RPO: Runtime Web Server Optimization Under Simultaneous Multithreading

Samira Musabbir, Diwakar Krishnamurthy
University of Calgary
Dept. of Electrical and Computer Eng.
Calgary, Canada
{smusabbi, dkrishna}@ucalgary.ca

Giuliano Casale
Imperial College London
Department of Computing
London, U.K.
g.casale@imperial.ac.uk

***Abstract*—Multicore architectures commonly feature simultaneous multithreading (SMT), a hardware technology to improve the performance of multi-threaded applications, such as web servers. By studying a TPC-W testbed we observe that the performance of SMT for a multi-tier application strongly depends on the workload mix in execution in the system, thus prompting the need for smart management policies to decide when to enable or disable SMT. To tackle this problem, we propose the Runtime SMT Performance Optimizer (RPO), a module for the Apache web server that automates SMT activation and deactivation at runtime. Decisions rely on the estimated mix of requests in execution in the system and a classification of transactions based on historical data about the ability of each request to benefit from SMT. Experimental results indicate that RPO can gain up to 40% in request latency compared to the best static SMT configuration policy, at the expense of a small overhead of 0.6% utilization on average for each core.**

I. Introduction

Simultaneous multithreading (SMT) is a popular hardware technology for multicore architectures, implemented in commercial solutions such as Intel Hyper-Threading [1]. SMT aims at increasing resource utilization by implementing on each core a set of hardware threads. These are processing units that share some hardware components (e.g., caches, register file, fetch bandwidth). At the operating system level, each hardware thread is abstracted as a logical core which provides the illusion to applications of running on a real physical core. In this way, SMT allows higher parallelism and better throughput for multithreaded code. However, this can come at the expense of higher latencies, due to the contention from other hardware threads or due to cache misses [11], [12].

Despite its diffusion and importance, past research on SMT has focused with few exceptions on architectural performance under micro-benchmarks [2]–[4]. Some studies have investigated the impact of SMT on enterprise servers [6] and databases [8], [9], however relatively little research has been done towards understanding the implications of SMT adoption from a web server management perspective. In this paper, we try to fill this gap by studying the performance implications of SMT on a basic TPC-W multi-tier application [15]. To the best of our knowledge, this is the first study that investigates web server *dynamic management* issues arising from SMT.

By running TPC-W workloads on a two-tier architecture, we find that the impact of SMT on web server response times can be dramatic, up to 50% on average on some experiments, either as a speedup or as a slowdown from the baseline depending on the workload. Root-cause analysis reveals SMT degradations to arise from a subtle chain of interaction originating from cache misses at the web tier and ending in excessive thread creation at the database. Rather than stressing the magnitude of such slowdowns, which can change depending on the workload and testbed, our work emphasizes the difficulty of deciding when to enable SMT and thus argues for the need of automated solutions. Further, we try to answer fundamental questions that arise on the achievable performance gains with dynamic SMT management.

To tackle this problem, we propose the Runtime SMT Performance Optimizer (RPO), an Apache web server module that automates SMT activation and deactivation at runtime. RPO stems from the assumption that the mix of requests in execution in a web server can drive the decision on enabling or disabling SMT. It then estimates the mix of requests in the system by intercepting the HTTP metadata upon request arrival to the web server and thus keeping track in a lightweight manner of the system state. We then propose a methodology to map a given mix into a SMT activation decision based on estimates from historical or test data of the ability of each request to benefit from SMT. Based on this, we introduce a classification for request mixes as being either SMT friendly or unfriendly. This classification is used by RPO at runtime to decide on SMT activation or deactivation.

Experimental results on workloads with different degrees of variability, obtained by mixing existing TPC-W workloads, reveal that RPO can gain up to 40% in request latency compared to the best static SMT configuration policy. Furthermore, it only imposes a small cost of 0.6% overhead per-core utilization. In particular, experiments show RPO to be robust to workload non-stationarity where the mix in the system changes rapidly.

The rest of the paper is organized as follows. Section 2 provides a motivating example for our work, showing the unpredictable behavior of SMT under different TPC-W workloads. Section 3 introduces RPO and discusses its implementation. Section 4 quantifies RPO performance and overheads for experiments under different workload mixes. Section 5 summarizes related work. Section 6 draws conclusions.

978-3-901882-50-0

II. Motivating Example

A. Testbed

The testbed used in this paper consists of a client machine and a server machine connected by a 1 Gbps Ethernet switch. The testbed runs a PHP implementation of the TPC-W benchmark multi-tier bookstore application [15]. Each machine has 32 GB of RAM and 2 quad-core Intel Xeon 5540 2.53GHz processor with 2 hardware threads per processor. Thus, the server has 8 physical cores and can use up to 8 cores without hyper-threading (NOHT configuration policy) or up to 16 cores when hyper-threading is enabled (HT configuration policy). We shall refer to the HT and NOHT configurations as SMT policies. L1 (64 KB) and L2 (256 KB) caches are private to each core, while the L3 cache (8MB) is shared by cores on the same processor.

The server machine runs Ubuntu Linux (kernel ver. 2.6.38), and hosts an Apache web server (ver. 2.2.16) and a MySQL database (ver. 5.0.75). Thus, we adopt a two-tier deployment. This represents a stress case for SMT, since it implies hardware resource sharing from different application tiers. The client and server systems are optimized for high concurrency. Apache is configured with MaxClients increased to 1300 and MySQL is configured with max_connections set to 2000.

