Adaptive SMT Control for More Responsive Web Applications

Hiroshi Inoue[†‡] and Toshio Nakatani[†]
[†] IBM Research – Tokyo, [‡] University of Tokyo
{inouehrs, nakatani}@jp.ibm.com

Abstract—**We developed a new task scheduling technique that improves the response times of Web applications. Most of the high-performance processors used in today's servers support thread-level parallelism with multiple hardware threads within each core using Simultaneous Multi-Threading (SMT). SMT capabilities can improve the peak throughput of a server by increasing the utilization of the computing resources in the processor. However, SMT may degrade the response time of Web applications if the server does not fully utilize its multicore CPUs, while SMT improves the response times regardless of the CPU utilization if the server is equipped with only one core. To precisely model this behavior, we created a new hierarchical queuing model to accurately predict the response times on multicore SMT processors, taking the CPU utilization and the number of CPU cores into account. Based on this model, we devised Adaptive SMT control, a technique to control the number of active SMT threads (SMT level) to minimize the response time. We measure the current CPU utilization and the number of cores, predict the response time for each SMT level and dynamically set the SMT level that yields the best response times for the Web applications. We show our technique successfully improves the response time of Web applications written in PHP, Ruby, and Java by up to 12.9% on Xeon, which employs 2-way SMT, and 12.4% on POWER7, which employs 4-way SMT, when the CPUs are not fully utilized. It is known that the CPU utilization levels are typically low in many commercial servers. Hence our new technique can improve the response times of many Web applications, thus improving the users' experience.**

I. Introduction

More and more server workloads are becoming Web-based on both the Internet and intranets. Such Web applications include not only relatively simple applications such as content management systems or cloud-based mail services, but also more complex enterprise applications such as online analytics processing (OLAP). The performance of Web application servers tends to be measured by peak throughput while holding the response times reasonably low so as to minimize the number of servers required to handle the incoming requests. However, improvements in the response times greatly affect the user experience with any interactive Web application [1]. Hence the response time is another important metric for the server performance.

In this paper, we focus on improving the response times of Web application servers. We introduce a new task scheduling technique that minimizes the response time at low CPU utilization. It is known that the CPU utilization levels are typically low in many commercial servers because the server capacity is often determined based on the peak load, which is much higher than the average load. For example, in 2013 Google reported that servers in their clusters spend most of their time within the 10% to 50% CPU utilization range [2]. This was unchanged from the data published in 2009. Also, Delimitrou and Kozyrakis also reported the CPU utilization was less than 50% at servers in a production cluster at Twitter [3]. Hence, focusing on relatively low CPU utilization conditions is reasonable for many real-world servers.

TABLE I. Summary of how SMT affects the response time

CPU utilization	on many cores	on 1 core
Low	<u>degrade</u> (common case in today's servers)	improve
High	improve	improve

To minimize the response time of the Web applications, this paper focuses on processors with SMT (Simultaneous Multi-Threading) capabilities [4, 5], which allow multiple hardware threads (*SMT threads*) to run on each CPU core. Many of today's high-performance processors used in servers do support thread-level parallelism with multiple cores and multiple SMT threads within each core. The SMT capabilities can improve the peak throughput of the server by increasing the utilization of the computing resources in the processors. Since the SMT typically gives higher peak throughput, most of the servers enable the SMT by default.

We first show how the SMT may degrade the response time of Web applications when a server equipped with multiple cores does not fully utilize its CPUs, while the SMT improves the peak throughput of the same Web applications. We studied two processors, Intel Xeon with 2-way SMT and IBM POWER7 with 4-way SMT. We also found that SMT always improves the response time when the server has only one core, irrespective of the CPU utilization, as summarized in Table I.

To predict this behavior precisely based on the CPU utilization and the number of CPU cores available in the system, we develop a queuing model that models the change in the single-thread performance due to SMT. We also model the task migration behavior of the OS task scheduler on multicore SMT processors, aggressively balancing the load among the SMT threads in a core but much less aggressively among different cores.

Our new model allows us to adaptively select the best SMT level (the number of active SMT threads in each core) based on the CPU utilization and the number of available cores. We can decide on the SMT level that yields the best response time by predicting the response time for each SMT level.

We implemented and evaluated our technique on Linux using two different processors, Xeon and POWER7. For the evaluation, we used Web applications written in PHP, Ruby,

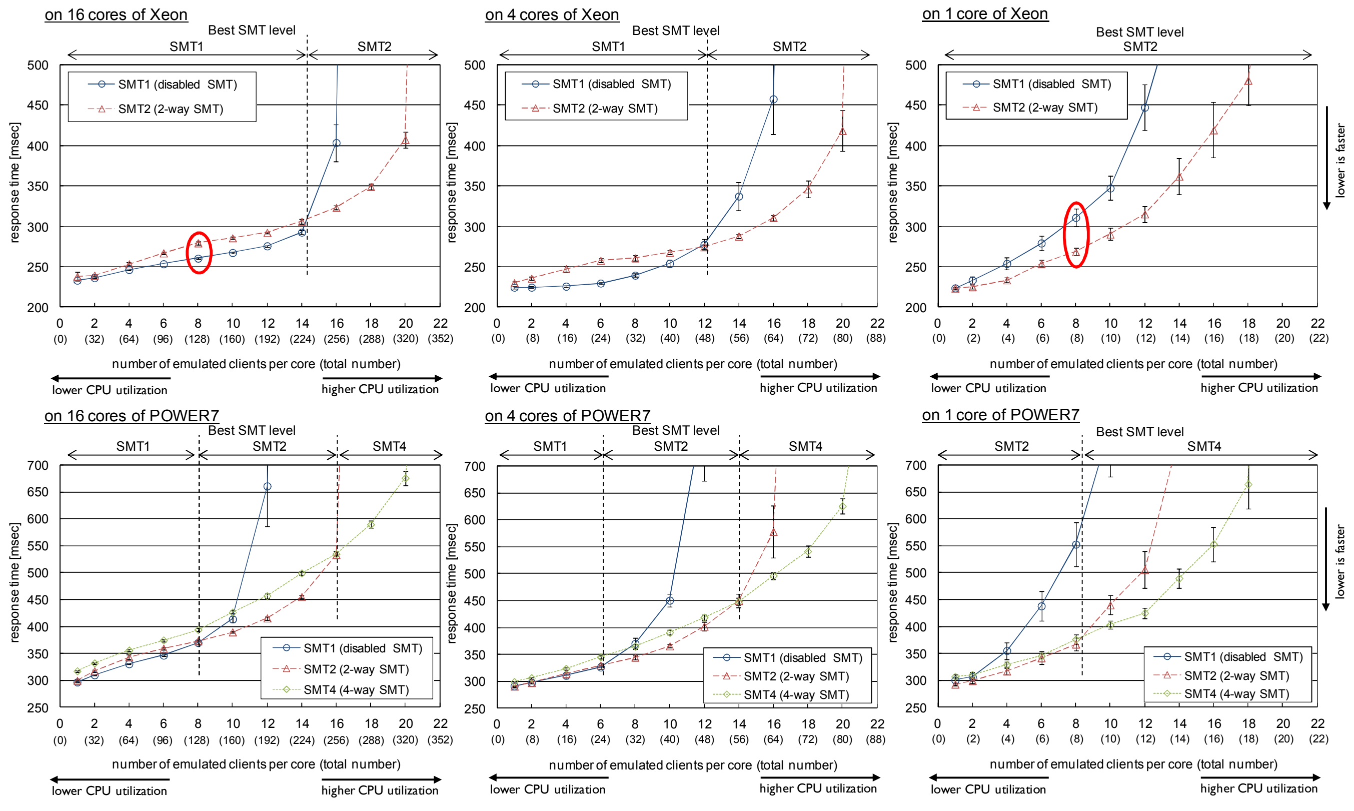

Figure 1. Average response time of MediaWiki with different SMT levels on Xeon and POWER7 using various numbers of cores. Error bars show 95% confidence intervals. The y-axis has a non-zero origin. The circles show the data points that appear in more detail in Figure 2.

