2013 IEEE 21st International Symposium on Modelling, Analysis & Simulation of Computer and Telecommunication Systems

Overcoming Limitations of Off-the-shelf Priority Schedulers in Dynamic Environments

Feng Yan[1], Shannon Hughes[1], Alma Riska[2], and Evgenia Smirni[1]

[1]College of William and Mary, Williamsburg, VA, USA, fyan,srhughes,esmirni@cs.wm.edu
[2]EMC Corporation, Cambridge, MA, USA, alma.riska@emc.com

***Abstract*—It is common nowadays to architect and design scaled-out systems with off-the-shelf computing components operated and managed by off-the-shelf open-source tools. While web services represent the critical set of services offered at scale, big data analytics is emerging as a preferred service to be co-located with cloud web services at a lower priority raising the need for off-the-shelf priority scheduling. In this paper we report on the perils of Linux priority scheduling tools when used to differentiate between such complex services. We demonstrate that simple priority scheduling utilities such as `nice` and `ionice` can result in dramatically erratic behavior. We provide a remedy by proposing an autonomic priority scheduling algorithm that adjusts its execution parameters based on on-line measurements of the current resource usage of critical applications. Detailed experimentation with a user-space prototype of the algorithm on a Linux system using popular benchmarks such as SPEC and TPC-W illustrate the robustness and versatility of the proposed technique, as it provides consistency to the expected performance of a high-priority application when running simultaneously with multiple low priority jobs.**

I. Introduction

Computer systems composed of off-the-shelf hardware running open-source operating systems are evolving to support emerging web applications and big data analytics at a large scale. The goal, exemplified by the *Open Compute* initiative, which was started by Facebook but has been widely adopted by the larger tech community, is to keep down the cost of very large systems, data, computation, and overall web services. While traditional computer systems, particularly those supporting enterprise applications, have included sophisticated (and often proprietary) resource management and scheduling modules, the industry is increasingly turning to commodity hardware and un-modified software to accomplish large-scale services and computation.

This trend offers a challenge. Such systems are expected to run a wide array of web applications alongside significant support applications ensuring data redundancy and computation to deliver individualized services to customers. For example, in a system with tens to hundreds of nodes supporting a web store, there may be background processes that would need to analyze the logs collected during the operation of the system, to generate preferences of the web store users, or even to analyze failures and generate failure detection, isolation, and handling rules that are critical in ensuring resilience at scale [1]. In order to enable systems of the scale proposed in [2], the system architecture would follow the *shared-nothing* model [3] where individual servers (or nodes) operate independently in a large distributed system while exchanging messages with other participating servers (or nodes). The individual servers are expected to be off-the-shelf hardware running general-purpose operating systems such as Linux, yet still provide enterprise-grade computing services.

In such systems, the focus is on the ability to locate important but relatively time-insensitive tasks alongside user-facing applications where demand fluctuates semi-randomly and short response time is critical to meeting service level agreements and maintaining user engagement. This problem, i.e., prioritizing systems work, is often solved via scheduling policies at the kernel [4], [5] or at the application level [6], [7], [8], [9]. In Linux, the most popular commodity operating system, priority scheduling is achieved through `nice` and `ionice`. Though these are static prioritization tools, dynamic amount of CPU and memory resources can be allocated to a given process via `renice`.

Using the web-driven TPC-W benchmark [10], [11] as a representative foreground application, we experiment with a number of background tasks from the SPEC benchmark suite [12] and also our own microbenchmark. We find that `nice` is at best erratic in its ability to isolate the performance of the high priority, time-sensitive application from the low priority time-insensitive background work in the system. We also explore the effect of adding `ionice` prioritization to the background processes, which helps background jobs that require significant memory resources, but can seriously damage effectiveness in CPU-intensive cases.

To address the limitations of off-the-shelf prioritization tools like `nice` and `ionice`, we develop `smart`, a portable tool which runs in user-space and observes the behavior of the foreground task in order to calculate reasonable parameters for suspending and resuming background work. Extensive experimentation shows that `smart` effectively utilizes system resources by scheduling background jobs only when these resources are lightly utilized by high priority processes. Additionally, `smart` better isolates the foreground performance than `nice` or `nice` plus `ionice`, as well as doing so more consistently than either of those off-the-shelf options. Thus, `smart` can be seen as a more intelligent tool that can effectively differentiate the level of service received by high and low priority processes, independent of the complexity of competing workloads.

The rest of the paper is organized as follows. In Section II, we provide results from characterizing the behavior of `nice` under several scenarios. Section III develops a new prioritization scheme which determines when and for how long to schedule the low priority processes according to the CPU utilization in the system. The new framework is evaluated via extended experiments in Section IV. We conclude the paper and summarize future work in Section VI.

1526-7539/13 $26.00 © 2013 IEEE
DOI 10.1109/MASCOTS.2013.72

II. Background and Motivation

Proprietary systems often have their own scheduling algorithms that allow them to maintain performance of user workload while other lower priority jobs are running in the background. The available off-the-shelf tools for priority scheduling in any Unix-based system are `nice`, which prioritizes access to the CPU resource, and `ionice`, which prioritizes access to the disk resource. While different distributions of Unix have different implementations of `nice` and `ionice`, they operate similarly: when enabled, they allow users to adjust the execution priority of processes.

A process that is invoked via `nice` can have a scheduling priority between -20 (the highest priority) and 19 (the lowest priority), as determined by a single parameter in the `nice` command. If the priority parameter of `nice` is set to zero or the process is invoked without the `nice` command then the process is run with the default (i.e., normal) priority. `nice` uses the priority parameter to determine the chunk of CPU time for a specific process, i.e., the higher the priority the larger the chunk of CPU time the process gets. The exact relation between the `nice` parameter and the amount of CPU time dedicated to a process are implementation dependent and vary between Unix/Linux distributions. The mechanism is generally simple to use and depends on fine-grained CPU consumption.

Similarly, `ionice` allows ranking the priority of a process from 0 to 3, where 3 is meant to designate a process that should be given IO resources only when the IO system is otherwise idle. A user may select to invoke both `nice` and `ionice`. In our experiments, we combine `nice 19` with `ionice 3` to give the lowest priority setting for both resources, which we label "allnice".

Independent of which resource the `nice` or `ionice` tool try to prioritize, they differentiate concurrent jobs by giving them a different time share on the resource. The time share depends on the total demands of *all* concurrent jobs, irrespective of their priority. Consequently, higher priority jobs receive their *proportional* share of the available resource rather than their own *absolute* demand. Thus, differentiating via `nice` or `ionice`, which operates at the kernel-level and in fine-grain time scales, may result in fluctuating performance for higher priority jobs especially if their resource demands fluctuate across time or high demands in multiple resources. Isolating the performance of high priority jobs under such conditions becomes challenging.