In our experiments, the cores of processor 0 are used for processing requests; processor 1 runs only monitoring tools. Thus, if not otherwise specified, Apache and MySQL use exactly 4 cores (NOHT) or 8 cores (HT), all of processor 0. We disable cores on both processors 0 and processor 1 that are not needed for an experiment. Server-side measurements are obtained with *collectl* and with the Intel Performance Counter Monitor (PCM)(ver. 2.0). Power consumption is collected by an external Lindy Power Meter 32710. Reported utilizations only consider cores executing processes belonging to TPC-W and their OS activity.

B. Experimental Results

To illustrate the impact of SMT on a multi-tier architecture, we experiment with the *browsing* and *ordering* workload mixes of TPC-W. The *browsing* mix is read-intensive at the database. In contrast, *ordering* frequently updates the database tables. We use the open-source *httperf* workload generator to emulate clients; *httperf* generates open arrivals of sessions, thus mean throughput and mean request arrival rate are identical in our experiments and decided before starting the experiment so as to impose a target utilization. Session inter-arrival times are exponentially distributed. Successive requests within a session are separated by an exponentially distributed think time with a mean of 2 seconds. For each experiment, multiple runs are performed to obtain tight confidence intervals, at 95% confidence, for the mean response time. The duration of an experiment is 900 seconds for *browsing* and 1500 seconds for *ordering*.

Table I illustrates mean performance metrics for a set of experiments for different processor frequencies and SMT policies. The fastest response time in each group is marked in bold. Frequency is scaled using the *cpufreq* kernel infrastructure in Linux. The results indicate that the mean response time is slightly lower for NOHT at low utilizations

TABLE I. Results: browsing mix

Freq.	*Cores*	*Policy*	*X*	*R*	*U*	*P*
2534	8	HT	356	13.50	0.34	184
2534	4	NOHT	356	**11.60**	0.53	193
2133	8	HT	356	16.10	0.40	170
2133	4	NOHT	356	**14.50**	0.62	176
1867	8	HT	356	**17.50**	0.46	167
1867	4	NOHT	356	18.70	0.69	174
1600	8	HT	356	**21.20**	0.55	167
1600	4	NOHT	356	43.10	0.86	173

Frequency is in KHz. X= Mean Throughput (rps), R= Mean Response Time (ms), U= Mean Core Utilization, P= Mean Power consumption (W).

TABLE II. Results: ordering mix

Freq.	*Cores*	*Policy*	*X*	*R*	*U*	*P*
2534	8	HT	48	41.50	0.22	173
2534	4	NOHT	48	**38.70**	0.38	202
2133	8	HT	48	103.70	0.33	164
2133	4	NOHT	48	**50.00**	0.48	179
1867	8	HT	48	*unstable*	0.61	168
1867	4	NOHT	48	**58.40**	0.56	169
1600	8	HT	48	*unstable*	0.64	164
1600	4	NOHT	48	**156.80**	0.69	167

Frequency is in KHz. X= Mean Throughput (rps), R= Mean Response Time (ms), U= Mean Core Utilization, P= Mean Power consumption (W), *unstable* = thread limit exceeded ($R \in$[18s, 55s]).

(2534KHz and 2133KHz). However, HT clearly outperforms NOHT at higher utilizations (1867KHz and 1600KHz). For example, with 1600KHz and NOHT we have seen that there is a 20% probability that the response time will be greater than 70 ms. In contrast, with HT the chance of response times exceeding 70 ms is less than 5%. As expected, the power consumption decreases with frequency for both HT and NOHT. Furthermore, power consumption is smaller for HT than NOHT for all load levels. Summarizing, NOHT performs *marginally* better than HT at low loads; at high loads, HT *significantly* outperforms NOHT. From this, HT appears to be, overall, a better configuration for this server under *browsing*.

Table II presents evidence that this conclusion, however, depends on the workload mix under study. The table reports experimentation for the *ordering* mix. While power consumption is similar to what is observed for *browsing*, we observe that the performance behavior of HT is vastly different for this mix. At 2534KHz, the mean response time for HT is 7% higher than for NOHT. At 2133KHz, the gap grows to 52%, still in favor of NOHT. At 1866KHz and 1600KHz under HT, the system becomes unstable due to a severe backlog of requests which causes both the Apache and MySQL thread limits to be exceeded. In contrast, the system remains stable under NOHT. We have further investigated the root causes of this gap using specialized monitoring tools as discussed next.