or Java. Our technique selected the most appropriate SMT level and reduced the response time by up to 12.9% on Xeon and 12.4% on POWER7 compared to a default configuration using the maximum SMT level supported by each processor. When the server load increases, our technique automatically enables all the SMT threads and hence it can still fully leverage the benefit of the SMT in improving the peak throughput.

There are three main contributions in this paper. (1) We demonstrate that SMT may degrade the response time of the Web application when the server is equipped with multiple cores and these cores are not fully utilized. Although previous studies (such as [6]) have shown that the SMT may degrade the throughputs of some applications, as far as we know, this is the first detailed study on how the SMT may hurt the response times of Web applications. (2) We developed a hierarchical queuing model that can accurately predict the response times on multicore SMT processors. (3) We show that our adaptive SMT control technique based on the model can reduce the response times of the Web applications. Although we focus on Web application servers in this paper, our technique is not specialized for Web applications and hence other types of interactive workloads may also benefit from using it.

The rest of the paper is organized as follows. Section II shows how SMT affects the response times of Web applications on servers with multicore SMT processors. Section III presents our new model to predict the response times on multicore SMT processors. Section IV describes our technique to adaptively control the SMT level. Section V describes our experiments with Web applications written in PHP, Ruby, and Java. Section VI covers related work. Section VII discusses possible future work. Finally, Section VIII summarizes this paper.

II. How SMT Affects the Response Times

In this section, we describe how the SMT capability of the processor in the application server affects the response times and hence the users' experience with the Web applications, using a PHP application as an example.

A. *System Setup*

To study the effects of the number of active SMT threads (*SMT level*) on the response times of Web applications, we choose two systems, one based on Intel Xeon and the other on IBM POWER7. The first system has two 2.9-GHz Xeon E5-2690 (SandyBridge-EP) processors with 96 GB of system memory, running Red Hat Enterprise Linux 6.4 (kernel 2.6.32-358.2.1.el6) as its OS. The POWER7 system we used is equipped with two 3.55-GHz POWER7 processors with 128 GB of memory and runs the same version of Red Hat Enterprise Linux. Both systems have 16 processor cores and hence the total number of SMT threads (logical CPUs) is 32 for Xeon using 2-way SMT and 64 for POWER7 using 4-way SMT. We disabled the dynamic frequency scaling on both systems for more consistent and reproducible results. To control the number of cores and SMT threads, we used the sysfs interface exported by the Linux kernel (/sys/devices/system/cpu/). We set the same SMT level for all of the cores used.

As a target Web application, we selected MediaWiki-1.21.1. We ran MediaWiki on PHP-5.3.3 and Apache-2.2.15, as included in the Red Hat distribution. MediaWiki is a wiki server program developed by the WikiMedia foundation for the Wikipedia online encyclopedia. We used 1,000 articles from a Wikipedia database dump. We configured MediaWiki to use memcached-1.4.4 running on the same machine to cache the results of the database queries. The database server and the client emulator ran on separate machines. To measure the performance, the clients opened randomly selected articles. Between each request, the emulated client introduced a think time defined by an exponential random value with an average of three seconds. The number of emulated clients was changed to control the total workload. The response time is measured in the emulated clients.

B. *Performance Results*

Figure 1 shows how the processor's SMT level affects the response time of MediaWiki with an increasing number of emulated clients, increasing the CPU utilization in each server, on 16 cores, 4 cores, and 1 core of Xeon and POWER7. In each of the cases shown here, enabling SMT yielded a higher throughput and hence better response time with higher CPU utilization (right side of the figures).

When the CPU utilization was low (left side of the figures), the throughput was not affected by the SMT level because the server was able to respond to all requests from the clients and hence the throughput was determined solely by the incoming request rate. With low CPU utilization, we discovered that the response times were better with SMT1 (SMT disabled) when using 16 cores on both platforms. Also, on POWER7, SMT2 (2-way SMT) yielded shorter response times than SMT4 (4-way SMT) if the number of emulated clients was less than 256. In contrast, SMT improved the response times for any CPU utilization if the server had only 1 core. On one core of POWER7, SMT2 yielded the best response time at low CPU utilization and SMT4 gave the best at high CPU utilization. Because there is a small performance gap among SMT1, SMT2, and SMT4 even at the lowest CPU utilization, where the contention among the SMT threads was almost negligible, the POWER7 system appears to have constant performance overhead due to SMT4.

When using a small number of cores, such as the 4 cores shown in the figure, we observed that the response times were better with the lower SMT level for low CPU utilizations. However, the crossover point was at lower CPU utilization compared to the case with 16 cores. These behaviors are not unique to MediaWiki or PHP applications, but we found the same trends in the other Web applications (as shown later).

For deeper insight into the causes of the differences in the response times, Figure 2 shows histograms of the Xeon response times with a 5-msec width on 16 cores and 1 core. For both the 16-core and 1-core cases, the peaks of the histograms appear at a shorter response time for SMT1 compared to SMT2. In general, SMT improves the total throughput by allowing multiple threads to run in each core in exchange for some degradation in single-thread performance

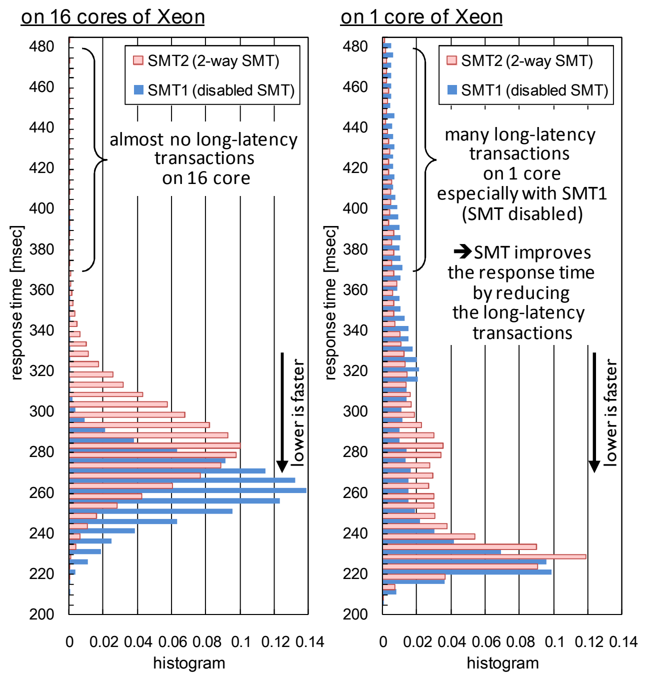

Figure 2. Histogram of the response time distribution of MediaWiki with different SMT configurations on Xeon using 8 emulated clients per core.

due to contention (in the execution units and also in the caches) among the multiple threads running on the same core. The lower single-thread performance makes the service times longer and shifts the peak of the histogram towards slightly slower response times for SMT2 compared to SMT1.

