frequently within a time quantum.

We evaluate the fluctuation *period* to see whether smaller fluctuation *period* has less significant impact on the parallel jobs. We observed that the fluctuation *period* is closely related to the third factor in our study - the *regularity* of fluctuations. Figure 4(d) shows a more regular segment we created from Figure 4(a). Its BTR fluctuates regularly between 34.4 trans./usec and 5.9 trans./usec (corresponding to the maximal and minimal BTR of the original segment) with a fixed *period* of 6ms.

The impact of varying fluctuation *degree* and fluctuation *period* on the weighted speedup and system-wide BTR of the concurrent jobs is shown in Figures 5(b), 5(e), 5(c), 5(f). Initially, due to increased fluctuation *degree*, the performance of regular segments in the concurrent jobs is worse than that of the original ones. However, when we decrease their fluctuation *period*s, the behavior of the regular segments in the concurrent jobs will approach those of *Hog*s, while the behavior of the original segments after decreasing the fluctuation *period*s remain nearly unchanged. This is because the bandwidth contention among the memory accesses in regular segments will force those segments to re-align themselves, and match a low BTR phase against a high BTR phase because the lengths of their *period*s are similar. This re-alignment is unlikely to happen in the original segments because their BTR fluctuates irregularly, and those high and low BTR periods of very different lengths are difficult to match and re-align. Our measured results show this is indeed the case. For the original segments, even if we reduce the fluctuation *period* by 1/500 to about 12us, the fluctuations still exert significant impact on the performance of the concurrent jobs.

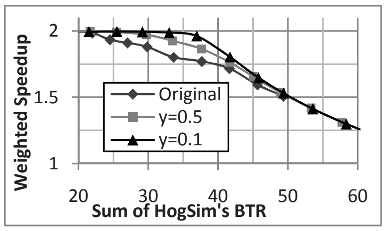
We have two conclusions from the evaluation result of fluctuation *period*. Firstly, our evaluated fine interval is 1ms in *HogSim*, and the memory accesses within each fine interval are as steady as *Hog*. As a result, we ignore the BTR fluctuation whose *period* is less than 1ms, so our evaluation result is still conservative, the practical impact of real job segments should no smaller than our evaluated results. Secondly, a straightforward solution for mitigating the impact of fluctuation is to decrease the size scheduling quantum. But above evaluations show the fluctuation *period* which cause significant impact may be very tiny, and it is not acceptable for a job scheduling framework for overhead issue.

Putting all of the factors together, we evaluate the impact of BTR fluctuation on general job combinations. We extract all job segments from all SPEC 2006 floating-point benchmarks, and randomly select 4 of them to run concurrently. The gray points in Figure 6 show the performance of 1k random combinations. Most of the compositions suffer performance (i.e. weighted speedup) degradation before their combined bandwidth requirements reaches the realistic peak bus bandwidth. The results also show that as the global bandwidth requirement increases linearly, the performance degradation goes super-linearly.

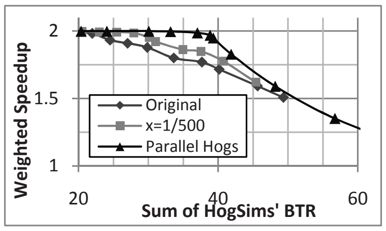
3. A NEW BANDWIDTH-AWARE SCHEDULING POLICY

3.1 Basic Idea

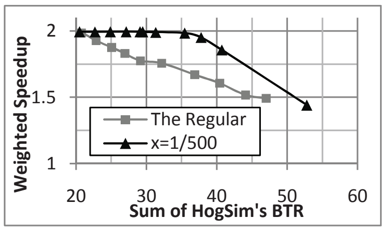
Before we discuss our new scheduling policy in detail, it is necessary to briefly introduce the existing bandwidth-aware scheduling policies. Equation (2) shows a typical fitness cal-



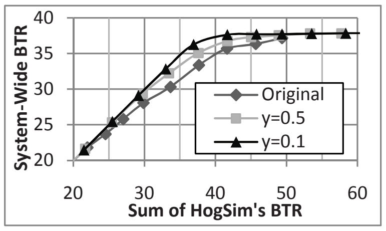
(a) Impact of fluctuation *degree* on weighted speedup



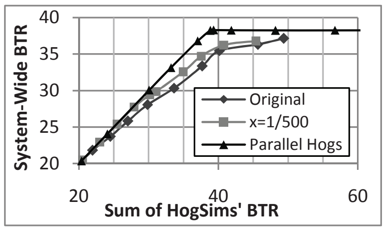
(b) Impact of fluctuation *period* on weighted speedup



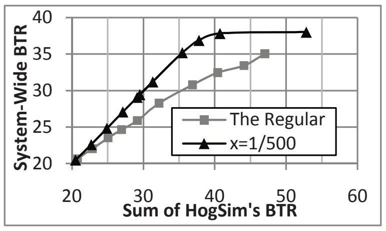
(c) Impact of fluctuation *period* on weighted speedup in regular segments



(d) Impact of fluctuation *degree* on system-wide BTR

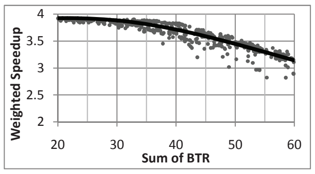


(e) Impact of fluctuation *period* on system-wide BTR

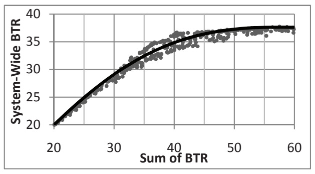


(f) Impact of fluctuation *period* on system-wide BTR in regular segments

Figure 5: Sensitivity to fluctuation *degree* and fluctuation *period*.