To illustrate the ineffectiveness of `nice` and `ionice` in preserving performance of high priority processes, we measure the slowdown of a high priority workload ("foreground") when executed concurrently with a low priority ("background") workload. All workloads are selected from the SPEC benchmark suite [12], see Section IV-A for a detailed description of the experimental setup. The foreground benchmark is scheduled using the default priority in the OS scheduler (i.e., corresponding to the value 0 of the `nice` priority parameter), while the background job is executed with `nice` with parameters ranging from 0 to 19 and also "allnice", i.e., with the lowest CPU and IO priority. The execution time of the foreground job is our target performance measure.

We show two scenarios in Figure 1, i.e., one where allnice works as expected (left graph), and one where `nice` and `ionice` fail (right graph). In both graphs, the x-axis illustrates the "amount" of the background work that is executed

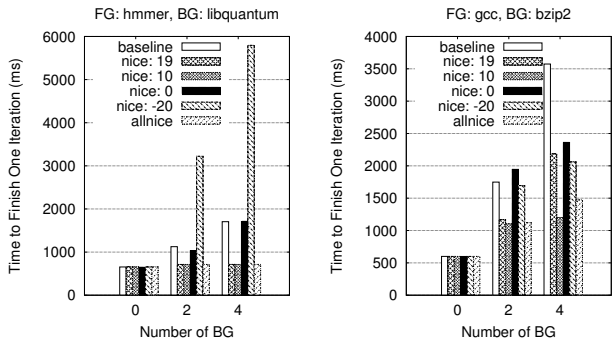

Fig. 1. Foreground performance with 2 and 4 background jobs (all jobs are SPEC benchmarks). For each experiment, the background job runs with default priority (no `nice`, baseline case), with `nice` 19, 10, 0, -20, and "allnice" (i.e., `nice 19` and `ionice 3`).

concurrently with the foreground job, i.e., no background, 2, and 4 concurrent background instances. In the left graph, SPEC `hmmer` is the foreground job and SPEC `libquantum` is the background one. In this experiment, `nice` and `ionice` are effective in isolating the performance of the foreground workload independent of the amount of background work in the system. Specifically, the baseline case (where the background job is scheduled without `nice`) achieves the same foreground performance as when the background job is scheduled with `nice 0`, as expected. Experiments where `libquantum` is scheduled with `nice 10` and `nice 19` maintain the performance of `hmmer` at the same level as without any background job. When `libquantum` is scheduled with priority -20, then indeed its priority is higher than that of `hmmer` and as a result `hmmer` suffers from significant performance slowdown.

The right graph of Figure 1 shows a very different behavior. Here, the performance trends of the foreground job become clearly unpredictable and does not follow the relative priority set by `nice`. In some cases *any* priority parameter (even -20) works better for the foreground job (see the second bar from right in the graph with 4 `bzip2` as background jobs). With "allnice", foreground performance improves but `gcc` still suffers from unexpected delays.

The results in Figure 1 corroborate that `nice` does not isolate performance of high priority jobs in the presence of memory and IO demands from the background jobs. The problem persists even with the added boost of `ionice`. A straightforward approach to remedy this problem is to limit the impact of background jobs on foreground performance by slowing down the background jobs in periods of CPU contention *only*, by suspending their execution periodically during those times.

To give a first proof-of-concept that intelligent suspending of the background will work, we conduct a controlled experiment where the foreground workload is TPC-W [10], a web-service benchmark that has significant variability across time in its CPU and memory demands [13] and background work consists of 2 and 8 simultaneous executions of `hmmer` from the SPEC suite. In these experiments, we deliberately control the demands of TPC-W on the various resources by changing the number of its emulated browsers such that we know *a priori* when TPC-W's resource demands are high or low. Figure 2 plots the CDH (Cumulative Distribution Histogram) of TPC-

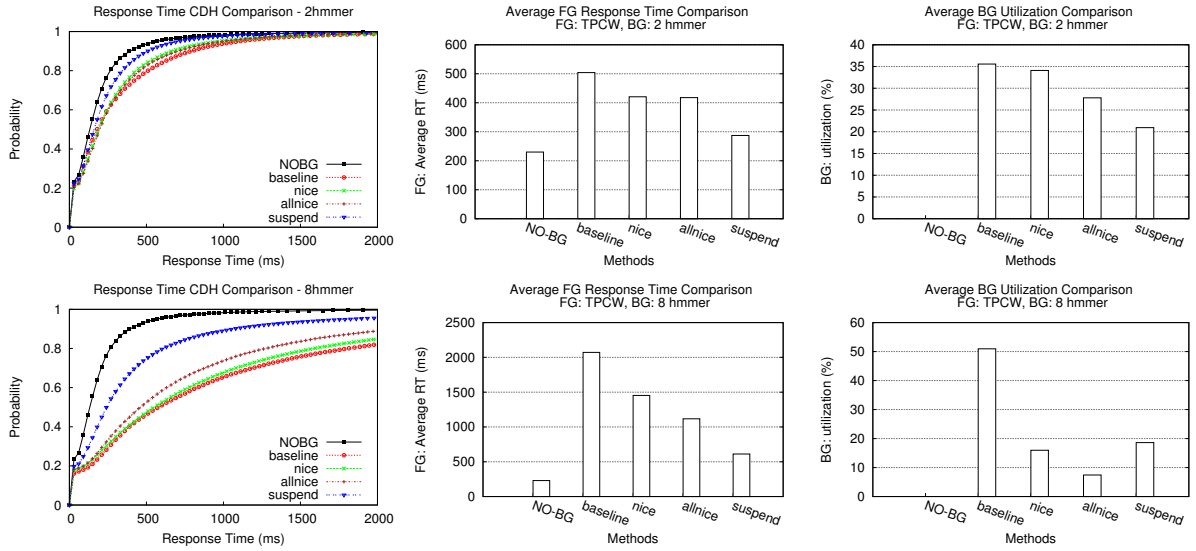

Fig. 2. Performance comparison for periodic suspending of the background jobs according to *a prior* known foreground behavior.

W's transaction response time, the overall average response time of the TPC-W transactions, and the CPU share that is allocated to the background job. We report on four different ways of handling the background work: 1) invoking it with the default OS priority, (i.e., without `nice` or `ionice`, this is the baseline case), 2) invoking it using `nice 19` (the lowest CPU priority; we label this "nice"), 3) invoking it using `nice 19` and `ionice 3` (the lowest CPU and IO priority; we label this "allnice"), and 4) suspend the execution of background job in periods of high CPU utilization (which are known *a priori*; we label this "suspend"). As a reference, the TPC-W response time with no background job is also reported. We observe that `nice` is very ineffective for both experiments (see the two rows of graphs in Figure 2, the top representing a light background workload of 2 `hmmers` and the bottom representing a heavy background workload of 8 `hmmers`). "allnice" improves TPC-W's performance compared to `nice` but `suspend` is steadily a better option. Remarkably, it improves TPC-W's performance while not starving the background job, see the rightmost column of background utilization graphs. These results motivate us to develop a smart scheduler that can remedy the pitfalls of `nice` and `ionice` while also being lightweight and easy to use.