With one core, we observed a large number of transactions suffering from unreasonably long response times, especially with SMT1, and these long-latency transactions result in the worsened average and 90-percentile response times of the SMT1 when running on only one core. In the 16-core case, such long-latency transactions were not frequent enough to affect the average response times. The long-latency transactions occur when the server cannot serve a transaction immediately because all of the SMT threads are already executing other transactions. When the server has only one core, SMT increases the number of runnable threads and hence reduces the long-latency delayed transactions. When the server has multiple cores, there tend to be idling SMT threads in the system at any time that the average CPU utilization is low (about 25% for SMT1 in Figure 2). Although the OS randomly picks a server task (a process or a software thread) to serve the new incoming request, the OS task scheduler can quickly move the new server task to an idle SMT thread with little delay on a multicore system if the task cannot start immediately. Thus the increased number of HW threads with SMT does not improve the response time on multicore systems. As a result, on today's servers with lots of cores, the lower single-thread performance caused by SMT tends to degrade the response time of Web applications when the CPU utilization is low, while SMT improves the response time on only one core by avoiding the long-latency delayed transactions.

This observation shows that the response time can be

TABLE II. SUMMARY OF HOW SMT AFFECTS THE *RESPONSE TIME*, CONSISTING OF THE *SERVICE TIME AND* THE *WAITING TIME*

CPU utilization	on many cores	on 1 core	importance of waiting time
Low	response time = **service time** (+ waiting time) *degrade* ← ***degrade*** *negligible*	response time = service time + **waiting time** *improve* ← *degrade* ***improve***	↓ relatively less important
High	response time = service time + **waiting time** *improve* ← *degrade* ***improve***	response time = service time + **waiting time** *improve* ← *degrade* ***improve***	relatively more important
importance of waiting time	relatively less important (waiting time can be alleviated by migrating waiting tasks to other cores) →	relatively more important (not alleviated by migrating waiting tasks)	

broken down into two components, *service time* (CPU time) and *waiting time*. SMT degrades the service time due to lower single-thread performance, but it reduces the waiting time. However, the waiting time does not matter for the overall response time on multicore systems with low CPU utilization as summarized in Table II. Hence to make good predictions of the response times on multicore servers, we need to consider contention among the SMT threads to predict the service time and also the task migration behavior of the task scheduler to predict the waiting time.

III. HIERARCHICAL QUEUING MODEL

Based on the observations of Section II, we created a technique that uses queuing theory to predict the response time for each SMT level based on the measured CPU utilization and the number of cores available in the system.

A. Overview

To accurately model the complex behaviors of a system with multicore SMT processors using queuing theory, there are two challenges: (1) modeling the single-thread performance as affected by the resource contention among the SMT threads running on the same core and (2) modeling the task migration behavior of the OS task scheduler, which aggressively balances the load among the SMT threads within each core while minimizing migrations among the cores. Although existing queuing models such as the standard single queue and multiple server model (e.g. M/M/s queuing model) have been used to model systems with multiple processors, they are not suitable to predict the response times on multicore SMT processors because they do not take these challenges into account.

To precisely model the observed response times, we divide the response time in server T_r into service time (time receiving services in the CPU) T_s and waiting time in the OS task queue T_w as

$$T_r = T_s + T_w. \quad (1)$$

Here, T_r, T_s, and T_w depend on the SMT level t, the number of cores p and the current CPU utilization level. As we noted in Section II, the SMT typically degrades the single-thread performance and hence the service time T_s, while it improves the waiting time T_w.

In the new model, we first focus on a core with multiple SMT threads to calculate the single-thread performance and the waiting time, both affected by the contention among multiple SMT threads, without considering task migrations (*In-core modeling step*). We employ an iterative method to predict the average single-thread performance with SMT using the standard M/M/s queuing model. In each iteration, we calculate the waiting time and probabilities of the number of running threads within each core, refine the estimated single-thread performance as affected by SMT and iterate until it converges.

Then we update the waiting time T_w taking into account the effect of the task migration behavior (*Out-of-core modeling step*). We calculate the probability that at least one core in the system is idle. We expect that the task scheduler of the OS will migrate tasks from the waiting queue of a busy core to an idle core if one exists. Hence we update the waiting time calculated in the in-core step by multiplying by the expected probabilities of task migration. In this model, we assume that all of the p cores in the system use the same SMT level t.

B. Detailed Steps

1) *Normalizing CPU utilization*: To evaluate the CPU utilization without regard to the current SMT level, we calculate the normalized CPU utilization $\tilde{\rho}$ as the input to the model. This normalized CPU utilization is calculated as the CPU utilization if this machine processes the current workload without using SMT (i.e. SMT1). When the current SMT level is larger than one, the normalized CPU utilization may exceed 100%.

2) *Modeling the effect of SMT (In-core modeling step)*: To estimate the response time at SMT level t based on the queuing theory, we need the service time. However the service time is affected by the single-thread performance, which is unknown at this stage due to the nature of an SMT processor. Hence we first assume that average single-thread performance, *avgThreadPerf*, is not affected by the SMT. Because we are not trying to estimate the absolute performance, we can define *avgThreadPerf* as the relative single-thread performance over SMT1. Thus we first assume that *avgThreadPerf* = 1.0. Then we iteratively refine the average single-thread performance.

With the assumed average single-thread performance, we calculate the queuing model for each core without any effects from the other cores in the system using the standard M/M/s queuing model. Here we use the SMT level (the number of available SMT threads) t as the number of servers s in the model to solve the service time T_s and the utilization ρ as

$$T_s = 1.0 \,/\, avgThreadPerf, \quad (2)$$

$$\rho = \tilde{\rho} \,/\, (avgThreadPerf \times t). \quad (3)$$

We calculate the waiting time T_w within one core (p = 1), which is the delay before the start of execution, as

$$T_w\big|_{p=1} = \frac{t^{t-1}\cdot\rho^t\cdot T_s}{t!\left(1-\rho\right)^2}\pi_1[0]\,. \tag{4}$$

The probability $\pi_1[n]$ is for the n tasks in this core, which we calculate based on the standard M/M/s model as shown in (5). For π_1, the number of servers in the model s is the number of SMT threads within each core (the SMT level) t.

$$\pi[n] = \begin{cases} \left(\sum_{j=0}^{s-1}\frac{(s\cdot\rho)^n}{j!} + \frac{(s\cdot\rho)^t}{s!(1-\rho)}\right)^{-1} & (n=0) \\ \frac{(s\cdot\rho)^n}{n!}\pi_1[0] & (0 < n \le t) \\ \frac{(s\cdot\rho)^n}{s!\cdot t^{n-t}}\pi_1[0] & (t < n) \end{cases} \tag{5}$$

Then we refine the *avgThreadPerf* based on $\pi_1[n]$. To update the *avgThreadPerf,* we define *threadPerf*[n], the relative single-thread performance when n SMT threads are running simultaneously in a core relative to the performance without SMT. Hence, *threadPerf*[1] = 1.0 and 0 < *threadPerf*[n] ≤ 1.0, where 1 < n. For *threadPerf*[n], we can use the dynamically measured values for the current workload or statically defined typical values.