(a) Weighted Speedup



(b) System-Wide BTR

Figure 6: Parallel performance of 1k randomly selected job segment combinations. The solid lines draw the tendency of the points.

culation in most existing scheduling policies. It is based on a job's memory bandwidth requirements. BW_{remain} and CPU_{remain} are the remaining bandwidth and processors on the system that to be allocated to new jobs, respectively. The formula quantifies the gap between the bandwidth requirement of the job waiting to be scheduled and the available bandwidth. The job with a smaller gap has a better fit, thus receive a higher priority to be scheduled next. After a job is scheduled, its bandwidth requirement is subtracted from BW_{remain}. This step is repeated until all CPUs are assigned a job and busy. The objective is to maximize the total bandwidth utilization toward the peak bandwidth of the system [4][5][15][10].

$$FITNESS^P = \frac{1}{|\frac{BW_{remain}}{CPU_{remain}} - BW^P_{required}|} \tag{2}$$

The analysis and evaluation in Section 2 show that the irregular fluctuation in bandwidth requirement is quite common in real applications. Existing bandwidth-aware scheduling policies will schedule the job the best fit the remaining bandwidth on each CPU. It very often could cause severe bandwidth contention, and degrade overall performance, especially for workloads with low bandwidth requirement. In contrast, the basic idea of our new policy is to control the system-wide bandwidth utilization close to a carefully selected target level as much as possible, and balance the system-wide bandwidth utilization with overall job performance. Such a balance is determined by the average bandwidth requirement of the workload. The strategy is to set proper scheduling bandwidth target, i.e., the initial value of BW_{remain}.

3.2 Theoretical Background of the New Scheduling Strategy

We introduce some terminologies. Given a workload, we assume it executes on an ideal system with infinite memory bandwidth, i.e., the concurrent jobs would not suffer performance degradation caused by memory contention. The turnaround time of the workload is T_i. We call T_i the *ideal turnaround time* of the workload, and it is the lower bound of its turnaround time. We also define the *ideal average bandwidth* (IABW) of the given workload as the ratio between the total number of memory accesses and its *ideal turnaround time*. IABW quantifies the memory bandwidth demand of a workload. Note that IABW could be higher than the realistic peak bandwidth available on the system.

When a workload executes on a real system, bandwidth contention could cause its turnaround time to be larger than T_i. The objective is to minimize such performance degradation. We view the execution of a workload as a process of finishing all its memory requests. As we show in Figure 6, parallel jobs often suffer super-linear slowdown when they try to achieve a higher bandwidth utilization. We instead try to schedule jobs whose combined bandwidth requirement equals to the IABW of the workload even if there may be still bandwidth available on the system. This is to minimize the impact of bandwidth contention on overall system performance. We give some theoretical arguments for this approach as follows.

Assume a workload has two phases in a segment: one phase has high bandwidth utilization and the other has low bandwidth utilization. We first present our arguments based on such a simplified two-phase segment. It could then be easily extended to the entire execution time of the workload.

The total amount of memory accesses in the segment is M, and its *ideal turnaround time* is T. The execution time of the two phases is a and b, respectively, and the corresponding bandwidth utilization is B_a and B_b. They satisfy the following relations:

$$M = a \times B_a + b \times B_b \tag{3}$$

$$T = a + b \tag{4}$$

The amount of slowdown due to bandwidth contention in the segment could be described as:

$$\begin{aligned} TotalSlowdown &= a \times f(B_a) + b \times f(B_b) \\ &= a \times f(B_a) + b \times f(\frac{M - a \times B_a}{b}) = g(B_a) \end{aligned} \tag{5}$$

In the above equation, function $f(x)$ describes the amount of slowdown (i.e. performance degradation) if the level of bandwidth utilization is x. We denote the relationship between B_a and the total amount of slowdown as a function g. To find the inflection point of g, we let $g'(Ba) = 0$, where

$$\begin{aligned} g'(B_a) &= a \times f'(B_a) + b \times f'(\frac{M - a \times B_a}{b}) \times (-\frac{a}{b}) \\ &= a \times [f'(B_a) - f'(\frac{M - a \times B_a}{b})] \end{aligned} \tag{6}$$

By making $g'(B_a) = 0$, we have $B_a = B_b = M/(a + b)$. This is the inflection point of the function g. As the performance slowdown is super-linear increased, we have the following relationship for function f:

$$f'(x) < f'(y), \text{ for any } x<y \tag{7}$$

If B_a is less than $M/(a + b)$, B_b would be greater than $M/(a + b)$. Hence, $g'(B_a)$ would be less than zero. Similarly, $g'(B_a)$ is greater than zero when B_a is greater than $M/(a + b)$. As a result, when $B_a = B_b = M/T$, function g achieves its minimum, that is $T * f(M/T)$. In another word, if we try to make the memory bandwidth utilization evenly distributed in those two phases, the total amount of slowdown will be kept minimal.