III. Methodology

Parameter	Description
$UTIL_{user}$	FG CPU util during the last Monitoring Window
$UTIL_{noBG}$	long term average FG CPU util, run in isolation
$UTIL_{wBG}$	long term average FG CPU util, run with BG
BW_{noBG}	Bursty Window size of FG CPU util, run in isolation
BW_{wBG}	Bursty Window size of FG CPU util, run with BG
BG_{status}	BG status, active or suspend

TABLE I. Summary of the parameters in algorithm.

As discussed in Section II, when CPU increases and there is contention among processes to utilize the resource, `nice` scales down the share of all processes according to their priorities. In a system where low priority jobs are treated as best-effort processes rather than requiring steady but limited resources, the performance of high priority jobs can be significantly improved by allowing the background jobs to utilize the CPU only during periods when the high priority processes are not requesting a large portion of the resource.

As clarification, we illustrate how the foreground workload, TPC-W, uses the CPU across time, see Figure 4. The CPU utilization and corresponding TPC-W response times are shown across time. In the top graph, TPC-W executes in isolation, i.e., there is no background work other than system processes. We can see that there are lulls, when TPC-W is not demanding much CPU time, these periods correspond to low TPC-W response times. These are the time periods where it would be most beneficial to schedule the background work. In the second graph, we see the behavior of TPC-W with 2 simultaneous executions of `hmmer`, the graph corresponds to the default priorities, i.e., the baseline experiment. Here, we see the dramatic effect of uncontrolled executions of `hmmer` on TPC-W's response time. In addition, the change trends of CPU utilization and response times suggests that monitoring the CPU utilization is a good choice for our purpose.

Our proposed algorithm, `smart`, aspires to schedule background/lower priority jobs only during low-demand time periods when the background jobs are not going to damage the responsiveness of the high priority jobs. When foreground jobs have high CPU demands, `smart` chooses to suspend the background jobs rather than allow the built-in scheduler to scale back all running processes to fit the available resources.

The `smart` scheduling algorithm observes the behavior of the foreground process to determine what level of CPU demand constitutes "high" activity for that process and how long the periods of high activity last. This allows `smart` to determine *when* to suspend the low priority processes and *for how long* to keep them suspended before checking the foreground demand again. The main premise of this scheduling algorithm is that the foreground job is expected to be an interactive application, having periodic bursts of demand punctuated by periods of low usage. Furthermore, we take response time to be the best measure of the foreground's performance, due to the interactive

nature of the application. By scheduling background work during the lulls in foreground activity and suspending them during the peaks, we can protect the responsiveness of the foreground job while serving background work at the most opportune times.

The algorithm monitors the foreground job both while it runs alone and while it runs with the background work. From the data collected, `smart` "learns" the stochastic characteristics of the foreground resource demands. Specifically, `smart` monitors the CPU utilization at 10 second intervals and at the end of the observation period, categorizes each interval as being either of high utilization or low utilization, based on the average observed utilization during the period. Finally, `smart` uses the collected information to determine the average window length of consecutive high utilization intervals, which we term Bursty Window (BW) length.

When `smart` is in active scheduling mode, it continues to monitor the foreground process at 10 second intervals as long as the background jobs are running. When an interval of higher than average utilization is detected, the background processes are suspended for a BW amount of time. After BW time elapses, `smart` checks the foreground utilization again and resumes the background processes only if CPU utilization is below the average value. Otherwise, `smart` keeps the background jobs suspended for another BW period. The scheduling decisions made by `smart` are based on the appropriate average utilization and BW lengths for the current system state.

1. if system in *characterization state* do
collect utilization information to calculate
$UTIL_{noBG}$, $UTIL_{wBG}$, BW_{noBG} and BW_{wBG}.
2. if system in *scheduling state* do
a. if BG_{status} = *suspend*
i. if $UTIL_{user} < UTIL_{noBG}$
resume BG work
$BG_{status} = active$
wait Monitoring Window
ii. else if $UTIL_{user} >= UTIL_{noBG}$
wait BW_{noBG}
iii.go to Step **2.a**
b. else if $BG_{status} = active$
i. if $UTIL_{user} < UTIL_{wBG}$
wait Monitoring Window
ii. else if $UTIL_{user} >= UTIL_{wBG}$
suspend BG work
$BG_{status} = suspend$
wait BW_{wBG}
iii.go to Step **2.a**
3. if detect system change events (e.g., new application added, system upgrade, system failure, etc.)
go to Step **1**

Fig. 3. The algorithm of smart scheduling.

A summary of the main parameters used in `smart` is given in Table I. All parameters labeled as $noBG$ correspond to measurements with no active background jobs, while wBG corresponds to measurements with active background work. The algorithm itself is given in Figure 3. We emphasize that the CPU utilization patterns may change over the time and keep on updating the scheduling parameters can reflect these changes, as described in the Step 3 in the algorithm, such update can be event driven or periodical.

Finally, the monitoring and suspending/resuming tools required by the algorithm are handy in the Linux system, so the algorithm is lightweight and can be implemented and deployed in the user space easily, please see experiment setup for more details about the prototype we implemented. By using the system tools, the overhead of the algorithm is almost negligible.

IV. Experimental Evaluation

In this section, we evaluate our proposed scheduling algorithm. First we give an overview of the experimental setup and then we outline and discuss our results.

A. Experimental Setup

All experiments presented in this paper are conducted on a Dell Precision WorkStation with Intel Pentium Dual Core 2.4GHz processor, 1GB memory, Seagate 7.2K SATA hard drives, running openSUSE 11.4 (64 bit). As foreground workload, we use a Java implementation of the TPC-W benchmark. As background, we use benchmarks from the SPEC CPU2006 suite and our own microbenchmark.

TPC-W is a web server and database performance benchmark. The Java implementation that we use in this paper is developed from the distribution by the University of Wisconsin - Madison[14]. We use tomcat as the web server and mysql as the database server. TPC-W provides a large number of parameters. We use the browsing mix with 50 emulated browsers and 100000 items in the database. TPC-W is a challenging workload for our purposes here because it is characterized by continuous variability in its resource demands across time [11], as also shown in Figure 4.

SPEC CPU2006 is an industry-standard, CPU-intensive benchmark suite [12]. SPEC CPU2006 is composed from a series of real-world applications designed to stress CPU and memory usage. The five workloads from SPEC we use here are the following: `bzip2` performs compression, decompression, and checking against the original at several compression scales for sample input. The SPEC version of the bzip2 algorithm prevents any IO beyond the initial read of the input so as to make this benchmark CPU and memory intensive with very little IO activity. `gcc` performs optimized compilation of a large sample program, with a slight alteration to the gcc algorithm to force more memory usage than would be typical of the real-world gcc compiler. `hmmer` performs searching and ranking of sequence matches in a database, simulating gene sequence matching. `libquantum` simulates factorization as it would be performed on a quantum computer. `povray` is a ray-tracer that simulates the way rays of light travel in a scene. For a fair comparison, when the SPEC benchmark is used as background workload, we repeat its execution so that it ends at the same time with TPC-W.