We update *avgThreadPerf* as the weighted average of *threadPerf*[n] with ($n \times \pi_1[n]$) as the weight and repeat the calculations for the queuing model with the updated *avgThreadPerf* until the waiting time converges or the number of iterations reaches a specified threshold (we used 10).

3) ***Modeling the task migration (Out-of-core modeling step)***: To model the effects of the task migration among the multiple cores in the system with p cores with SMT level t, we introduce another step with the standard M/M/s model. We calculate the probability that m customers (tasks) are in this system, $\pi_2[m]$, based on (5). This time, the number of servers s is the total number of SMT threads, $p \times t$. Then we obtain the probability that at least one core in the system is not executing any task, *migratableRatio*. By assuming that m tasks are randomly distributed among all cores, we can calculate *migratableRatio* as

$$migratableRatio = 1.0 - \left(\sum_{i=1}^{t} a[i]\right)^{p-1} \tag{6}$$

Here, p is the number of cores and $a[i]$ is the probability that i customers (tasks) are on a core. We calculate it by

$$a[i] = \begin{cases} \sum_{j=i}^{p\cdot t}\left(\pi_2[j]\cdot C(j,i)\cdot\frac{(p-1)^{j-i}}{p^j}\right) & (i < t) \\ 1.0 - \sum_{k=1}^{t-1} a[k] & (i = t) \end{cases} \tag{7}$$

For Equations (7), $C(n, k)$ shows a k-combination of n. When j tasks are randomly distributed among p cores, $C(j, i) \cdot (p-1)^{j-i} / p^j$ shows the probability that i tasks are on a specified core and $(j-i)$ tasks are on the other $(p-1)$ cores.

The OS task scheduler should migrate tasks in the waiting queue of a busy core to an idle core. Hence we update the waiting time T_w by multiplying (1.0 – *migratableRatio*) by the waiting time calculated in (4) where migration was ignored. We estimate the average waiting time T_w for SMT level t as

$$T_w = T_w\big|_{p=1} \times (1 - migratableRatio) \tag{8}$$

As we show in (1), we calculate the overall response time T_r for SMT level t as the sum of this waiting time T_w in (8) and the service time T_s in (2).

C. Evaluation

To evaluate the response-time predictions of our model, we calculated the response times for each SMT level. We used typical static values for *threadPerf.* On Xeon, we set *threadPerf*[n] = {1.0, 0.6}. This means that 2-way SMT gives a 20% improvement in the aggregate throughput (60% × 2 = 120%), while it degrades the single-thread performance by 40%. On POWER7, we used these values: *threadPerf*[n] = {1.0, 0.75, 0.57, 0.45}. This means that the relative aggregated throughputs for each SMT level are 1.0, 1.5, 1.7, and 1.8, respectively. We originally decided on these values of *threadPerf* as typical values for large Web applications written in Java, but they also worked well for the PHP and Ruby applications without specific tuning.

Figure 3 shows the predicted response time from our model on 16 cores, 4 cores, and 1 core of Xeon and POWER7. The x-axis shows the normalized CPU utilization as described in Section III-B. We show two lines for each configuration: one for the overall response time T_r and the other for the service time T_s. The gap between the two lines shows the waiting time in the queue T_w.

Comparing Figure 3 to Figure 1, our model successfully predicts the trends in the observed response times. On 16 cores, SMT1 yields the best response time at low CPU utilization (left side of the figures) for both processors and higher SMT levels gave a better response time when the CPU utilization increased. Other configurations were also consistent with the results in Figure 1.

Because our model simplifies the real systems, there are some minor differences between the measured and the predicted performances. For example, our model emphasizes the resource contentions among SMT threads in the same core and does not consider the contention among different cores, such as in the shared cache or the memory bus. Hence our model predicts the service time T_s as unchanged by the CPU utilization for SMT1, but this is somewhat unrealistic for most of the large applications. Another example in Figure 1 is that SMT4 on one core of POWER7 shows a slightly slower response time even at the lowest CPU utilization point and hence it appears to have constant overhead in the single-thread performance due to SMT as discussed in Section II. Our current model did not consider such overhead and so our model predicts SMT4 performs best on one core regardless of the CPU utilization, while SMT2 was best for low CPU utilization in Figure 1.

Comparing our model to the naive M/M/s model, where s is the total number of SMT threads, $p \times t$, the naive model strongly favored the higher SMT levels and simply predicts the maximum SMT level is best even on the 16-core configuration. This is because the naive model cannot predict