This conclusion could still apply if we further divide each phase into finer phases. On the other hand, if we extend the segment to the full extent of the entire workload execution, the same conclusion also applies.

This result gives us a guideline that, instead of always targeting peak bandwidth utilization when we schedule jobs, we should aim for keeping the total bandwidth requirement at a steady level that equals to the IABW of the entire workload, if all possible. It could minimize performance degradation due to contention.

3.3 Estimation of the Optimal Scheduling Bandwidth Target

From the above arguments in Section 3.2, we need to determine a proper scheduling bandwidth target so we could minimize the overall performance degradation. Similar to what required on batch processing systems [19] and systems with QoS guarantee [9], our policy requires jobs to be submitted with its resource requirements specified, such as number of processors, estimated memory bandwidth, and estimated execution time assuming all resources are granted.

Assume there is a workload consists of N jobs executed on a system with C processors (N>C). For each job j, its estimated execution time is T_j, and its average BTR is B_j. On our platform, there is no shared cache among the cores. The total amount of memory accesses of the workload can be calculated as follow:

$$TotalMemoryRequest = \sum_{j=1}^{N}(T_j \times B_j) \tag{8}$$

For systems with an on-chip cache shared by the cores, the total number of memory accesses may be hard to estimate. Some widely-investigated techniques, such as cache partition [13], together with the miss-rate-curve (MRC) estimation [18], could also give good estimates of the total number of memory accesses even in a share-cache environment.

We also need to estimate the *ideal turnaround time* of the workload. For a set of concurrent jobs with different execution time, the *ideal turnaround time* depends on how jobs are scheduled on processors. In our study, we assume a round-robin policy is used for our estimation. In following section we will explain that the bandwidth-aware scheduling policies only partly follow this round-robin order, hence, the *ideal turnaround time* here is just a best-effort estimate. However, our results show that it is a good enough estimate to guide our bandwidth-aware scheduling.

$$IdealTurnaroundTime = \frac{\sum_{i=1}^{N-C} T_{n_i}}{C} + T_{n_N} \tag{9}$$

In above equation, jobs are ordered according to their T_j in increasing order, and their specified execution time forms a new sequence $T_{n_j}(1 \leq j \leq N)$. To further explain it, consider an example with 8 concurrent jobs running on 4 cores. Firstly, since jobs are run in a round-robin order and there are only 4 cores, the time the shortest job takes to finish is $T_{n_1} * 8/4$. After that, only 7 jobs left, and the time the second shortest job takes to finish is $(T_{n_2} - T_{n_1}) * 7/4$. The calculation is repeated until the number of remaining jobs is less than the number of processors. After that, the time the remaining jobs take to finish is $T_{n_8} - T_{n_5}$. Equation (9) adds those time components to come up with an estimation of the *ideal turnaround time*.

The IABW of the workload is calculated as follow:

$$IdealAverageBandwidth = \frac{TotalMemoryRequest}{IdealTurnaroundTime} \tag{10}$$

The estimated IABW cannot be used as the bandwidth target of our scheduling strategy. It is because the scheduling policy we proposed in Section 3.2 is based on the bandwidth requirement when each job runs alone. But at runtime, we can only get the bandwidth each job obtained when it competes with other concurrent jobs. The bandwidth each job obtained is usually smaller (often much smaller) than that observed when each job runs alone due to fine-grained bus contention or even bus saturation. In order to maintain the same scheduling decisions, a realistic bandwidth target for our scheduling policy should also be smaller than the estimated IABW. In our experiment, we tune the performance of some representative workloads by adjusting the bandwidth target, and quantify the relationship between the IABW and the selected bandwidth target. The result is further applied to other unknown workloads. More details are in Section 5.2.

4. SCHEDULER IMPLEMENTATION

4.1 Framework

In order to evaluate the effectiveness of our policy, we implemented a user-level process scheduler on Linux. It executes as a daemon. When a program is submitted, the scheduler forks the process and inserts it to a global run queue. A PMU sampling context is also created and attached to each program. The IABW of current workload is also calculated. The scheduler set a timer for counting the scheduling time quantum. When the timer expires, it is notified by a Linux signal. The scheduling routine is accomplished in the signal handler follow these steps:

Step1: Block all the running processes by ptrace_attach, so that these processes will not be scheduled by Linux kernel.

Step2: For each program executed at last quantum, read and process its PMU data. Predict the bandwidth requirement at next quantum for each scheduling candidate.

Step3: Select proper jobs to run. Firstly, the processes executed at last quantum are moved to the tail of the run queue, and the job on the head of run queue is always selected. This step avoids starvation of any process. Afterwards, select other processes according to their bandwidth requirement, as we introduced in Section 3.1. In our policy, BW_{remain} is initialized to the estimated optimal scheduling bandwidth target according to the IABW of the workload.

Step4: unblock the selected processes using ptrace_detach. Reset the timer and let the scheduler itself sleep.

4.2 Bandwidth Requirement Prediction

Most existing scheduling framework use the observed bandwidth, or the average of the observed bandwidth of the last few time quanta, as the predicted bandwidth requirement for the next quantum [4][5][15]. However, there are two shortcomings with this method.