We also wrote our own microbenchmark to run as background work. Most importantly, this allows us to get precise metrics for the behavior of the background task under `smart` and the comparison methods. Additionally, the microbenchmark allows us to experiment with a broader range of CPU, memory, and IO demands from the background task. In the results reported here, the microbenchmark steadily consumed approximately 25% of the system's total CPU resource and

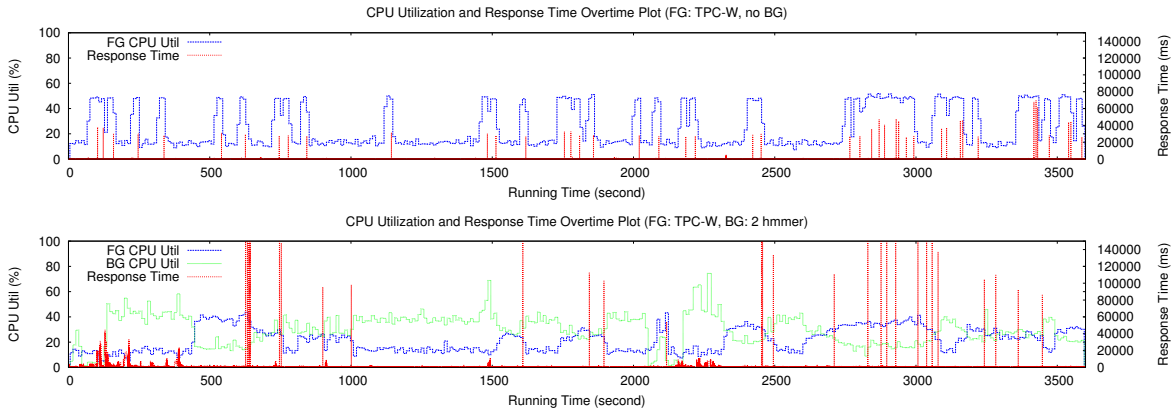

Fig. 4. TPC-W utilization and response time across time without background work and under 2 hmmer bookmarks as background work.

20% of memory capacity [1]. This is accomplished simply by performing multiplications in a tight loop which is embedded in a larger loop containing array initialization and file writes.

B. Implementation

In order to provide a simple and easily portable implementation, our monitoring and scheduling algorithms are implemented entirely in user space, making use of the readily available Linux commands (e.g., `pidstat` and `kill`). For monitoring, we launch a shell script to call classifying the results into three main categories: foreground (TPC-W-related) processes, background (SPEC-related or microbenchmark) processes, and other system processes. The coarse granularity of our intervals is quite different from what is generally seen in scheduling algorithms in the literature, where measurements and decisions are made at the microsecond level. The long intervals are actually a benefit to our method, since we are performing a predictive analysis of trends. Th monitoring interval here 10 seconds for the purpose of balancing overhead and accuracy.

To control the execution of the background work, we use the STOP and CONT signals and pass them to process by the `kill` command to "pause" and "resume" the background task execution. The process is suspended by being starved of resources, but because it is not actually killed, it can be immediately resumed from where it is paused.

C. Results

To thoroughly evaluate the performance of `smart`, we run TPC-W as the foreground task with a variety of SPEC benchmarks and our own microbenchmark as background tasks. In Figures 6, 7, and 8, we report key performance metrics from each experiment in order to compare the `smart` algorithm to four existing possibilities. As an upper bound, we take the case where TPC-W runs alone with no background tasks. As a lower bound, the "baseline" case is where TPC-W and the background tasks run together with no attempt to control their behavior. We also include two competitors to the `smart` algorithm: the case where the background task runs under lowest `nice` priority and the "'allnice" case, where the background runs with both lowest `nice` priority and lowest `ionice` priority.

To evaluate the performance of the foreground tasks, we focus on the CDH of the TPC-W response times, which is the best measure of perceived responsiveness for an interactive application. The horizontal axis shows response time lengths in milliseconds, while the vertical axis shows probability. Thus, if the curve intersects the point (500, 0.80), this means that 80% of the observed response times were 500 ms or less in that experiment. So, the more quickly the curve rises initially and the more tightly it makes the knee bend toward the asymptote of 1, the better the foreground performance the users perceive.

First, we conduct experiments to show the importance of "when" to initiate the suspension of background work. In this experiment, we also report results with a periodic suspension of the background job (labeled "suspend"). Note that in this experiment we assume no *a priori* knowledge of the foreground workload demands, i.e., the experiment is not the same as in Figure 2 where perfect future workload knowledge was assumed. Figure 5 illustrates than when compared to the periodic suspension, which is completely oblivious of the variability in resource demands of the foreground work, `smart` improves response time for the foreground task without further reducing the CPU time given to the background task. Indeed, in the 2 `hmmer` case, `smart` actually increases the resources given to the background (see the utilization graphs), without hurting the performance of TPC-W. We can conclude that the `smart` strategy is an improvement over the periodic sleep strategy.

Observation 1: The `smart` implementation improves foreground performance and sometimes also background performance better than periodically throttling the background tasks.

We show the results of TPC-W run with our microbenchmark as background in Figure 6. We can see that `smart` is able to significantly improve the response time for TPC-W, e.g., up to 60% less compared to nice, while at a relatively low expense of slowing down the background task, as can be seen in the BG iterations and BG utilization graphs in the third and fourth rows. Furthermore, the `smart` algorithm is less susceptible to degrading the foreground response time in the presence of a larger number of background processes than the other methods.

[1] we also experiment with different CPU, memory and IO demands settings for the microbenchmark, but due to the interest of space and similar observations, we only show one case here as an example.