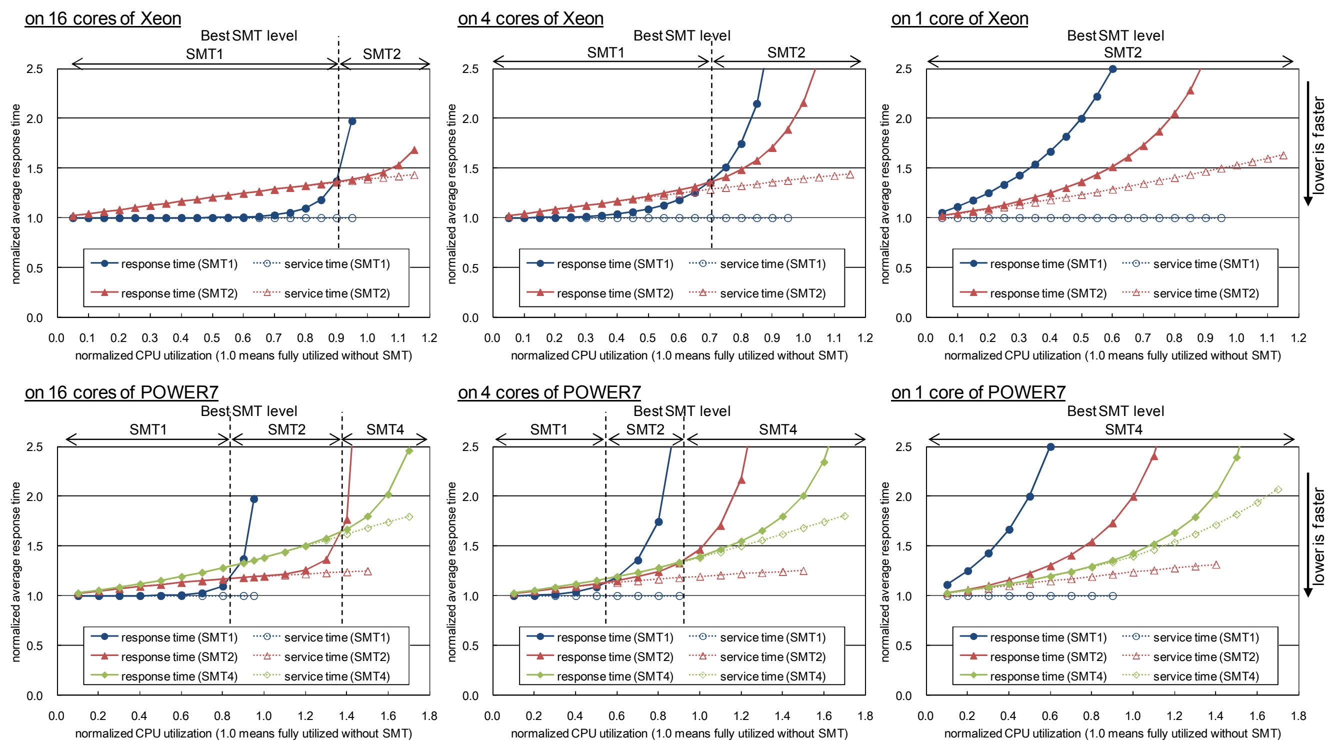

Figure 3. Predicted response time T_r and service time (CPU time) T_s with different SMT levels on Xeon and POWER7 using various number of cores. The gap between two lines for each configuration shows waiting time T_w, because the response time consists of the service time (CPU time) and the waiting time.

the degradation in the service time due to the contention among SMT threads and hence a higher SMT level always yields a better response time due to the reduced waiting time.

IV. Adaptive SMT Control based on Our Model

Based on our model, we designed an adaptive control technique for the SMT level to minimize the response times of Web applications. Our technique targets operating systems used in Web application servers.

A naive approach to identify the best SMT level is measuring the response time of the Web application for each SMT level and picking the SMT level that yields the best response time. However, it is not trivial to find the start and end times of a transaction to measure its response time in the OS task scheduler without interlocking with the running applications. In this paper, instead of relying on profiling, we calculate the response times for all of the SMT levels t ($1 \le t \le t_{max}$, t_{max} as the maximum SMT level supported by the processor, $t_{max} = 2$ for Xeon and $t_{max} = 4$ for POWER7) using our model and pick the best SMT level that minimizes the estimated response time. We assume that all of the cores in the system use the same SMT level t. We also assume that the number of cores is a constant during each run.

Instead of enhancing the task scheduler in the Linux kernel, we implemented a user-space daemon that controls the SMT level from user land. The daemon reads the current CPU utilization of each logical CPU from the proc file system (/proc/stat) once each 5 seconds. It uses the peak CPU utilization within the most recent 60 seconds (i.e. the last 12 measured intervals) as the input for the queuing model calculations to avoid overly frequent changes of the SMT level. The daemon calculates the best SMT level and sets the active CPU via sysfs (/sys/devices/system/cpu/). We use the same SMT level for all of the cores and do not control the SMT level for individual cores. On the 16-core Xeon system we used, the logical CPU x and x+16 are running on the same core. On the 16-core POWER7 system, the logical CPU 4x to 4x+3 are running on the same core. Because the daemon only runs every 5 seconds, the performance overhead caused by the model calculation was negligible on both platforms.

We obtain the normalized CPU utilization as the input to our model from the measured average CPU utilization of each SMT thread, $util[n]$, $1 \le n \le t_{max}$. On POWER7, because the task scheduler of the Linux kernel, the *completely fair scheduler* (*CFS*), uses the SMT thread with a smaller logical CPU id first within one core (*asymmetric SMT scheduling*), $util[n_1]$ is typically larger than $util[n_2]$ for $n_1 < n_2$. Hence, we expect that CPU time $util[n]$ - $util[n+1]$ will be spent while executing n SMT threads in a core. For example, when $util[1]$ = 80% and $util[2]$ = 50%, we expect that a core executes only one SMT thread in 30% (= $util[2]$ - $util[1]$) of the CPU time and two SMT threads in 50% (= $util[2]$) of the CPU time. The normalized CPU utilization is calculated as $\tilde{\rho}$ = ($threadPerf[1] \times 1$) $\times$ 30% + ($threadPerf[2] \times 2$) $\times$ 50%.

On Xeon, we assume that the CPU utilizations of the SMT threads are uncorrelated, because there is no asymmetric SMT scheduling. For the above example, we would expect that a core executes two SMT threads in 40% (= $util[1] \times util[2]$) of the CPU time and executes one SMT thread in 50% (= $util[1]$ + $util[2] - 2 \times util[1] \times util[2]$) of the CPU time. Hence, the normalized CPU utilization is calculated as $\tilde{\rho}$ = ($threadPerf[1] \times 1$) $\times$ 50% + ($threadPerf[2] \times 2$) $\times$ 40%.

TABLE III. WORKLOADS USED IN OUR MEASUREMENTS

Workload	Language	Descriptions of the workload		Avg. Response time	Transaction mix
MediaWiki	PHP	A wiki server program	open a wiki article	230 msec	1 type
			search the wiki for a word	185 msec	1 type
Ruby on Rails	Ruby	A Web application framework	list all entries and open one entry	95 msec	2 types
Cognos BI	Java + native (C++)	An interactive OLAP and business intelligence application	OLAP with balanced workload	210 msec	2 types
			OLAP to emphasize Java part	410 msec	2 types
			OLAP to emphasize native part	220 msec	7 types

- Transaction mix shows the number of different types of transactions involved in the scenario.
- Response times are measured on Xeon with a lightly loaded condition and averaged the response times for all types of transactions