The first is that the observed bandwidth of a job by PMU does not always represent the real required bandwidth of the job. The online measured BTR could be much lower than its real bandwidth requirement if bus saturation has occurred. In our scheme, we calculate system-wide BTR using PMU sampled data. We set a bus saturation threshold T_B which is slightly smaller than the realistic peak bandwidth. If the system-wide BTR is smaller than T_B, it would indicate that the bus was not saturated, and the observed bandwidth can be used to estimate the requirement in next quantum. Otherwise, the information is discarded. Note that, even before bus is saturated, bandwidth contention could still exist.

The second shortcoming is that, the bandwidth requirement of some applications changes significantly in adjacent quanta, e.g. 459.GemsFDTD and 481.wrf. Using observed bandwidth for the following time quantum would yield very low prediction accuracy. Many previous studies show that program execution usually has some phase behavior [17][7]. We try to classify the scheduling quanta into phases. We sample a job's instruction pointer (IP) at runtime using PMU, and the sampling interval is much smaller than the scheduling quantum. We evenly divide the executable binary code into R code regions, and build an IP vector (IPV) whose length is also R. When a sampled IP falls into a code region, the corresponding element of IPV is incremented by one, then the IPV is normalized. If the Manhattan distance of two IPVs is smaller than a similarity threshold T_S, the corresponding program segments are in the same phase. We maintain a bandwidth requirement info for each detected phase. At last, we use a simple Markov predictor to predict the phase change and the corresponding bandwidth. We regard each phase as a state, and form a transfer matrix based on the phase history. In the beginning, if the predictor fails to predict the next phase due to the lack of necessary history, a last-value predictor is used.

5. EVALUATION

5.1 Workloads and Evaluation Metrics

We build some workloads using SPEC CPU 2006 floating-point benchmarks and single-threaded NPB benchmarks [1]. We evaluate the throughput by measuring the workloads' turnaround time. We take the average of three consecutive runs as our final result, and calculate their relative standard deviation (%RSD). We report the average %RSD of all workloads due to space limitation.

We use train input size for the SPEC benchmarks. The execution time of the SPEC benchmark varies widely from 13.6 seconds (436.cactusADM) to 116.0 seconds (435.gromacs). One of our experiments shows that a simple long-job-first policy could improve the turnaround time of some workloads by more than 25% because it could balance their workload among the processors. To give each benchmark the same weight, we execute all benchmarks alone for 2 minutes and record the number of times it executes. For the last un-finished execution, we record the number of instructions completed. When a benchmark is selected to add to a workload, the exact 2-minute worth of its execution is added to the workload. Such a scheme has two advantages.

Firstly, different from closed-systems like the one in [15], the workload composition is fixed. Hence, the measured throughputs under different policies are comparable. Secondly, it also allows our studies to focus only on the effects of the bandwidth-aware policy itself. As real applications could have varying lengths, we also present the results on *normal* workloads, i.e., the workload with only one instance of each selected benchmark.

The degree of multiprogramming is randomly set between 2 and 4, i.e. there would be 8 to 16 benchmarks in each workload for our 4-core experiments. The benchmarks are selected and mixed randomly. The workloads are generated offline and recorded for re-execution. Table 1 shows the workload by their IABW in an increasing order. Table 2 shows the scheduler parameters.

5.2 Determining Scheduling Bandwidth Target

As discussed in Section 3.3, the optimal scheduling bandwidth target should be set at a level lower than the IABW. We use the SPEC training workloads to approximate the relationship between the two metrics. As an example, Figure 7 shows the tuning process for WL#06. When we gradually decrease the scheduling bandwidth target from IABW, WL#06 achieved the shortest turnaround time when the target is set at 28 trans./usec, which is 4.1% better over the OS scheduler. It also shows that when the workload achieved the best performance, the distribution of BTR is concentrated in regions between 20 and 36. The quanta with BTR below 20 and above 36 are quite small. This result is consistent with our argument that maintaining uniformly distributed memory bandwidth utilization can benefit the

Table 1: IABW of Workloads

SPEC training Workloads	Index	#11	#12	#13	#05	#14
	IABW	23.2	25.0	27.1	28.1	29.2
	Index	#09	#15	#06	#03	#10
	IABW	31.3	31.9	33.3	38.0	43.7
	Index	#07	#04	#08	#01	#02
	IABW	45.2	45.3	49.4	58.1	62.9
SPEC Workloads	Index	#06	#08	#04	#02	#03
	IABW	28.5	31.0	32.2	35.0	42.0
	Index	#01	#09	#10	#07	#05
	IABW	42.3	42.4	48.2	48.5	53.0
NPB Workloads	Index	#02	#03	#09	#01	#08
	IABW	20.4	26.0	26.1	26.9	27.4
	Index	#05	#10	#06	#04	#07
	IABW	29.4	33.4	34.1	34.8	44.8

Table 2: Parameter Setup in the Scheduler

Parameters	Value
Max processor cores	4
Scheduling quantum	100 ms
PMU sample interval	1 million instructions
IPV similarity threshold T_S	0.4
Bus saturation threshold T_B	35 bus trans./usec

throughput of workloads. We will discuss the performance of other workloads under different scheduling targets in the next section.