C. Root Cause Analysis

To explain the behavior we have just discussed, we perform a root cause analysis using L3 cache data from the PCM tool. Since the L3 cache is shared between hardware threads, it is a major source of contention in SMT. In this campaign we pin all Apache processes to processor 0 while MySQL processes are pinned to processor 1. This setup allows us to discriminate

TABLE III. Database Profiling : Ordering

Mix	*Policy*	*R*	Threads	
			Max Running	Max Created
ordering	HT	179.70	44	185
ordering	NOHT	114.20	22	85

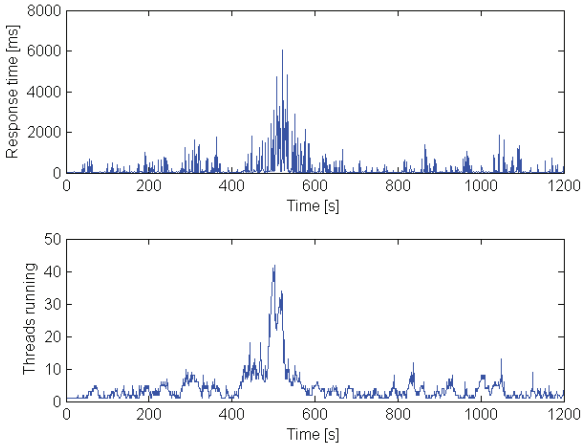

Fig. 1. Database Contention of ordering - Hyper-Threading: (a) Response Time (b) Number of Threads Running

the impact of HT and NOHT on web server and database and effectively emulates a multi-tier deployment. Table IV illustrate the results. We define the L3 misses per completion (MPC) metric as the ratio between the total number of L3 misses and the total number of requests completed in the run. Table entries are sorted for increasing response times.

Table IV indicates that *ordering* suffers much larger MPC values than *browsing*. We attribute this partly to the fact that *browsing* is a read-intensive workload, thus it can benefit from increased cache locality compared to *ordering*. We also note that the changes observed in the mean response times for *ordering* across the four experiments have a 98% correlation with the variation in web tier MPCs and a fair correlation (36%) with the database tier MPCs for the same experiments. Conversely, under *browsing* there is a negligible correlation (-5%) between response times and web tier MPCs as well as a fair correlation (33%) between response times and database tier MPCs. These results are consistent with the different behaviors seen for HT and NOHT in Section II-B: under HT *browsing* does not incur major MPC penalties and can fully take advantage of the increased parallelism of HT. Conversely, under *ordering* there are major performance degradations due to significantly increased cache misses at the web tier.

We have further investigated *ordering* to better understand the impact of cache misses at the web tier. In this analysis, we have used the "SHOW STATUS" MySQL command to collect performance measures at the database tier. Our conjecture is that, if transactions are slowed down at the web tier there will also be an increase in the number of concurrent transactions at the database tier. Indeed, Table III shows that this is the case: the slowdown at the Web tier causes severe congestion at the database tier as well. From the table, the maximum number of concurrent database threads running and the maximum number of database threads created is approximately double with HT than with NOHT. Figure 1 provides a more detailed view of the contention at the database server triggered by HT. The figure shows the time series plots of transaction response times, averaged over 1 second intervals, and the number of running threads, collected every 1 second. Clearly, there is a very good

TABLE IV. L3 CACHE MISSES PER COMPLETION (MPC)

Mix	*Policy Web*	*Policy DB*	*Web MPC*	*DB MPC*	*R*
browsing	HT	HT	9364	597	48
browsing	HT	NOHT	7121	557	48
browsing	NOHT	NOHT	4656	667	71
browsing	NOHT	HT	8789	721	258
ordering	NOHT	NOHT	6853	24486	659
ordering	HT	NOHT	17796	25912	1023
ordering	NOHT	HT	22036	20952	1249
ordering	HT	HT	33741	27723	1460

correlation between the peaks of the thread count series and the response time series. This completes the root cause analysis by confirming that the cache misses at the web tier result in large performance degradations at the database due to the subsequent increase of the number of active threads.

D. Summary

Summarizing, the impact of HT on cache performance, and hence on overall response times, can in general have complex dependencies with processor characteristics, application characteristics, tier configuration, multiprogramming limits, and workloads. This strongly motivates the need for a methodology that automatically selects between HT and NOHT for a given system based on the mix of incoming transactions into the system. For example, if over a given time window of arrivals there are more transactions that can benefit from HT than from NOHT, then a reasonable policy might be to turn on HT. In what follows, we propose a heuristic approach that leverages the results presented in this section and the previous section to classify transactions into two classes namely, HT-Friendly and NOHT-Friendly.

III. RUNTIME SMT PERFORMANCE OPTIMIZER (RPO)

The choice to enable HT depends on the transaction mix. In this section, we develop the Runtime SMT Performance Optimizer (RPO), a runtime controller that dynamically enables or disables HT based on the active transaction mix observed in operation. To do so, first we propose to classify requests based on their expected ability to benefit from HT. Next, we describe the architecture and implementation of RPO as an Apache module.

A. Request Classification

We begin with the observation that *browsing* favors HT while *ordering* favors NOHT. Then, in the proposed approach, a transaction which occurs more frequently in *browsing* than in *ordering* is classified as favoring HT (HT-Friendly). Other transactions that occur more frequently in *ordering* than in *browsing* are classified as NOHT-Friendly. When classification is applied to more than two mixes, this line of reasoning can be applied in the same way after forming two groups, one for experiments that benefit more from HT and one for those that favor NOHT. We note that more sophisticated per-tier classification schemes may be needed for systems where individual tiers are on different physical hosts. We defer this to future work.