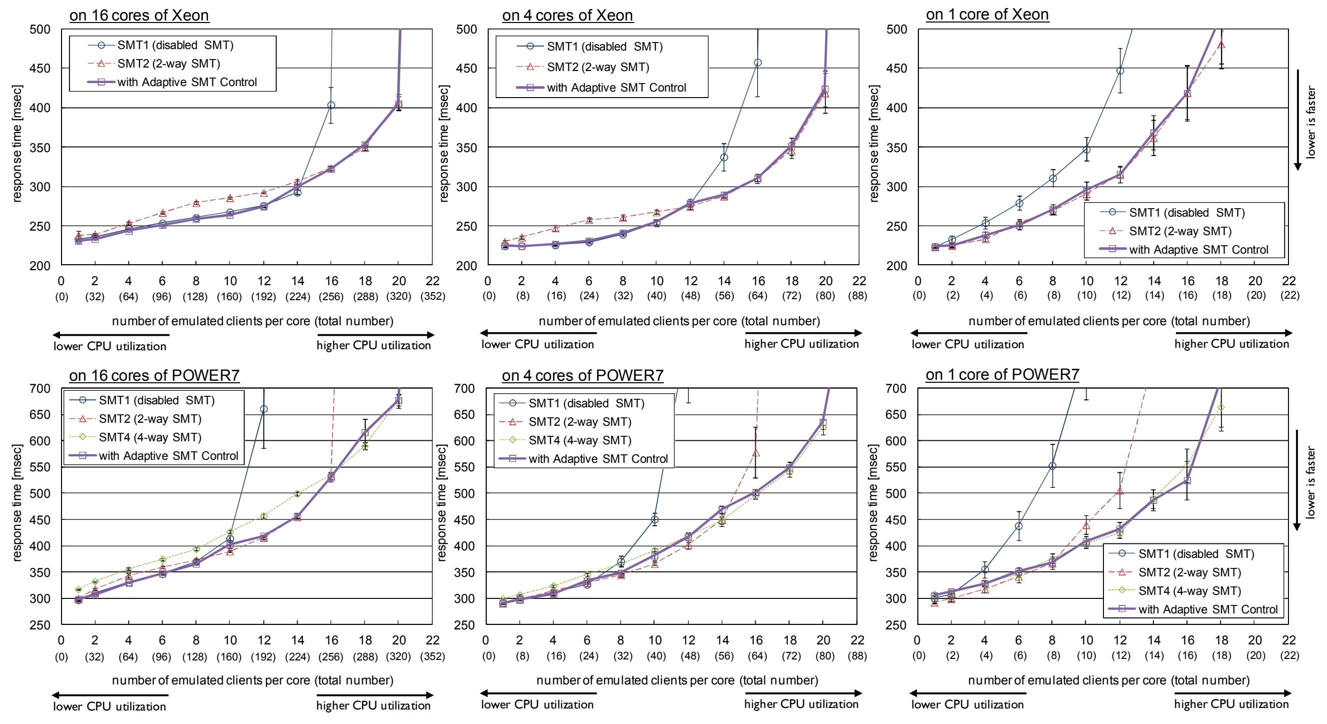

Figure 4. Average response time of MediaWiki with our adaptive SMT control on Xeon and POWER7 using various numbers of cores. Error bars show 95% confidence intervals. The y-axis has a non-zero origin.

V. Performance Results

We evaluated the performance benefits of our adaptive SMT control technique using Web applications written in PHP, Ruby, or Java on the two systems with 16 cores of Xeon and POWER7 described in Section II. As a PHP application, we used MediaWiki from Section II. We measured the response times of MediaWiki using two different scenarios. One is reading content as in Section II and the other is searching for a randomly selected word. For the Ruby workload, we selected Ruby on Rails-3.2.14 as a workload for the evaluation. Ruby on Rails is a framework for developing Web applications. We ran Ruby on Rails on ruby-1.8.7 included in the OS distribution. To drive the workload, lighttpd-1.4.13 is used as the Web server. It communicates with the Ruby runtimes using the FastCGI protocol. We built a simple telephone-directory application on top of the framework to focus on the performance of the framework. We generated 100 records in the database and measured the performance for the following scenario: reading a table of all of the records, selecting one record randomly, and opening that record. For this workload, the emulated client added think times defined by an exponential random value with an average of one second between each request. For the Java application, we used Cognos Business Intelligence (BI) v10.2, which analyzes and visualizes business data stored in a database. Cognos BI is a commercial application implemented using both Java and C++. We ran the Cognos BI on IBM WebSphere Application Server v8.5 as the Java application server. We used three different scenarios to drive the tests of the Cognos BI. One balanced the CPU utilization in Java and native code (*scenario 1*) and the others emphasized the performance of the Java part (*scenario 2*) or the native part (*scenario 3*). For all of the tested workloads, the database server and the client emulator ran on separate machines. Table III summarizes these workloads.

Figure 4 illustrates how our adaptive SMT control worked for MediaWiki on 16 cores, 4 cores, and 1 core of Xeon and on POWER7. In most cases, our technique successfully selected the best SMT level and improved the response time of the Web application compared to the default case using all

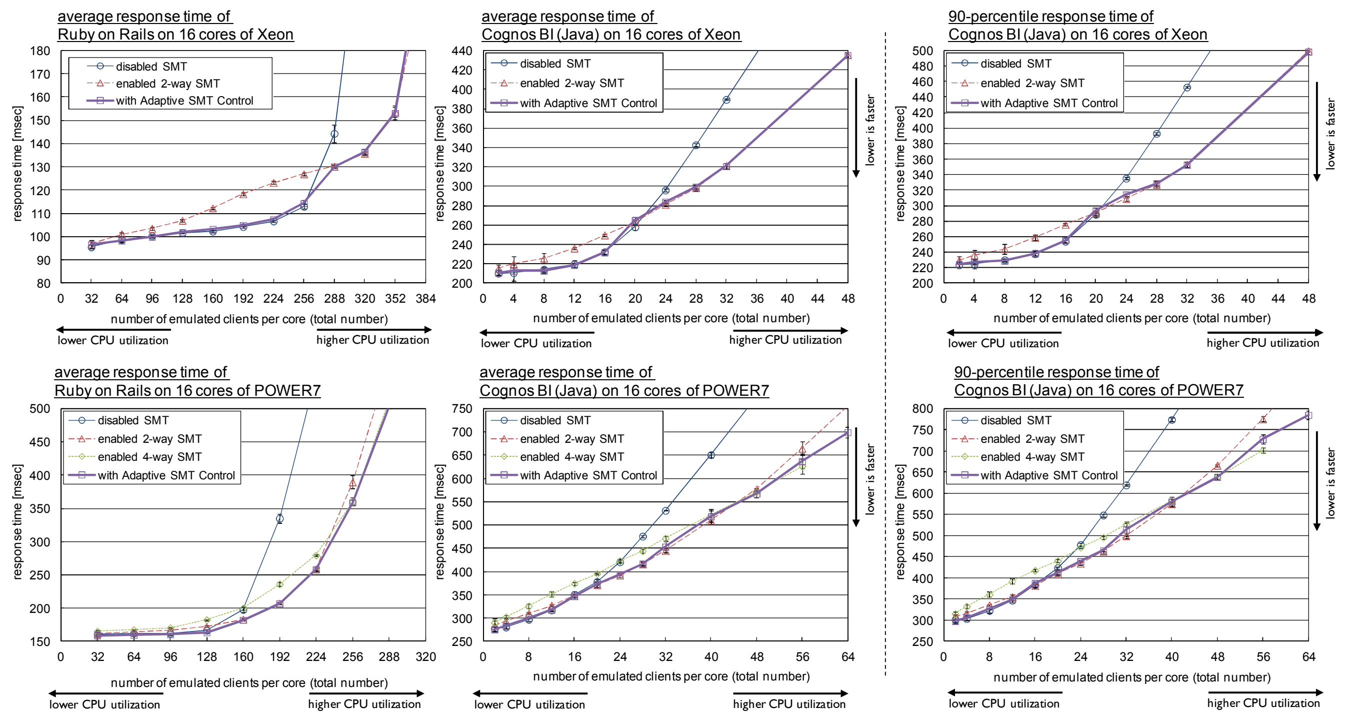

Figure 5. Average and 90-percentile response times of Cognos BI (scenario 1) and Ruby on Rails with our adaptive SMT control on 16 cores of Xeon and POWER7. The error bars show the 95% confidence intervals.

