Agile Middleware for Scheduling: Meeting Competing Performance Requirements of Diverse Tasks

Feng Yan
College of William and Mary
Williamsburg
VA, USA
fyan@cs.wm.edu

Shannon Hughes
College of William and Mary
Williamsburg
VA, USA
srhughes@cs.wm.edu

Alma Riska
EMC Corporation
Cambridge
MA, USA
alma.riska@emc.com

Evgenia Smirni
College of William and Mary
Williamsburg
VA, USA
esmirni@cs.wm.edu

ABSTRACT

As the need for scaled-out systems increases, it is paramount to architect them as large distributed systems consisting of off-the-shelf basic computing components known as compute or data nodes. These nodes are expected to handle their work independently, and often utilize off-the-shelf management tools, like those offered by Linux, to differentiate priorities of tasks. While prioritization of background tasks in server nodes takes center stage in scaled-out systems, with many tasks associated with salient features such as eventual consistency, data analytics, and garbage collection, the standard Linux tools such as `nice` and `ionice` fail to adapt to the dynamic behavior of high priority tasks in order to achieve the best trade-off between protecting the performance of high priority workload and completing as much low priority work as possible. In this paper, we provide a solution by proposing a priority scheduling middleware that employs different policies to schedule background tasks based on the instantaneous resource requirements of the high priority applications running on the server node. The selection of policies is based on off-line and on-line learning of the high priority workload characteristics and the imposed performance impact due to low priority work. In effect, this middleware uses a *hybrid* approach to scheduling rather than a monolithic policy. We prototype and evaluate it via measurements on a test-bed and show that this scheduling middleware is robust as it effectively and autonomically changes the relative priorities between high and low priority tasks, consistently meeting their competing performance targets.

Categories and Subject Descriptors

C.4 [**Computer Systems Organization**]: Performance of Systems

General Terms

Performance, Algorithms

Keywords

priority scheduling; performance guarantee; background throughput; learning performance patterns; decision map; Linux priority utilities; workload characterization

1. INTRODUCTION

Scaling of web services is mostly achieved by deploying distributed systems in large data centers or even across them. Traditional computer systems, particularly those supporting enterprise applications, do not scale well, especially with regard to cost. To mitigate cost at a large scale, the industry is increasingly turning to commodity hardware and open-source software to accomplish large-scale services and computation. In order to scale-out and still operate effectively, the building blocks are off-the-shelf server nodes that operate mostly independently, while exchanging messages with other participating nodes [20]. This goal, exemplified by the *Open Compute* [2] initiative which has been widely adopted by the broader tech community, is to keep down the cost of systems that host big data and provide large scale analytics and other important web services.

One of the salient characteristics of the large distributed systems hosting a wide range of web services is supporting a wide range of features, including eventual consistency of data, data replication, garbage collection, and log data analysis, that run asynchronously in the background, at the level of the individual server node. The goal is to serve user workload as fast as possible and handle most of the management tasks *only when system resources are moderately utilized*. To illustrate the existence of opportunities for effective scheduling of background tasks in real systems, we illustrate in Figure 1 the arrival intensity of requests to store new data or read existing ones in one of the nodes of a large scale web service over a three day period. The strong daily

pattern in the arrival intensity allows the system to schedule other important but less time sensitive tasks, e.g., garbage collection, during periods of low user activity, ensuring that these tasks do not affect the user quality of experience.
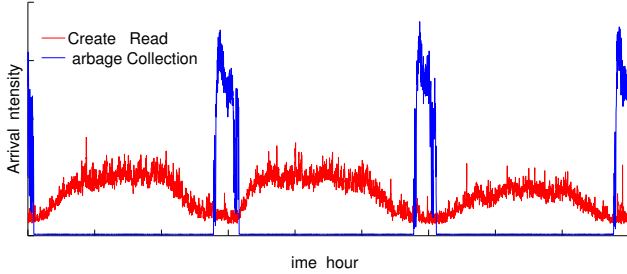

Figure 1: Overtime plot of arrival intensity for a large scale web service .