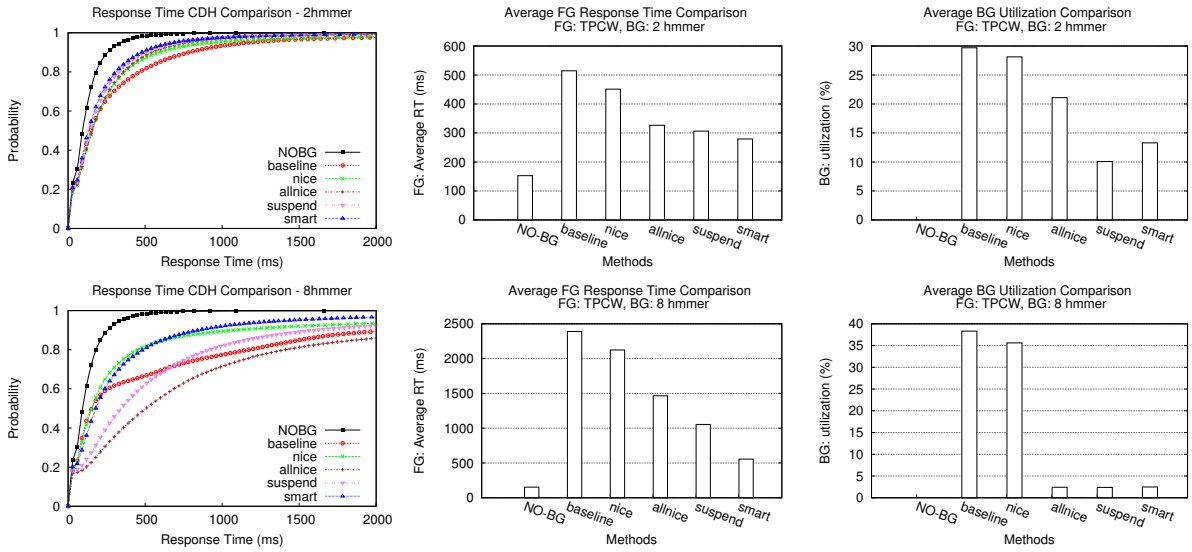

Fig. 5. Performance comparison between suspend, smart and other methods.

Observation 2: The `smart` implementation is a significant improvement over off-the-shelf methods, especially in cases of memory contention.

For the rest of our experiments, we report only the utilization given to the background task because the SPEC benchmarks take too long to complete to be able to use the number of iterations as a useful metric. As expected and shown in the microbenchmark results in Figure 6, the amount of utilization given to the background task closely aligns with the number of complete iterations.

Figure 7 shows results for two instances of SPEC background tasks running with TPC-W. It is easy to see that for the `gcc` and `bzip2` background cases, the `smart` algorithm bends the response time CDH very close to the ideal case of no background work, significantly improving this key metric over the result achieved by `nice`, which actually performs worse than the expected lower bound `baseline` case, or "allnice", which does little better than the lower bound. We note that `smart` makes this dramatic improvement in the foreground response time while giving the background task approximately the same, admittedly small, CPU share as "allnice".

In the case of `povray` as shown in Figure 7, `smart` still improves the response time of the foreground, but not as dramatically. Compared to the `gcc` and `bzip2` cases, there is simply less room for improvement between the lower and upper bounds. This is because `povray` is a CPU-intensive benchmark with relatively less IO and memory activities, which plays to the strengths of the built-in schedulers. It is worth noting, however, that `smart` also gives a small boost to the background performance while improving the foreground performance.

Observation 3: The `smart` implementation has the potential to produce a win/win situation, where both the foreground and background performance benefit over the built-in priority methods.

In Figure 8, we report analogous results for experiments with 8 instances of the background task running. These results show a much bigger spread between the upper and lower bounds for all three cases, more like the 2 `bzip2` case. This is because as multiple instances run together, the lower IO and memory usage of the `povray` benchmark add up and start to become an issue. As the graphs show, the `smart` algorithm performs solidly well in all three cases, in contrast to `nice`, which falls near the baseline in all cases, and "allnice", which consistently lies below `smart`. Additionally, we note that in these cases `smart` is able to achieve its stronger protection of the foreground without penalizing the background more than "allnice".

Realistically, no systems administrator would attempt to run this level of background demand on a server whose response time mattered. However, `smart` clearly handles this kind of poor judgment better than the built-in priority mechanisms do.

Observation 4: The `smart` implementation is robust in the face of unreasonably heavy amounts of attempted background work.

Finally, we examine the behavior of the `smart` algorithm across time in the graphs shown in Figure 9. Here, the top graph shows the lighter background load of 2 `hmmer` instances and the bottom graph shows the heavier 8 `hmmer` load. We can see that length of suspend periods has increased when the background load is heavier and that this is providing the necessary extra protection to the foreground task.

Observation 5: The `smart` implementation correctly learns the behavior of the foreground task and permits the background tasks to run in a complementary manner.

To further evaluate and improve `smart`, we plan to experiment with less regular background work, especially less regular memory and IO access patterns. We also plan to work with multiple targets that can also provide throughput or completion deadline protection for background work. There may be more complex workload that can not be well-served by `smart`'s reliance on the average CPU utilization of the foreground task as the key threshold for scheduling decisions. In that case, we