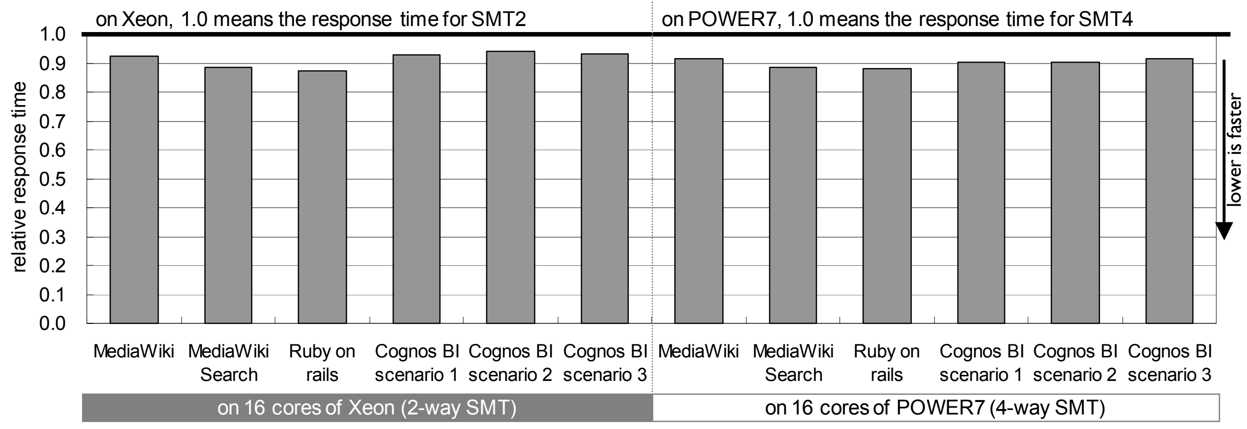

Figure 6. Maximum improvements in the average response times from our adaptive SMT control compared to the default SMT level (SMT2 for Xeon and SMT4 for POWER7) for all of the tested workloads on 16 cores of Xeon and POWER7.

of the SMT threads (SMT2 on Xeon and SMT4 on POWER7).

When using 16 cores, our model correctly predicted that the lower SMT level would yield better response times at low levels of CPU utilization. At higher CPU utilizations, our technique enabled all of the SMT threads provided by the hardware and achieved the same peak throughputs. This means that our technique takes advantage of both the higher throughput of the high SMT level and also the better response time of the low SMT level.

When running on only 1 core, our technique enabled all the SMT threads regardless of the current CPU utilization, as expected. On one core of POWER7, our technique selected SMT4 even though SMT2 achieved the best response time. This is because our model does not consider that the single-thread performance of SMT4 can be slower even without contentions among the SMT threads.

When comparing the results from 1 core to 16 cores, we have more optimization opportunities with larger numbers of cores. As illustrated in Figure 4, the range of the CPU utilization where our technique can give the improvements was smaller when using a smaller number of cores. Hence our technique is especially efficient on larger servers.

To show that our technique is not specialized for the specific MediaWiki application or for PHP applications, Figure 5 shows how our technique worked with other applications written in Ruby (Ruby on Rails) and Java (Cognos BI scenario 1). We show the average response time on 16 cores of Xeon and POWER7. The results were very similar to what we show for MediaWiki in Figure 4. Our technique selected the best SMT levels regardless of the CPU utilization for both applications.

Figure 5 also shows the 90-percentile response times for the Java workload on both platforms using 16 cores. Our technique improved the 90-percentile response times as well as the average response times.

Figure 6 shows the maximum improvements in the response times from our adaptive SMT control compared to the response time with the default SMT levels (SMT2 for Xeon and SMT4 for POWER7) for all of the workloads on both Xeon and POWER7 using 16 cores. We observed improvements for all of the tested workloads, with improvements ranging from 6.0% (Cognos BI scenario 2 on Xeon) to 12.9% (Ruby on Rails on Xeon). We did not observed significant differences in the improvements in the response times for Xeon and POWER7, even though they are completely different implementations and support different numbers of SMT threads per core.

VI. Related Work

In this paper, we focused on how we improved the response times of Web applications by improving the task scheduling on the multicore SMT processors. The performance of server-side software tends to be measured by peak throughput as long as the response times remain reasonably low. However additional (sub-second) improvements in the response times of Web applications are essential for better user experiences and matters for Web services [1]. Therefore, improving the general response times of Web servers deserves more study.

To improve the single-thread performance and response time at low CPU utilization, recent versions of the AIX Operating System use a scheduling policy called *raw throughput* mode [7]. This does not use the second or higher SMT threads until the primary SMT threads of all of the cores in the system are loaded to at least 50% utilization. This policy has the same goal as our technique, but is based on a static and predefined threshold. Our technique uses a queuing model to adaptively decide on the best SMT level for the number of cores available in the system and considering the current CPU utilization. As already discussed in this paper, the effects of the SMT level on the response time of Web applications heavily depends on the number of the CPU cores. Hence we can use our model to find the best threshold for the raw throughput mode for each CPU count in the system and the maximum SMT level supported by the processor.

Previous work on task scheduling for SMT processors focused on such aspects as achieving higher throughputs [6, 8-13] or satisfying real-time constraints on the SMT processors [14-19]. Identifying the best tasks to co-schedule on the SMT threads in one core (*symbiotic job scheduling*) has been an important topic for task scheduling on SMT processors. Typically, a symbiotic task scheduler first profiles the characteristics of each task running in the system by using existing hardware performance monitor or custom hardware. Then it decides on the best tasks to schedule together to minimize the resource contention. In our work, we focused on controlling the SMT level and did not control the scheduling for each task. A Web application server dedicated to only one Web application typically does not have enough variety in its microarchitectural characteristics to find opportunities for symbiotic job scheduling, because each task in the machine independently executes the same program for different users. For servers that simultaneously host a variety of different Web applications, we can integrate the ideas of task-sensitive symbiotic scheduling into our technique by finding the best task to co-schedule after our model selects the best SMT level.

Funston *et al.* [6] pointed out that the SMT-based improvements in the throughputs depend on the benchmark and that SMT sometimes degraded the throughput for a variety of compute-intensive programs. They proposed a metric to identify the best SMT level to achieve the highest throughput based on the performance events tracked with the hardware performance monitor of the processors using POWER7 and core i7. They showed that their new metric successfully predicted the benefits from SMT and this model can help the task scheduler to decide on the best SMT level and pick tasks to co-schedule. We also developed a model to decide on the best SMT level, but with a different focus, response time, not throughput.