The result of such a classification applied to *browsing* and *ordering* is shown in Table V. From the table, the HT-Friendly class contains mostly transactions that issue reads to

TABLE V. REQUEST CLASSIFICATION

Class Name	*Request Type*
HT-Friendly	Home, New Products, Best Sellers, Product Detail
NOHT-Friendly	Search Request, Search Results, Admin Request, Shopping Cart, Buy Request, Admin Confirm, Buy Confirm, Customer Registration

the database whereas the NOHT-Friendly class mostly contains transactions that write to the database. Furthermore, additional experiments, not reported due to space constraints, reveal that the mean service time for the majority of NOHT-Friendly transactions are two orders of magnitude higher than for HT-Friendly transactions. This suggests that, in the absence of historical information, a size based classification of transactions into large and small may be adopted as an alternative classification[1]. Using the proposed HT-Friendly/NOHT-Friendly classification, for *browsing* 73% of the total requests are HT-Friendly, whereas for *ordering* only 48% are HT-Friendly.

B. RPO Controller Design

The underlying idea of RPO is to estimate the mix of transactions in the application over a group of arrivals and activate HT only if HT-Friendly transactions form a relative majority in the system. This is achieved as follows. RPO observes requests arriving to Apache recording their type. Every time a batch of b requests has arrived, RPO computes the fractions of HT-Friendly and NOHT-Friendly transactions in the batch using the classification scheme discussed in the previous subsection. If there is a majority of HT-Friendly transactions in the batch, then HT is enabled until a new decision is computed for the following batch. Othewise, RPO switches to the NOHT policy. Note that RPO heuristically *estimates* the mix in operation based on historical information of the last b arrivals. For example, it does not guarantee that the first arrived request is still in the system at the time when the batch is completed and the SMT policy for the following period is decided. A more accurate approach requires synchronization between multiple Apache modules which could affect Apache performance. Furthermore, careful choice of the b batch size parameter can alleviate the impact of any inaccuracies from our approach as we show in Section IV-D.

The logic described above is implemented in RPO by three components, namely the *Request Collector*, the *Request Classifier*, and the *SMT Switcher*. The *Request Collector* component is in the path of Apache request processing and, in our implementation, it is a custom Apache module that hooks itself to the metadata processing phase of the web server. It intercepts a request just after it is received by Apache but before it is processed by web server content generation engine. The *Request Collector* extracts the name of the request URL from the request header and transmits it to the *Request Classifier*. The *Request Classifier* classifies each URL it receives into either HT-Friendly or NOHT-Friendly. Once b requests have been received, this component computes the fractions of HT-Friendly and NOHT-Friendly transactions for this batch. It then sends these fractions to the *SMT Switcher* component which either enables or disables HT according to the relative majority of HT-Friendly or NOHT-Friendly. We stress that *Request Classifier* and *SMT Switcher* work independently of Apache request processing, hence they do not produce direct overhead on incoming requests, yet they contribute to the processor utilization.

[1]We leave the investigation of this possibility for future work, however we have experimentally observed that this approach is promising.

C. Implementation

Several factors need to be considered to guarantee RPO effectivess. First, the work done by the *Request Collector* module should not be significant given that it is in the request processing path. Next, the batch size b must be chosen carefully: a large value will diminish the agility in responding to fluctuations in the mix and reduce accuracy in the estimation of the mix in the system. Conversely, a small value will cause frequent switching between HT and NOHT causing overheads. Finally, the time taken to calculate the mix of a batch and switch between HT and NOHT should be very small otherwise many transactions in the batch could complete before switching to the optimal policy for that batch. We experimentally characterize RPO overheads in Section IV-C and RPO sensitivity to the batch size parameter in Section IV-D.

The main challenge of RPO implementation is achieving a fast execution time for the *SMT Switcher* module. To simplify design and ensure low overhead, we have decoupled Apache request processing from the SMT policy switching. We have attempted to implement SMT policy switching via both the Linux CPU *hotplug* mechanism and the Linux *taskset* utility. However we have found in both cases the switching overheads to be too large with respect to the request response times. RPO solution is then to use the *cgroups* mechanism of Linux. This feature provides a way to map a set of processes and their future children to a group with specific attributes. Group attributes can be set to control the amount of resources allocated to its processes, e.g., the quantum of CPU and memory resources. To support RPO, we have created a group that contains the Apache and MySQL processes. We enable and disable HT by controlling the CPU resources assigned to this group. Assigning all cores in the system to this group is akin to enabling HT. Assigning only the common cores of NOHT and HT achieves the effect of switching to NOHT. We have found that this approach is remarkably faster than the previous two approaches. It takes around 1 ms to effect the switch. This is at least 5 times faster than the smallest transaction response time observed with the lightest possible system load (1 client).

IV. EXPERIMENTAL VALIDATION

We have performed an exhaustive validation of RPO: Section IV-A presents RPO results for the *browsing* and *ordering* mixes. Section IV-B evaluates RPO with custom mixes of *browsing* and *ordering* sessions. Section IV-C analyses the overhead introduced by RPO. Section IV-D performs sensitivity analysis on the batch size input parameter. If not otherwise stated, in the reported experiments we have configured RPO with a batch size $b = 20$. Furthermore, unless stated otherwise, the *Request Classifier* and *SMT Switcher* modules of RPO execute alongside the monitoring tools on processor 1 while the TPC-W processes belonging to the RPO-enabled Apache and MySQL execute on processor 0.