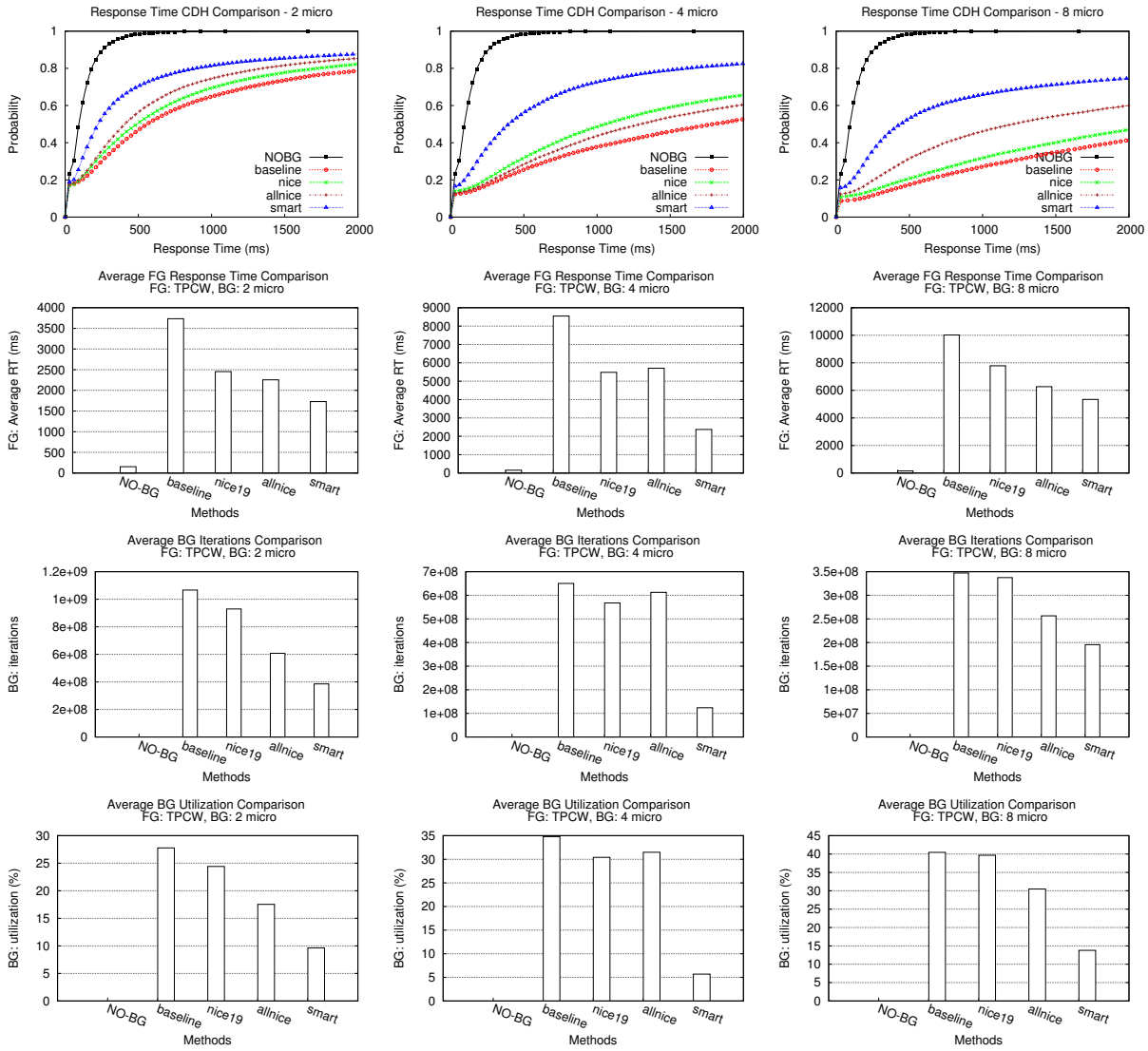

Fig. 6. Performance results for microbenchmark as background.

plan to add more scheduling parameters by capturing more sophisticated statistical information to make our scheduling framework even more intelligent and robust.

V. Related Work

In this paper we have presented an implemented approach to protect the response time of a high-priority task which has bursty behavior that cannot be matched to a predictable schedule but nevertheless can be statistically characterized in some useful ways. This is quite different from traditional work on real-time scheduling disciplines, which is heavily focused on strictly or semi-strictly predictable periodic tasks, such as media players, and which generally require kernel modification, changes to application code in order to take advantage of the system, and keeping track of specific deadline information for every task [6], [7], [8].

We also differ from works such as [4], which look to provide kernel support for differentiating Quality of Service for individual customers. We are focused on preventing background tasks on the server from interfering with any response-time-sensitive tasks rather than on separating tasks requiring different QoS. Indeed, our course-grained predictive approach to background task management could be combined with QoS differentiation schemes by using different thresholds to protect higher QoS processes more than lower.

Recent scheduling research has often focused on the particular problems of scheduling jobs on multicore machines and computing clusters [15], [16]. When priority schedulers are considered, it is generally with the intention of improving their fairness or maintaining fairness when adapting a scheduler to more complex circumstances [16], [17]. The individual characteristics of particular tasks are often taken into account for scheduling purposes, for instance to save energy during periods of low utilization [18] or to spread out intensive tasks to prevent thermal damage to a machine [19]. In some cases the non-linear interaction of different co-located jobs is taken into

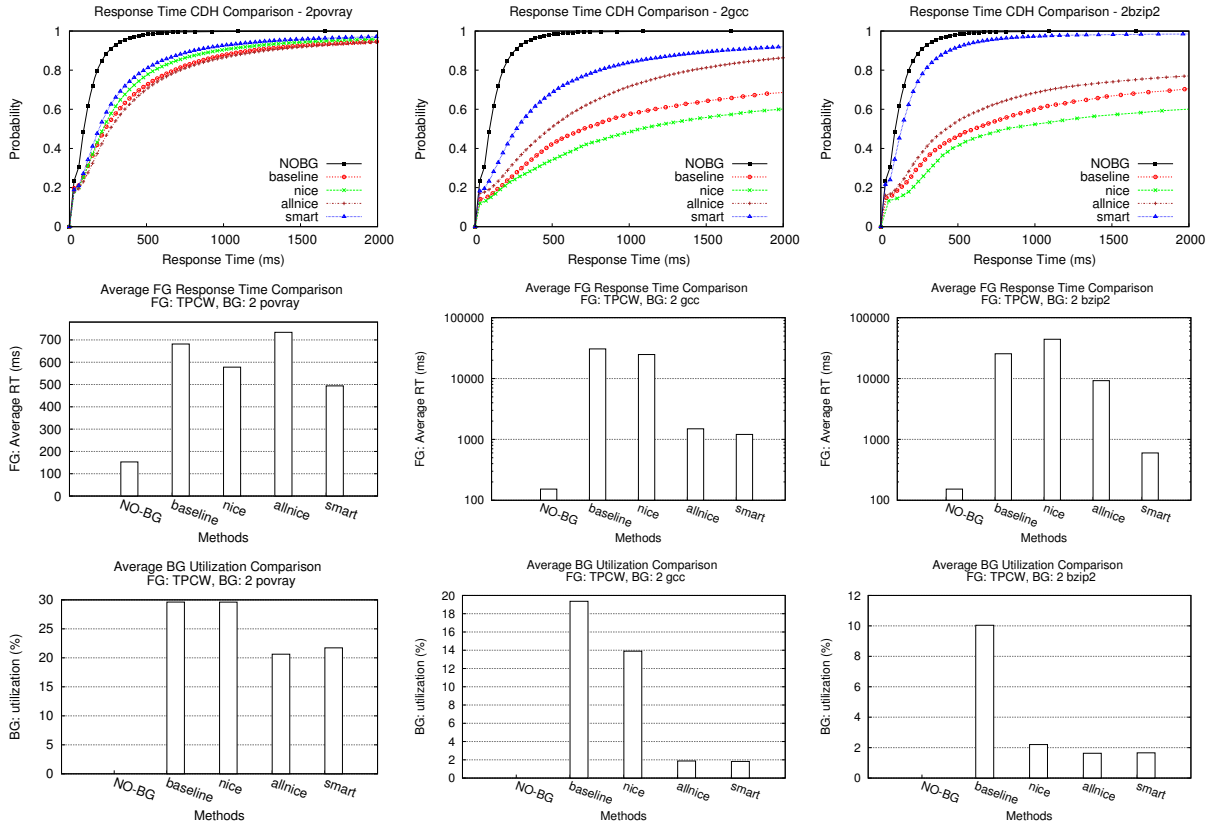

Fig. 7. Performance results for 2 SPEC benchmarks as background tasks.

account [20]. In this work, we look to use as much of the spare capacity as possible for time-insensitive background tasks, as in the case of a server handling the continuous and bursty workload of foreground user traffic while also intending to perform replication, integrity checking, data analysis, or other work [21], [22].

Virtual machines can also be used to isolate high priority tasks [5]. However, this approach requires significant overhead to manage, monitor, and adjust resource allocation to the different virtual machines. Our approach is less expensive, simpler to use, and does not require the deployment of any additional software.

Priority-based schedulers are wide-spread and intuitive, but it is well-known that they cannot strongly protect a high-priority foreground task, especially in the case of numerous background tasks, because they never starve the background tasks [23], [24]. We have found that under certain circumstances, the behavior of Linux `nice` can be quite unpredictable, sometimes with worse foreground performance when background processes are given the lowest `nice` priority than when `nice` is not used at all.