The square points in Figure 8 show IABW (x-axis) and the corresponding scheduling targets (left y-axis) for all manually turned workloads. We use a polynomial regression method to quantify the relationship between the IABW and the scheduling bandwidth target, which is plotted as the solid line. The final curve-fitting function for the solid line is:

$$Sched.Target = -0.0118 \times IABW^2 + 1.4571 \times IABW - 6.2403 \tag{11}$$

We can see from the solid line that, as IABW increases, the difference between the IABW and the optimal scheduling bandwidth target also increases. This is because the difference comes from the bandwidth measured in a multi-programming environment and the bandwidth profiled when the benchmark runs alone, i.e. without any bandwidth contention. As IABW increases, bandwidth contention becomes

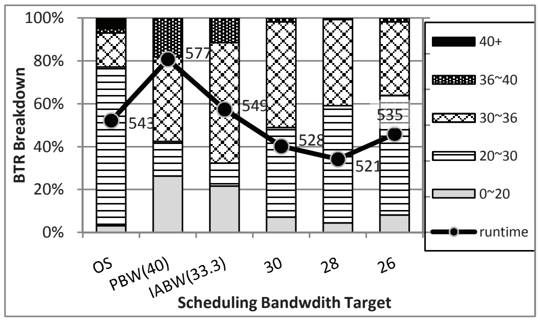


Figure 7: Tuning performance for WL#06 by varying bandwidth target.

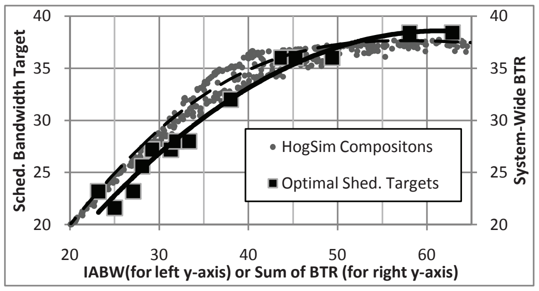


Figure 8: IABW-Sched.Target Appoximation.

more severe and their gap becomes larger. The solid line finally approaches the system peak bandwidth, which indicates that for memory-intensive workload, it is more important to increase the bandwidth utilization.

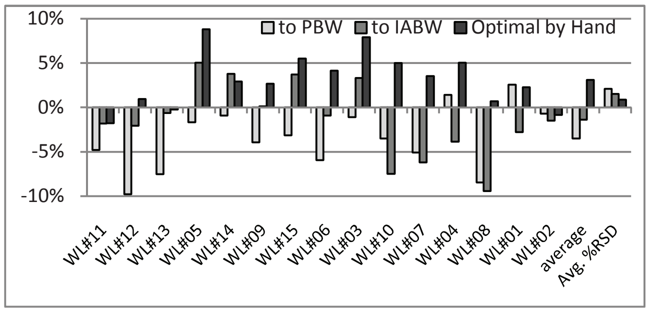
In Figure 6, we use *HogSim* to evaluate random job segments from all the SPEC 2006 floating point benchmarks running concurrently. It is a characterization of the relationship between the bandwidth requirement (measured when it ran alone) and the utilized bandwidth when it ran concurrently with other jobs. We embed Figure 6 in Figure 8 using the gray points (x-axis and the right y-axis) and the dashed line is the polynomial regression result of those 1k points. We can see that the solid line and the dashed line are similar in shape as expected. The solid line is on the right of the dashed line, because the evaluation result using *HogSim* is conservative, as we discussed in Section 2.3.

5.3 Study of Performance Tendency

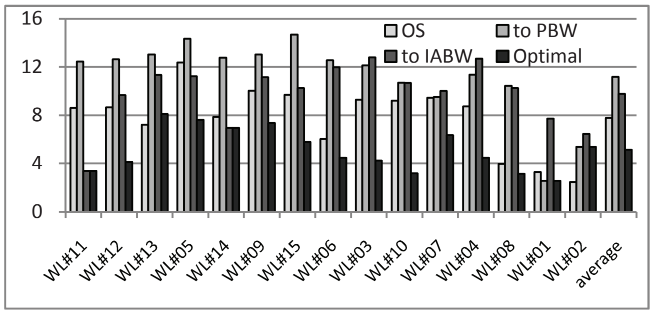
We first show the evaluation results of all SPEC training workloads. The three bars in Figure 9(a) show the improvement in the turnaround time over the native OS scheduler under different scheduling bandwidth targets: (1) set to peak system bandwidth (to PBW); (2) set to ideal average bandwidth (IABW); and (3) a manually selected optimal target (Optimal by Hand). In order to better understand the performance discrepancy, we record the system-wide BTR in each quantum for each workload, and compute the standard deviation. A lower standard deviation means that the bandwidth utilization is distributed more evenly among all quanta. The results are shown in Figure 9(b).

Initial observation shows that workload performance is correlated with the standard deviation of BTR. Under existing scheduling policies which always set the scheduling bandwidth target to PBW, some workloads suffer obvious slowdown, especially for those workloads with low IABW, such as WL #11, #12 and #13, where the slowdown could be up to 9.8%. As example, Figure 10 presents the BTR breakdown of WL#13. It shows that the OS scheduler may not cause severe bandwidth saturation, but setting scheduling target to peak bandwidth (PBW) could increase the proportion of both high-BTR (40+) and low-BTR (0-10).

The second bar in Figure 9(a) shows the performance when setting the scheduling bandwidth target to IABW. We have discussed in Section 3.3 that IABW is not the best practical scheduling bandwidth target, but we still include it because it could achieve a good bandwidth distribution. From Figure 9(b), we can see the standard deviations of IABW are usually smaller than PBW. For some workloads such as WL#14, it already increases the throughput over the



(a) Throughput improvement over native OS.