TABLE VI. EXPERIMENT RESULTS OF BROWSING AND ORDERING MIX WITH RPO

Mix	*Freq.*	*Cores*	*Policy*	*X*	*R*	*U*	*P*
browsing	1600	8	HT	356	21.2	0.55	165
browsing	1600	4	NOHT	356	43.1	0.82	170
browsing	1600	dynamic	RPO	356	22.4	0.55	167
ordering	2133	8	HT	48	103.7	0.33	164
ordering	2133	4	NOHT	48	50.0	0.48	179
ordering	2133	dynamic	RPO	48	56.3	0.27	170

Legend is same as Table I.

A. Results with browsing and ordering

Table VI compares the performance of HT, NOHT, and RPO for the TPC-W mixes on two representative cases of Table I. The results in Table VI demonstrate good promise for RPO. For both mixes, RPO performs almost as well as the best observed policy. Specifically, for *browsing*, the response time of RPO is almost the same as the response time for HT, which is the optimal policy for this mix. For *ordering*, RPO performance is close to that of the optimal NOHT policy. We note that although the mean response times with RPO are slightly higher than those of the corresponding best observed policies, the differences are not statistically significant. The 95% confidence intervals of mean response time for RPO and HT overlap for the *browsing* mix. Similarly, RPO and NOHT confidence intervals overlap for *ordering*. We also observe that the power consumption with RPO for a given mix is closer to the power consumption of the best observed performance SMT policy for that mix.

We have also investigated the distribution of response times for *browsing* and *ordering* for HT, NOHT, and RPO. We have found that, for *browsing* the distribution function of response times of HT and RPO is nearly identical. For *ordering*, RPO lies in the middle between the NOHT curve and the HT curve for the distribution body, but has a nearly perfect match of the NOHT distribution tail.

We have also tracked the decisions of the *SMT switcher* module over time. While for *browsing* RPO recommends that most of the time the system use HT, for *ordering* the situation is the opposite. Figure 2 plots a period of the experiment where *ordering* RPO frequently switches between HT and NOHT. This period is qualitatively very similar to the rest of the trace. In the figure a core count of 4 corresponds to NOHT while a core count of 8 corresponds to HT. The figure shows high dynamism of RPO, which switches SMT configuration policy every few seconds. Table VII illustrates the root causes for such a frequent switching behavior under *ordering*. All the batches encountered by RPO while serving *browsing* are dominated by HT-Friendly transactions. Thus, RPO enables SMT and performance is similar to HT. In contrast, a balanced amount of HT-Friendly and NOHT-Friendly batches appears in *ordering*, and this explains the frequent switches between the SMT policies.

Table VIII investigates the effect of the batching done by RPO. For example, in *browsing*, batches that have a majority of HT-Friendly requests still have a significant percentage, i.e., 27%, of NOHT-Friendly transactions. Similarly, the NOHT-Friendly batches in *browsing* have 44% of HT-Friendly transactions. Transactions that are a minority in a batch are unlikely to benefit from RPO policy choice. However, from the results

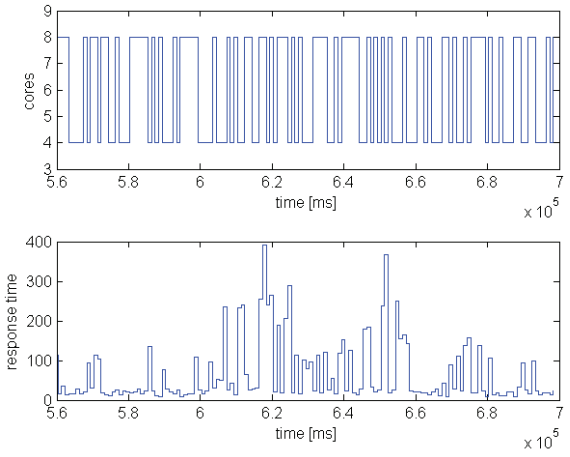

Fig. 2. Switching between HT and NOHT by RPO for the *ordering* workload.

TABLE VII. RPO BATCH CLASSIFICATION

Mix	batches	
	HT-Friendly	NOHT-Friendly
browsing	99.3%	0.7%
ordering	51.2%	48.3%

it appears that the incurred penalties do not dominate over the performance gains enjoyed by the majority of transactions in a batch. This argues for the robustness of RPO to the batching-based classification.

Summarizing, RPO does nearly as well as the optimal policies for the *browsing* and *ordering* mix. This is a powerful argument in favor of RPO since it eliminates the need for a system administrator to choose between HT and NOHT while configuring an enterprise system. In the next section, we construct customized mixes to see if there exist situations where RPO can even outperform both HT and NOHT.

B. RPO with Custom Workloads

We created custom workloads by mixing two groups of sessions, 2000 from the *browsing* mix and 2000 from the *ordering* mix. The first group of sessions has only HT-Friendly transactions while the second group has only NOHT-Friendly transactions. Sessions are defined in order to include all transactions types in Table V. All workloads contain the same set of sessions but in different orders, thus they have the same transaction mix of 71% HT-Friendly and 29% NOHT-Friendly transactions. Upon arrival of a session, a unique *switching probability* value is used to choose if the following session will be of the same type, i.e., HT-Friendly or NOHT-Friendly. By changing the value of the switching probability, we have defined four custom workloads: S(0), S(5), S(15), and S(50), where the number between brackets denotes the switching probability in percentage points. The S(0) workload has been generated in such a way to contain a single transition from HT-Friendly sessions to NOHT-Friendly sessions, where the NOHT-Friendly period lasts about 10 times the HT-Friendly one. Thus, S(0) exhibits *near* zero probability of switching between session types. Conversely, S(50) has the highest

TABLE VIII. RPO BATCHING EFFECTS

Mix	HT-Friendly requests in NOHT-Friendly batches	NOHT-Friendly requests in HT-Friendly batches
browsing	44%	27%
ordering	38%	44%

switching probability of 50%, which leads to observing a rapid alternation of session types in the system.