There are a variety of approaches to address this problem in the literature, though differing in both approach and ultimate goals from our own. Cucinotta et. al. focus on meeting acceptable throughput for ”soft real-time” applications, specifically media streaming, which has a range of acceptable frame rates from ideal to tolerable for brief periods [25]. To do this, they take a signal processing approach to characterize the activity periodicity behavior of the blackbox legacy applications they are attempting to control, and use the results to budget resources for each application. Their implementation requires kernel modification and does not explicitly stop low priority background tasks in order to better protect foreground tasks, as ours does. Meehean et. al. propose a very flexible system which requires kernel modification and demonstrate a scenario similar to ours [26].

Other researchers have focused on the progress rate of applications to determine appropriate resource sharing between them [27], [28]. Ferguson et. al. describe a weighted fair-sharing system that uses the progress rate to effectively balance between jobs with specific deadlines of varying importance [28]. Douceur and Bolosky share our goal more clearly, identifying very low priority tasks that should not be allowed to impact the foreground task [27]. To determine whether the background task should be run or temporarily stopped, they monitor the progress rate of the background applications, assuming that when the progress rate falls below a particular threshold, it must be because of foreground process contention for shared resources. The background tasks are then stopped for a window of time, then tried again. Inspired by the TCP congestion control mechanism, the sleep window increases exponentially as resource contention is repeatedly observed. These approaches work well, but require a way to monitor the progress rate of background applications, by the application themselves reporting an application-specific measure during execution or by a test run paired with detailed information

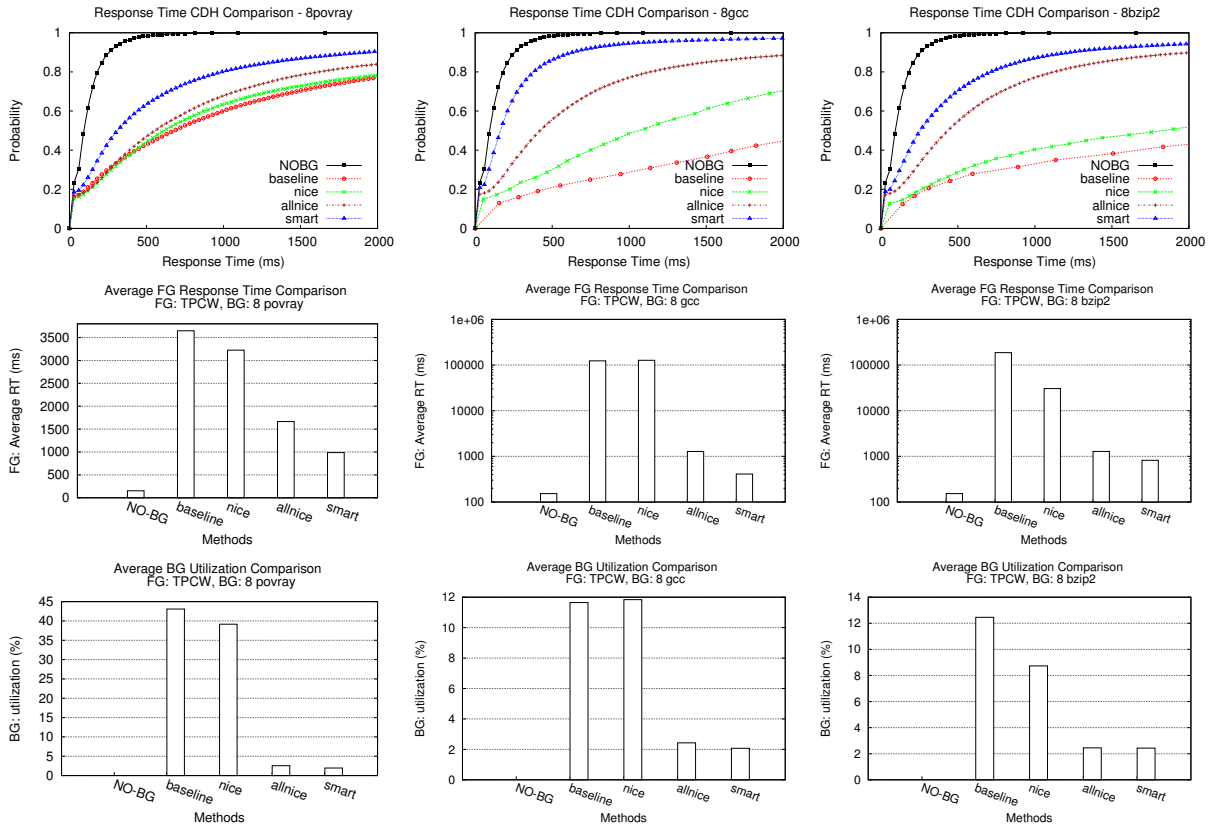

Fig. 8. Performance results for 8 SPEC benchmarks as background tasks.

Fig. 9. TPC-W utilization and background status across time under the `smart` algorithm.

about the job.

Also closely related to our work, Abe et. al. consider distributed computing projects like SETI@home, which allow individuals to donate computing time to scientific calculations when their computer is otherwise idle [23]. Similar to our work, Abe et. al. find built-in priority scheduling insufficient to protect foreground performance and choose to turn off background processing when the system detects resource contention with foreground processes. Similar to systems focused on background task progress rate mentioned above, Abe et. al. monitor the background process to detect this contention and apply an exponential back off to reduce the impact on the foreground. Instead of attempting to measure the progress of the background tasks, however, they monitor the share of resources given to the background process. If the share drops, they assume that the foreground processes are now demanding more resources and could benefit from the background dropping altogether. In contrast, we focus on the behavior of the foreground task, looking for the best periods in which to perform background work. We also include a learning phase, in which we characterize the statistical distribution of the fore-

ground traffic's busy periods to determine the optimal periods to suspend the background job execution. This additionally allows our monitoring intervals to be much longer than most of other approaches, which reduces the overhead of `smart` and allows it to function entirely in the user space without special kernel modification.

To sum up, `smart` scheduling differs from all the above work in that it does not require changing the kernel or depend on complex software. It does not require making changes to the foreground application or its processes, it can be even deployed without interrupting the current services. Therefore, it is lightweight, portable and flexible.

VI. Conclusions

Scheduling high-priority jobs together with low-priority ones in off-the-shelf systems using `nice` and `ionice`, a standard non-proprietary software that is available with any Unix-based distribution, can be erratic, often resulting in severe performance inconsistencies, especially when the resource consumption of the high priority job is not constant across time and when all jobs compete for more resources than just the CPU. To remedy this, we present `smart`, a new algorithm for improving the performance inconsistencies of `nice` and `ionice`, which bases its operation on restricting the resource consumption of background tasks when necessary, such that service differentiation across jobs with different priorities is consistent. `smart` is based on online monitoring of the CPU consumption of the foreground job and on observing differences between the average CPU utilization of the high-priority job versus the utilization observed within a short time window. Based on these differences, best-effort jobs are suspended and restarted.