(b) Standard deviation of BTR in scheduling quanta.

Figure 9: SPEC training workloads: performance under different scheduling bandwidth targets.

native OS. While for the workloads with larger IABW, the performance is still decreased because the difference between the optimal scheduling bandwidth target and the IABW is larger.

At last, we show the performance achieved by adjusting the scheduling bandwidth target manually for each workload, as we did in Section 5.2. For all training workloads, up to 8.8% improvement could be achieved, and the average improvement is 3.1%. From Figure 9(b) we also see the BTRs are distributed most evenly.

5.4 Generality of the Policy

In this section, we use Equation 11 which was derived from the training workloads, and apply it on other workloads to derive their scheduling bandwidth targets. We want to see how general Equation 11 is to other workloads on our platform.

Figure 11 and Figure 12 show the evaluation results of the SPEC and NPB workloads respectively. Although using existing scheduling policy of setting the target to PBW, we could achieve up to 13.0% improvement for NPB WL#03,

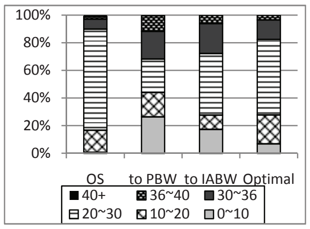


Figure 10: BTR Breakdown of WL#13.

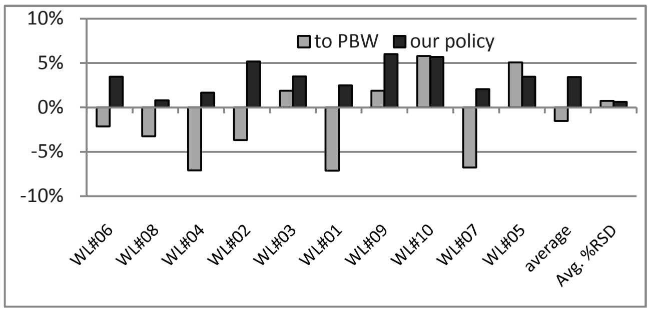


Figure 11: SPEC workloads: throughput improvement over native OS.

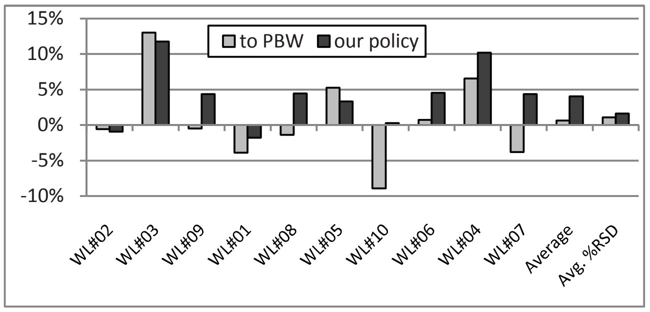


Figure 12: NPB workloads: throughput improvement over native OS.

most workloads would suffer throughput degradation. Compare to the native OS scheduler, scheduling to PBW degrades the throughput of SPEC workloads by 1.4% on average, and it achieves merely 1.1% better on the NPB workloads. While using our selected bandwidth target, up to 11.7% improvement could be achieved and the average improvement is 3.4% and 4.9% for SPEC and NPB workloads respectively. Due to the adaptability of the scheduling bandwidth target, the most severe slowdown suffered among all workloads is only 1.8%.

5.5 Effect on Normal Workloads

The above evaluation uses workloads that consist of benchmarks with the same length to eliminate the impact of the length of the job execution time. In this section, we present the results of more *normal* workloads. The composition of the benchmarks is the same as before, but only one instance of each benchmark is in the workload.

We also evaluate the performance under the long-job-first policy, in which the job with longer execution time has a higher priority to be scheduled. Under this policy, the processor fragmentation could be reduced, and the job-balancing among the processors could be improved. Figure 13 shows the evaluation results of the training workloads. Because the length of SPEC benchmarks varies widely, the average performance under the long-job-first policy turns out to be the best. Up to 25% improvement is achieved on WL#01. On average, the performance improvement is 6.1%. For the two bandwidth-aware policies, our policy did not degrade the performance of any workload, and achieved an average of 3.4% improvement, while setting the bandwidth target to peak bandwidth will degrade the average performance by 3.8%.

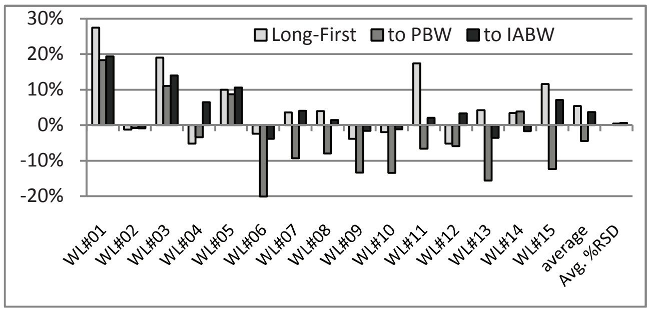


Figure 13: "Normal" SPEC workloads: throughput improvement over native OS.