Because SMT causes unpredictable single-thread performance, guaranteeing fairness and real-time constraints, such as the deadlines for task completion or the lowest guaranteed single-thread performance, on SMT processors is another challenging topic for task schedulers. For example, He and Hong [19] controlled the hardware priority of each SMT thread, which determines the distribution of the shared resources among the threads, by using a hardware performance monitor to guarantee the lowest single-thread performance on POWER7.

VII. Future Work

Though our model correctly predicts the response times of the tested Web applications, our model still simplifies the actual behavior of the systems. For example, we use a first-come first-served discipline in the model similar to the standard queuing models and ignore the effects of time slicing among multiple tasks. To add the effects of such features should make the model more accurate. Also, testing more workloads, such as applications with frequent lock contentions or interactive workloads other than server-side Web applications, will help to identify important features for better models.

In the current implementation, we statically use typical values as the performance gain from SMT in *threadPerf* in our model. These values may depend on the current workload and hence using dynamically measured values may improve the model's accuracy in exchange for additional runtime overhead to measure the single-thread performance for each SMT level.

Our queuing model correctly predicts the best SMT level with the current Linux kernel task scheduler (CFS). Though our queuing model does not include features specific to CFS, the load balancing behaviors of task schedulers are very complicated [20]. Therefore testing our technique with other task scheduler implementations is an important future project.

Although we implemented our technique as a user-space daemon without modifying the Linux kernel in this paper, an alternative implementation approach could be done inside the OS task scheduler to reduce the overhead of setting the SMT level by individually enabling or disabling the logical CPUs via sysfs.

VIII. Summary

In this paper, we describe our technique to dynamically control the SMT level in Web application servers to minimize the response times of the applications. Most of the high-performance processors used in today's servers provide SMT capabilities that can improve the peak throughput of the server by increasing the utilization of the computing resources in the processor. However, we found that SMT may actually degrade the response times of our Web applications when the server is not fully utilizing its multicore CPUs, while SMT improves the response times regardless of the CPU utilization when the server uses only one core. Based on these observations, we introduce a new hierarchical queuing model that can accurately predict the response times on multicore SMT processors. We devised a new technique to adaptively control the number of active SMT threads using this new model to minimize the response times of Web applications. Our evaluations showed that the new technique improved the response times of Web applications written in PHP, Ruby, and Java by up to 12.9% on Xeon, which has 2-way SMT, and on POWER7, which has 4-way SMT, when the CPUs are not fully utilized. It is known that the CPU utilization levels are typically low in many commercial servers. Hence our new technique can improve the response times of many Web applications, improving the users' experience.

References

[1] Nicole Sullivan. *Design Fast Websites*. Slideshare. Oct 14, 2008. http://www.slideshare.net/stubbornella/designing-fast-websites-presentation

[2] Luiz André Barroso, Jimmy Clidaras, and Urs Hölzle. *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines, Second Edition.* 2013.

[3] Christina Delimitrou and Christos Kozyrakis. Quasar: resource-efficient and QoS-aware cluster management. In *Proceedings of the 19th international conference on Architectural support for programming languages and operating systems* pp. 127-144. 2014

[4] Dean M. Tullsen, Susan J. Eggers, Joel S. Emer, Henry M. Levy, Jack L. Lo, and Rebecca L. Stamm. Exploiting choice: Instruction fetch and issue on an implementable simultaneous multithreading processor. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture.* pp. 191-202. 1996.

[5] Dean M. Tullsen, Susan J. Eggers, and Henry M. Levy. Simultaneous multithreading: maximizing on-chip parallelism. In *25 Years of the International Symposia on Computer Architecture* (selected papers). pp. 533-544

[6] Justin R. Funston, Kaoutar El Maghraoui, Joefon Jann, Pratap Pattnaik, and Alexandra Fedorova. An SMT-Selection Metric to Improve Multithreaded Applications' Performance. In *Proceedings of the 2012 IEEE 26th International Parallel and Distributed Processing Symposium.* pp. 1388-1399. 2012.

[7] Steve Nasypany. Optimizing for POWER7 & AIX What's New, *Michigan IBM i and AIX Technical Education Conference.* 2013.

[8] Allan Snavely and Dean M. Tullsen. Symbiotic job scheduling for a simultaneous multithreaded processor. In *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems*. pp. 234-244. 2000.

[9] Sujay Parekh, Susan Eggers, Henry Levy, and Jack Lo. Thread-sensitive scheduling for SMT processors. *Technical report, Dept. of Computer Science & Engineering, Univ. of Washington.* 2000.

[10] Allan Snavely, Dean M. Tullsen, and Geoff Voelker. Symbiotic job scheduling with priorities for a simultaneous multithreading processor. *SIGMETRICS Perform. Eval. Rev.* 30(1). pp. 66-76. 2002.

[11] Alex Settle, Joshua Kihm, Andrew Janiszewski, and Dan Connors. Architectural Support for Enhanced SMT Job Scheduling. In *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques.* pp. 63-73. 2004.

[12] Stijn Eyerman and Lieven Eeckhout. Probabilistic job symbiosis modeling for SMT processor scheduling. In *Proceedings of Architectural Support for Programming Languages and Operating Systems.* pp. 91-102. 2010.

[13] Stijn Eyerman and Lieven Eeckhout. Probabilistic modeling for job symbiosis scheduling on SMT processors. *ACM Trans. Archit. Code Optim.* 9(2), Article 7. 2012.

[14] Kun Luo, Jayanth Gummaraju, and Manoj Franklin. Balancing thoughput and fairness in SMT processors. In *Proceedings of IEEE International Symposium on Performance Analysis of Systems and Software.* pp. 164-171. 2001.

[15] Rohit Jain, Christopher J. Hughes, and Sarita V. Adve. Soft Real-Time Scheduling on Simultaneous Multithreaded Processors. In *Proceedings of the 23rd IEEE Real-Time Systems Symposium.* pp. 134-145. 2002.

[16] Francisco J. Cazorla, Peter M. W. Knijnenburg, Rizos Sakellariou, Enrique Fernandez, Alex Ramirez, and Mateo Valero. Architectural support for real-time task scheduling in SMT processors. In *Proceedings of the 2005 International Conference on Compilers, Architectures and Synthesis for Embedded Systems.* pp. 166-176. 2005.

[17] Hiroshi Inoue, Takao Moriyama, Yasushi Negishi, and Moriyoshi Ohara. CPU Resource Reservation for Simultaneous Multi-Thread Systems. *IBM Research Report.* 2006.

[18] Carlos Boneti, Francisco J. Cazorla, Roberto Gioiosa, and Mateo Valero. Soft real-time scheduling on SMT processors with explicit resource allocation. In *Proceedings of the 21st International Conference on Architecture of Computing Systems.* pp. 173-187. 2008.

[19] Zhengyu He and Bo Hong. PMU-guided Priority Adjustment to Guarantee Thread Performance on IBM POWER SMT Processor. *IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum IPDPSW.* 2012.

[20] Joseph T. Meehean, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Miron Livny. Uncovering CPU load balancing policies with harmony. In *Proceedings of the ACM International Conference on Computing Frontiers.* Article 13. 2013.