Since our previous experiments suggest that SMT selection is more crucial at high load, one of our objective was to observe system behavior at a consistently high load when both types of sessions are present. Specifying a single mean session inter-arrival time was insufficient to achieve this effect since the NOHT-Friendly transactions place much higher CPU demands than the HT-Friendly transactions. Therefore, we have specified different mean session inter-arrival times for HT-Friendly and NOHT-Friendly sessions. Specifically, a mean of $0.4s$ is used for NOHT-Friendly sessions, while we explore two different mean session inter-arrival times of $0.02s$ (high load) and $0.03s$ (lower load) for HT-Friendly sessions. As with the standard TPC-W workloads, both sets of inter-arrival times are exponentially distributed.

1) S(0) Results: Table IX presents the results of the S(0) workload with HT, NOHT, and RPO. For the first group in the table, the mean inter-arrival time for the HT-Friendly and NOHT-Friendly sessions is $0.02s$ and $0.4s$, respectively. In contrast to *browsing* and *ordering*, where RPO only did as well as the best observed policy for a given mix, RPO outperforms both HT and NOHT. One can observe that RPO is significantly better than solely using either HT or NOHT. The mean response time with RPO is about 40% better than that of the next best policy which is HT. The mean response time with NOHT is 3.2 times that of RPO. The 95% confidence intervals of mean response times suggest that these differences are statistically significant. We have also investigated the distribution of response times and found that RPO improves response times over all regions of the distribution.

For the second case in Table IX, the mean inter-arrival time for HT-Friendly sessions is increased from $0.03s$ to $0.02s$ while the mean inter-arrival time for NOHT-Friendly sessions is unchanged. This causes a drop in utilization during the 10% of the time when the system has solely HT-Friendly sessions. Since the utilization is unchanged for the rest of the 90% of the experiment duration, a period during which the system receives only NOHT-Friendly sessions, the overall utilization reduces only marginally from the previous case as seen for NOHT in Table IX. However, the response time behavior is markedly different from the previous case. In contrast to the high load case, NOHT now outperforms HT confirming results from our motivating example that the benefits of HT are apparent only at high load. Furthermore, RPO outperforms the best observed policy of NOHT, although the gains are more modest when compared to the high load case.

2) S(5), S(15), S(50) Results: The S(5), S(15), and S(50) workloads result in RPO batch classifications that are less biased to a single transaction type than for S(0), as shown in Table XI. Table X investigates how RPO reacts to this workload characteristic. In this experiment, we reduce the cores by half to 2 (NOHT) and 4(HT) to observe system behavior at a high load. From Table X, for all three workloads the performance of RPO is still the best, followed by HT, and then by NOHT. Specifically, RPO can speedup the mean response time by a factor of up to 3 over an incorrect SMT policy choice, i.e., NOHT. The mean response times with RPO are 2% to 15% lower than those of HT. However, in general the gains from RPO over the next best policy seem to diminish when the switching probability increases (Table XI). Results for a lower load scenario, not included due to space constraints, confirm previous findings that RPO performs the same as the best observed static SMT policy.

TABLE IX. EXPERIMENT RESULTS OF S(0) MIX WITH RPO

Mix	*Freq.*	*Cores*	*Policy*	*X*	*R*	*U*	*P*
S(0)	1600	8	HT	43	122.7	0.42	157
S(0)	1600	4	NOHT	43	233.4	0.57	164
S(0)	1600	dynamic	RPO	43	73.3	0.44	157
S(0)	1600	8	HT	43	147.6	0.42	158
S(0)	1600	4	NOHT	43	77.6	0.54	164
S(0)	1600	dynamic	RPO	43	72.3	0.49	156

Legend is same as Table I.

TABLE X. EXPERIMENT RESULTS OF S(5), S(15), S(50) MIX WITH RPO AT HIGH LOAD

Mix	*Freq.*	*Cores*	*Policy*	*X*	*R*	*U*	*P*
S(5)	1600	4	HT	36	162.9	0.64	154
S(5)	1600	2	NOHT	36	246.2	0.82	162
S(5)	1600	-	RPO	36	140.9	0.60	154
S(15)	1600	4	HT	36	143.6	0.66	154
S(15)	1600	2	NOHT	36	286.3	0.81	162
S(15)	1600	-	RPO	36	139.9	0.64	155
S(50)	1600	4	HT	36	155.8	0.66	154
S(50)	1600	2	NOHT	36	400.5	0.79	163
S(50)	1600	-	RPO	36	135.4	0.67	154

Legend is same as Table I.

Summarizing, this experimental campaign shows that RPO outperforms a static choice of SMT policies at high loads. Gains are particularly significant when there is pronounced skew to a single transaction type within RPO batches, as in the S(0) mix. As this skew diminishes, RPO still performs the best although gains become more modest due to a larger fraction of transactions in batches not benefiting from the SMT policy selected by RPO.