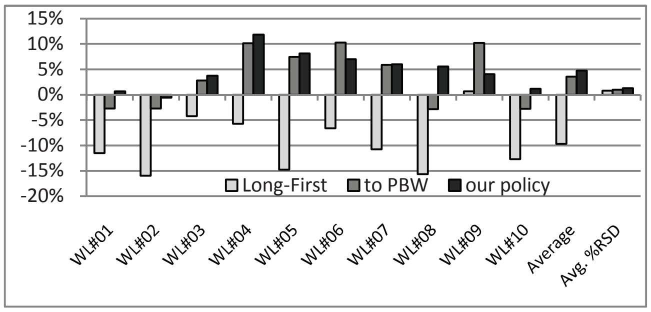


Figure 14: "Normal" NPB workloads: throughput improvement over native OS.

As shown in Figure 14, for most NPB workloads, long-job-first policy hurts the throughput. There are two reasons. Firstly, jobs with long execution time happen to have high BTR in NPB suite. For example, the first three jobs with the longest execution time happen to have two of the highest BTR (mg.C and cg.B). Hence, long-job-first policy would cause bus bandwidth to saturate when both of them are selected to run concurrently. Secondly, the variance of job lengths in NPB benchmarks is not as severe as that of the SPEC benchmark. Hence, long-job-first policy would not have much performance gain from improving the job-balancing among the processors. Using either PBW or our policy could improve the throughput. On average, the improvement of our policy is 4.75%, and the improvement of PBW is 3.56% for NPB workloads.

5.6 Bandwidth Requirement Prediction Accuracy

We evaluate each single benchmark using different BW prediction mechanisms, including: (1) our IPV-based phase detection and prediction scheme described in Section 4.2, with its similarity threshold set at 0.3 and 0.4; (2) the last-value prediction scheme; and (3) last-window prediction scheme, with the window size set at 2 and 3 quanta. In a time quantum, if the difference between the measured BTR and the predicted BTR is smaller than 5% of the realistic peak bandwidth, we consider it to be a successfully predicted quantum. Figure 15 shows the measured results. Our IPV-phase detection and prediction can improve the prediction accuracy for those 3 benchmarks (434.zeusmp, 459.GemsFDTD, 481.wrf) with most frequent BTR fluctuation.

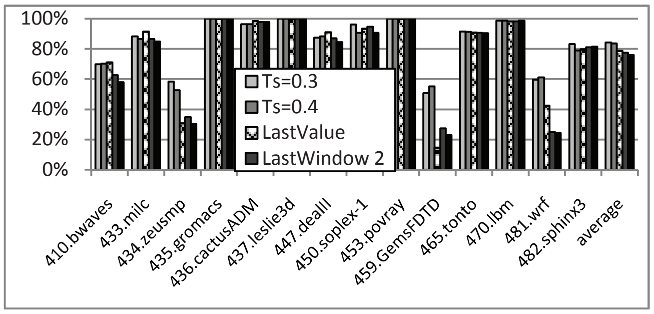


Figure 15: BTR prediction accuracy: percentage of successfully predicted quanta.

6. RELATED WORK

The bandwidth-aware job scheduling is widely investigated. Antonopoulos et al. proposed a bandwidth-aware job scheduling policy [4]. It selects the jobs based on the principle that total bandwidth requirement of co-scheduled jobs equals to the maximal capability of bus. Their future work [5] proposes a flexible and realistic design: they track the bandwidth consumption through the performance monitoring unit (PMU) to predict the jobs' bandwidth requirement in the near future, and the scheduling is based on the prediction. The bandwidth-aware scheduling policy is also applied as an important component together with other consideration on job schedulers. On the case of hybrid multiprocessors with hardware multi-threading support, the bandwidth-aware policy is applied to select jobs inter-processors [15]. Koukis et al. expanded the policy to SMP clusters [10], where the scheduler is conscious of both the memory bandwidth inside one SMP node and the network bandwidth among the nodes.

However, to the best of knowledge, all the existing scheduling frameworks try to keep the bus utilization to the peak bandwidth of the bus, but ignore the impact from the fine-granularity fluctuation. The most relevant work to ours is [12], in which showed that the "memory access pattern" also impact the bus utilization as bandwidth itself. They discussed an ideal solution that if the processor has large enough on-chip cache, and it is well scheduled, the memory accesses of programs can be forced into "burst" mode, i.e., a program would occupy 100% of the available bandwidth once it start to access the main memory, or issues no request. The ideal situation would simplify the design of bandwidth-aware policy. However, for real system, the size of L2 cache is limited. In addition, the working set of a program is usually hard to predict due to executive dependency. As a result, the "burst" mode seems not practical. In comparison, we don't try to alter the memory access behavior of a program but seek a scheduling solution which minimizes the impact of such fluctuation in memory access by carefully control the system-wide bandwidth utilization.

7. CONCLUSIONS

In this paper, we study how concurrent jobs perform when they share the limited bus bandwidth. We found that the bandwidth requirements of real applications fluctuate greatly when measured using very fine time intervals. Such fluctuations could cause bandwidth contention that could distort the parameters used in a job scheduling framework. We quantify the impact of bandwidth contention to the performance of concurrent jobs, and found that their performance degradation is super-linear as the bandwidth utilization approaches the system peak bandwidth. We give a theoretical argument that when the overall bandwidth requirement of co-scheduled jobs equals to the ideal average bandwidth of the workload, the throughput degradation due to bandwidth contention will be minimal.