`smart` is effective for high-priority workloads that are resource-hungry with a periodic or bursty pattern, but sometimes at the detriment of the low priority jobs, i.e., the throughput or completion times of low priority jobs is not part of the algorithm. Our on-going work focuses on addressing this by providing multiple targets in terms of different performance metrics (e.g., throughput versus response times versus completion deadlines) and different jobs (e.g., both high and low priority ones). Furthermore, a more intelligent and robust algorithm will be developed by capturing and taking advantage of more sophisticated statistical information.

Acknowledgments

This work has been supported by NSF grants CCF-0937925 and CCF-1218758.

References

[1] A. Rabkin and R. Katz, "Chukwa: a system for reliable large-scale log collection," in *Proceedings of the 24th international conference on Large installation system administration*, ser. LISA'10. Berkeley, CA, USA: USENIX Association, 2010, pp. 1–15. [Online]. Available: http://dl.acm.org/citation.cfm?id=1924976.1924994

[2] "The open compute project," http://www.opencompute.org/, 2011.

[3] M. Stonebraker, "The case for shared nothing," *IEEE Database Eng. Bull.*, vol. 9, no. 1, pp. 4–9, 1986.

[4] R. Zhang, T. Abdelzaher, and J. Stankovic, "Kernel support for open qos-aware computing," in *Real-Time and Embedded Technology and Applications Symposium, 2003. Proceedings. The 9th IEEE*, 2003, pp. 96–105.

[5] P. Padala, K.-Y. Hou, K. G. Shin, X. Zhu, M. Uysal, Z. Wang, S. Singhal, and A. Merchant, "Automated control of multiple virtualized resources," in *EuroSys*, 2009, pp. 13–26.

[6] L. Sha, T. F. Abdelzaher, K.-E. rzn, A. Cervin, T. P. Baker, A. Burns, G. C. Buttazzo, M. Caccamo, J. P. Lehoczky, and A. K. Mok, "Real time scheduling theory: A historical perspective." 2004, pp. 101–155.

[7] J. Nieh and M. S. Lam, "A smart scheduler for multimedia applications." 2003, pp. 117–163.

[8] C. W. Mercer, S. Savage, and H. Tokuda, "Processor capacity reserves: Operating system support for multimedia applications." in *ICMCS*, 1994, pp. 90–99.

[9] J. Hwang and T. Wood, "Adaptive dynamic priority scheduling for virtual desktop infrastructures," in *IWQoS*, 2012, pp. 1–9.

[10] "TPC-W," http://www.tpc.org/tpcw/.

[11] E. Cecchet, A. Ch, S. Elnikety, J. Marguerite, and W. Zwaenepoel, "A comparison of software architectures for e-business applications," In Proc. of 4th Middleware Conference, Rio de, Tech. Rep., 2002.

[12] J. L. Henning, "Spec cpu2006 benchmark descriptions," *SIGARCH Comput. Archit. News*, vol. 34, no. 4, pp. 1–17, Sep. 2006. [Online]. Available: http://doi.acm.org/10.1145/1186736.1186737

[13] Q. Wang, Y. Kanemasa, M. Kawaba, and C. Pu, "When average is not average: large response time fluctuations in n-tier systems," in *Proceedings of the 9th international conference on Autonomic computing*, ser. ICAC '12. New York, NY, USA: ACM, 2012, pp. 33–42. [Online]. Available: http://doi.acm.org/10.1145/2371536.2371544

[14] T. Bezenek, T. Cain, R. Dickson, T. Heil, M. Martin, C. McCurdy, R. Rajwar, E. Weglarz, C. Zilles, and M. Lipasti, "Java tpc-w implementation distribution," http://pharm/ece.wisc.edu/tpcw.shtml, 2011.

[15] M. Isard, V. Prabhakaran, J. Currey, U. Wieder, K. Talwar, A. Goldberg, and A. Goldberg, "Quincy: fair scheduling for distributed computing clusters." in *SOSP*, 2009, pp. 261–276.

[16] M. Zaharia, D. Borthakur, J. S. Sarma, K. Elmeleegy, S. Shenker, I. Stoica, and I. Stoica, "Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling." in *EuroSys*, 2010, pp. 265–278.

[17] C. Krasic, M. Saubhasik, A. Sinha, A. Goel, and A. Goel, "Fair and timely scheduling via cooperative polling." in *EuroSys*, 2009, pp. 103–116.

[18] E. Thereska, A. Donnelly, D. Narayanan, and D. Narayanan, "Sierra: practical power-proportionality for data center storage." in *EuroSys*, 2011, pp. 169–182.

[19] A. K. Coskun, R. D. Strong, D. M. Tullsen, T. S. Rosing, and T. S. Rosing, "Evaluating the impact of job scheduling and power management on processor lifetime for chip multiprocessors." in *SIGMETRICS/Performance*, 2009, pp. 169–180.

[20] S.-H. Lim, J.-S. Huh, Y. Kim, G. M. Shipman, C. R. Das, and C. R. Das, "D-factor: a quantitative model of application slow-down in multi-resource shared systems." in *SIGMETRICS*, 2012, pp. 271–282.

[21] N. Mi, A. Riska, X. Li, E. Smirni, and E. Riedel, "Restrained utilization of idleness for transparent scheduling of background tasks," in *Proceedings of the Eleventh International Joint Conference on Measurement and Modeling of Computer Systems, SIGMETRICS/Performance*, 2009, pp. 205–216.

[22] F. Yan, A. Riska, and E. Smirni, "Fast eventual consistency with performance guarantees for distributed storage," in *ICDCS Workshops*, 2012, pp. 23–28.

[23] Y. Abe, H. Yamada, K. Kono, and K. Kono, "Enforcing appropriate process execution for exploiting idle resources from outside operating systems." in *EuroSys*, 2008, pp. 27–40.

[24] L. Eggert and J. D. Touch, "Idletime scheduling with preemption intervals," in *SOSP*, 2005, pp. 249–262.

[25] T. Cucinotta, F. Checconi, L. Abeni, L. Palopoli, and L. Palopoli, "Self-tuning schedulers for legacy real-time applications." in *EuroSys*, 2010, pp. 55–68.

[26] J. Meehean, A. Arpaci-Dusseau, R. Arpaci-Dusseau, and M. Livny, "CPU Futures: Scheduler support for application management of cpu contention," Technical Report at: http://research.cs.wisc.edu/techreports/2010/TR1684.pdf, 2011.

[27] J. R. Douceur, W. J. Bolosky, and W. J. Bolosky, "Progress-based regulation of low-importance processes." in *SOSP*, 1999, pp. 247–260.

[28] A. D. Ferguson, P. Bodk, S. Kandula, E. Boutin, R. Fonseca, and R. Fonseca, "Jockey: guaranteed job latency in data parallel clusters." in *EuroSys*, 2012, pp. 99–112.