C. Overhead Analysis

Table XII exemplifies the overhead of adding the *Request Collector* module to the request processing path for *browsing* and *ordering* experiments. From the table, the mean response time with and without the RPO module is almost the same. This indicates that the module does not impose a significant overhead. We next characterize the impact of the other RPO components.

As stated earlier, two separate sets of cores have been used to run the *Request Classifier* and *SMT Switcher* modules of RPO and the processes belonging to the TPC-W system. Monitoring data shows that the *Request Collector* and *SMT Switcher* components add only 0.6% and 0.5% to the mean utilizations of their respective cores. We next evaluate the overhead of RPO when there are no spare cores available for these 2 components and they are forced to share the cores used by the TPC-W processes.

TABLE XI. RPO BATCHES ON CUSTOM MIXES

Mix	HT-Friendly requests in NOHT-Friendly batches	NOHT-Friendly requests in HT-Friendly batches
S(0)	0%	0%
S(5)	32%	22%
S(15)	40%	25%
S(50)	42%	27%

TABLE XII. OVERHEAD OF RPO *Request Collector*

Mix	*Freq.*	*Cores*	*RPO*	*X*	*R*	*U*
browsing	1600	8	OFF	356	22.6	0.58
browsing	1600	8	ON	356	21.2	0.55
ordering	1600	8	OFF	48	42.00	0.22
ordering	1600	8	ON	48	42.05	0.22

Legend is same as Table I.

TABLE XIII. OVERHEAD CHARACTERIZATION OF RPO

Mix	*Freq.*	*Cores*	*X*	*R*	*U*
browsing	1600	4 (0,2,4,6)	416	70.20	0.71
browsing	1600	2 (3,7)	416	73.20	0.71
browsing	1600	4 (0,2,4,6)	357	41.55	0.59
browsing	1600	2 (3,7)	357	39.95	0.59
ordering	1600	4 (0,2,4,6)	22	132.55	0.36
ordering	1600	2 (3,7)	22	130.1	0.36

Legend is same as Table I.

TABLE XIV. IMPACT OF BATCH SIZE *b* OF RPO

Mix	*b*	*Freq.*	*X*	*R*	*U*
S(50)	100	1600	36	175.00	0.68
S(50)	40	1600	36	160.23	0.68
S(50)	20	1600	36	135.38	0.67
S(50)	10	1600	36	138.17	0.67
S(50)	5	1600	36	166.03	0.65
S(50)	1	1600	36	324.67	0.65

Legend is same as Table I.

Table XIII shows the results of this experiment. We first consider experiments with the *browsing* workload that cause a high per-core processor utilization of 71%. For the first set of these experiments we pin *Request Classifier* and *SMT Switcher* to the same cores (core 0,2,4,6) where the Apache and MySQL processes are executing. For the second set we pin *Request Classifier* and *SMT Switcher* to cores 3 and 7 that were not used by the TPC-W processes. From Table XIII, the mean response times are quite close in both cases. The table also shows similar trends for a lower load *browsing* case as well as an *ordering* case. These results suggest that RPO imposes very little overheads and can be even used in high load scenarios, which can benefit significantly from dynamic adaptation between HT and NOHT.

D. Batch Size Sensitivity

Finally, we conclude the validation by exploring the impact of the batch size parameter *b* of RPO. We compare the results obtained from the S(50) workload using several batch sizes. Results are shown in Table XV, while Table XIV shows mean response times obtained with RPO. Table XV indicates that small batch sizes result in better RPO performance. However, Table XIV indicates that small batch sizes, and thus the frequent use of the *SMT Switcher*, trigger high latency overheads due to the cost of migrating the context of processes, e.g., cached data, from the deactivated cores or viceversa. Conversely, setting batch values to the order of several tens becomes ineffective with respect to the workload fluctuations. Batch sizes of 10-20 seem to strike the right balance between reducing these overheads and achieving agility in reacting to incoming workload changes.

V. RELATED WORK

A vast majority of studies that characterize the behavior of SMT have focused on scientific and batch workloads. While some of these studies [4] report consistent performance benefits from SMT, other studies indicate that SMT can be harmful for some workloads [7] and may only be effective at high loads [10]. Furthermore, performance benefits of SMT processors have been found to depend on task co-scheduling policies and cache performance [11], [12]. Relatively few studies have focused on SMT impact on components of multi-tier applications deployments, such as web and database servers. Lo *et al.* examined database performance on SMT processors using traces from an Oracle database management system (DBMS) [8]. Using simulation, the authors show that SMT can introduce additional data cache conflicts. Hassanein *et al.* [9] instead report a performance improvement with HT of up to 16% in TPC-C queries and up to 26% in TPC-H queries on a Pentium 4 processor. Ruan *et al.* [6] evaluated three versions of the Hyper-Threaded Intel Xeon processor for several Web servers. Their evaluation suggested that the Xeon processor can provide better performance gain with SMT only when it employs larger L3 caches. The only study we are aware of that focuses on a multi-tier application is that by Cain *et al.* [13]. In contrast to our work which considered a real system, the authors study the behavior of a Java-based TPC-W application on a simulated SMT processor. Their results show that SMT can cause higher cache misses but can achieve higher throughputs due to its ability to hide long memory latencies. A larger body of work has focused on automatically reconfiguring a multi-tier system in response to workload fluctuations, e.g., [5]. However, these studies only explore strategies such as modifying the amount of resources, e.g., processors, allocated to application tiers. To the best of our knowledge, no previous case exists in the literature of a performance management controller integrating SMT control for a multi-tier web application.