We proposed a new bandwidth-aware scheduling policy which first estimates the ideal average bandwidth of workloads, and set proper scheduling bandwidth target accordingly. Our policy does not need any complicate modification to the traditional scheduler. The evaluation method isolates the impact of job-balancing among the processors. The measurements show that, our policy could achieve up to 11.7% throughput improvement over the native OS scheduler, and the average of the improvement is 4.1% for all random workloads. Our policy could automatically adapt to the change of the memory bandwidth utilized by the workload, so it performs well for workloads with various bandwidth requirements. In our future work, we plan to extend our scheduling policy to shared-cache CMP architectures. We also plan to apply our bandwidth analysis to manage the quality of service (QoS) through job scheduling.

8. ACKNOWLEDGEMENT

This research was supported in part by the National Natural Science Foundation of China (NSFC) under the grant 60736012, the Innovation Research Group of NSFC (60921002), the National Basic Research Program of China (2005CB321602), the National High Technology Research and Development Program of China (2007AA01Z110), the National Science and Technology Major Project of China (2009ZX01036-001-002), and by the U.S. National Science Foundation under the grant CNS-0834599, the Taiwan National Science Council under the grant NSC98-2220-E-001-001.

9. REFERENCES

[1] Nas parallel benchmarks. http://www.nas.nasa.gov/resources/software/npb.html.

[2] The perfmon2 website. http://perfmon2.sourceforge.net/.

[3] The sream benchmark website. http://www.streambench.org/.

[4] C. D. Antonopoulos, D. S. Nikolopoulos, and T. S. Papatheodorou. Scheduling algorithms with bus bandwidth considerations for smps. In *Proceedings of the 2003 International Conference on Parallel Processing (ICPP'03)*, page 547, Oct 2003.

[5] C. D. Antonopoulos, D. S. Nikolopoulos, and T. S. Papatheodorou. Realistic workload scheduling policies for taming the memory bandwidth bottleneck of smps. In *Proceedings of the 2004 IEEE/ACM International Conference on High Performance Computing (HiPC'04)*, pages 286–296, 2004.

[6] D. Burger, J. R. Goodman, and A. Kägi. Memory bandwidth limitations of future microprocessors. In *Proceedings of the 23rd annual international symposium on Computer architecture (ISCA'96)*, pages 78–89, New York, NY, USA, 1996. ACM.

[7] A. S. Dhodapkar and J. E. Smith. Comparing program phase detection techniques. In *Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture (MICRO'03)*, page 217, Washington, DC, USA, 2003. IEEE Computer Society.

[8] E. Ebrahimi, O. Mutlu, and Y. N. Patt. Techniques for bandwidth-efficient prefetching of linked data structures in hybrid prefetching systems. In *HPCA-15*, 2009.

[9] F. Guo, Y. Solihin, L. Zhao, and R. Iyer. A framework for providing quality of service in chip multi-processors. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'07)*, pages 343–355, Washington, DC, USA, 2007. IEEE Computer Society.

[10] E. Koukis and N. Koziris. Memory and network bandwidth aware scheduling of multiprogrammed workloads on clusters of smps. In *Proceedings of the 12th International Conference on Parallel and Distributed Systems (ICPADS'06)*, pages 345–354, Washington, DC, USA, 2006. IEEE Computer Society.

[11] A. Krste, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, W. L. Patterson, David A. andPlishker, J. Shalf, S. W. Williams, and K. A. Yelick. The landscape of parallel computing research: A view from berkeley. Technical Report UCB/EECS-2006-183, University of California, Berkeley, 2006.

[12] J. Liedtke, M. Völp, and K. Elphinstone. Preliminary thoughts on memory-bus scheduling. In *Proceedings of the 9th workshop on ACM SIGOPS European workshop*, pages 207–210, New York, NY, USA, 2000. ACM.

[13] J. Lin, Q. Lu, X. Ding, Z. Zhang, X. Zhang, and P. Sadayappan. Gaining insights into multicore cache partitioning: Bridging the gap between simulation and real systems. In *IEEE 14th International Symposium on High Performance Computer Architecture*, pages 367–378, 2008.

[14] N. R. Mahapatra and B. Venkatrao. The processor-memory bottleneck: problems and solutions. *Crossroads*, page 2.

[15] R. L. McGregor, C. D. Antonopoulos, and D. S. Nikolopoulos. Scheduling algorithms for effective thread pairing on hybrid multiprocessors. In *Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS'05)*, page 28.1, 2005.

[16] C. McNairy and R. Bhatia. Montecito: A dual-core, dual-thread itanium processor. *IEEE Micro*, 25:10–20, 2005.

[17] T. Sherwood, S. Sair, and B. Calder. Phase tracking and prediction. *SIGARCH Comput. Archit. News*, 31(2):336–349, 2003.

[18] D. K. Tam, R. Azimi, L. B. Soares, and M. Stumm. Rapidmrc: approximating l2 miss rate curves on commodity systems for online optimizations. In *Proceeding of the 14th international conference on Architectural support for programming languages and operating systems (ASPLOS'09)*, pages 121–132, 2009.

[19] J. Wang, S. Zhou, K. Ahmed, and W. Long. Lsbatch: A distributed load sharing batch system. Technical report, Computer Systems Research Institute, University of Toronto, 1993.