VI. CONCLUSION

This paper has presented a case for automatically managing the choice of SMT policy for multi-tiered systems. We have proposed the RPO solution that implements such a capability. Extensive validation results show that RPO outperforms a static choice of SMT configuration policy for many workloads. For the other workloads, it typically performs closely to the optimal static configuration policy. RPO is robust towards workloads that display rapid alternations between SMT friendly and SMT unfriendly mixes. Also, it places negligible overheads on core utilization and transaction response times.

Future work will focus on generalizing our results for other SMT processors, applications, multi-tier configurations, and workloads. Our preliminary results on an alternative system setup where the web and database tiers were each allocated their own dedicated processor yielded similar trends leading us to believe that RPO may be effective for more complex multi-tier deployments as well. Our effort will also be centred

TABLE XV. RPO BATCHES FOR VARIOUS b VALUES

Mix	b	HT-Friendly requests in NOHT-Friendly batches	NOHT-Friendly requests in HT-Friendly batches
S(50)	100	49%	28%
S(50)	40	46%	28%
S(50)	20	42%	27%
S(50)	10	37%	26%
S(50)	5	35%	21%
S(50)	1	0%	0%

around automating the classification of the SMT friendliness of transactions and the choice of the optimum batch size.

ACKNOWLEDGEMENT

This work has been financially supported by Natural Sciences and Engineering Research Council (NSERC) Canada, Hewlett Packard Labs, and by the European project MODAClouds (FP7-318484).

REFERENCES

[1] Intel Hyper-Threading. http://www.intel.com/content/www/us/en/architecture-and-technology/hyper-threading/hyper-threading-technology.html

[2] J. R. Bulpin and I. A. Pratt HT Aware Process Scheduling Heuristics. USENIX Annual Tech. Conf. pp. 399-402, 2005.

[3] J. L. Lo, J. S. Emer, H. M. Levy, R. L. Stamm D. M.Tullsen, and S. J. Eggers. Converting thread-level parallelism to instruction-level parallelism via simultaneous multithreading. ACM T. on Comp. Sys., 15(3):322–354, Aug 1997.

[4] N. Tuck and D. M. Tullsen. Initial observations of the simultaneous multithreading Pentium 4 processor. Proc. of PACT, pp. 26–34. IEEE Comp. Soc., Sep 2003.

[5] R. Singh, U. Sharma, E. Cecchet, and P. Shenoy. Autonomic Mix-Aware Provisioning for Non-Stationary Data Center Workloads. Proc of ICAC, 21–30. Jan 2010.

[6] Y. Ruan, V. S. Pai, E. Nahum, and J. M. Tracey. Evaluating the Impact of Simultaneous Multithreading on Network Servers Using Real Hardware. Proc. of ACM SIGMETRICS, 315–326, 2005.

[7] H. M. Mathis, A. E. Mericas, J. D. McCalpin, R. J. Eickemeyer and S. R. Kunkel. Characterization of simultaneous multithreading (SMT) efficiency in POWER5. IBM J. RES. & DEV., 49(4/5), Jul 2005.

[8] J. L. Lo, L. A. Barroso, S. J. Eggers, K. Gharachorloo, H. M. Levy, and S. S. Parekh. An Analysis of Database Workload Performance on Simultaneous Multithreaded Processors. Proc. of ISCA, Jul 1998.

[9] W. M. Hassanein , L. K. Rashid, M. A. Hammad. Analyzing the Effects of Hyperthreading on the Performance of Data Management Systems. Int. J. of Parallel Programming, 36:206–225, Apr 2008.

[10] TMurgent Technologies. Hyper–Threading and Multiiprocessor System Performance On Server 2003. White paper. June 25, 2003.

[11] J. Kihm, A. Settle, A. Janiszewski and D. Connors. Understanding the Impact of Inter-Thread Cache Interference on ILP in Modern SMT Processors. J. of Instruction-Level Parallelism, pp.1-28, June 2005.

[12] S. Parekh, S. Eggers, H. Levy and J. Lo. Thread-Sensitive Scheduling for SMT Processors. U. of Washington Tech. Rep. 2000-04-02, 2000.

[13] H. W. Cain, R. Rajwar, M. Marden, and M.H. Lipasti. An architectural evaluation of Java TPC-W. Proc. of the International Symposium on High-Performance Computer Architecture, 229–240, 2011.

[14] J. R. Funston, K. E. Maghraoui, J. Jann, P. Pattnaik, and A. Fedorova. An SMT-Selection Metric to Improve Multithreaded Applications' Performance. 2012 IEEE 26th International Parallel and Distributed Processing Symposium , pp. 1388–1399, 2012

[15] C. Amza, A. Chanda, A.L. Cox, S. Elnikety, R. Gil, K. Rajamani, W. Zwaenepoel, E. Cecchet, and J. Marguerite, Specification and Implementation of Dynamic Web Site Benchmarks. Proc. of IISWC,3–13, Nov 2002.