Here, we develop scheduling middleware that builds upon standard scheduling prioritization tools that are available in any Linux distribution, which often is the operating system of choice in the individual nodes of scaled-out systems. Standard distributions provide monolithic tools for priority scheduling but these are usually not reactive to changing workload conditions as those depicted in Figure 1. The scheduling middleware that we propose is based on effectively launching `nice` and `ionice`, the most common prioritization tools, with the appropriate priority levels that best match the existing system conditions. Furthermore, these priority levels are *continuously adjusted* throughout the lifetime of the system to control relative priorities between the user (or foreground) traffic and the background system features in order to guarantee quality of service targets for foreground work while maximizing completion levels of background features.

While prioritization features have been proposed at the kernel level [26, 18] or at the application level [19, 17, 15], our focus is to provide middleware that is built upon standard tools that are available in any Linux distribution and most importantly operate in user-space. By utilizing `nice` and `ionice` as building blocks, we ensure that at fine time scales (i.e., microseconds) there is correct differentiation of the processes based on their priorities. At coarse time scales (i.e., minutes), we control and manage these priorities (i.e., via `renice` and other utilities) to ensure that foreground performance is protected and background work is completed as efficiently as possible. The middleware that we propose is based on several standard monolithic scheduling policies (e.g., `nice` and `ionice`) but also on `smart`, a new (but still monolithic) mechanism that suspends background work briefly when foreground load spikes [23]. During the lifetime of the execution of the background work, the proposed middleware switches among the various basic policies as considered best fit.

Using the web-driven TPC-W benchmark [1, 5] as a representative foreground application, we experiment with a range of background tasks, as defined by controllable micro benchmarks. Our extensive set of experiments shows that we can effectively utilize system resources by scheduling background jobs with the *best* monolithic policy depending on resource availability, maintaining an overall uniform utilization of the system. We stress that the selection of the best

policy is left to the middleware and this can change during the course of the execution.

The rest of the paper is organized as follows. In Section 2, we provide results from characterizing the behavior of `nice` under several scenarios. Section 3 develops our new prioritization scheme which determines the priority of the background tasks. The new framework is evaluated via extended experiments in Section 4. We discuss related work in Section 5. We conclude the paper and summarize future work in Section 6.

2. PRELIMINARIES

In this section, we first present an overview of the available off-the-shelf scheduling tools for priority scheduling and continue with an overview of resource demands across time for a typical workload to illustrate how background scheduling can become truly opportunistic. Finally, we show with evidence that a single background scheduling policy cannot be effective under all circumstances, which further corroborates the need for agile middleware that continuously adjusts priority scheduling parameters in a transparent and autonomic way.

2.1 Prioritizing Background Work

Proprietary systems often have their own priority scheduling algorithms that allow them to maintain performance of user workload while other lower priority jobs are running in the background. For systems built of off-the-shelf Unix components, the readily available tools for priority scheduling 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). A process invoked with nice 0 or without nice command runs with the default (i.e., normal) priority. `nice` determines 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 is implementation dependent and varies between Unix/Linux distributions. The mechanism is generally simple to use and depends on fine-grained CPU consumption. A user can change the priority of a process via `Eunice`.

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 idle. Adding `ionice` can help boost `nice`'s performance in cases of memory-intensive background tasks.

A user may select to invoke both `nice` and `ionice` together. Combining `nice 19` with `ionice 3` gives the lowest priority setting for both resources. We label this combination as **allnice** and it represents the most straightforward way for the background work to minimally effect the performance of foreground work using commodity utilities.

In [23], a scheduling policy named `smart` is developed that focuses on adapting background job scheduling to foreground work with demands that are variable across time. The basic premise is to observe and effectively *predict* periods of low and high utilization of the foreground work and

launch or suspend background work based on monitoring system utilization levels. Suspending and resuming utilizes system resources by scheduling background work only when resources are lightly to moderately utilized by high priority processes. Additionally, it better isolates the foreground performance than `nice` or `allnice`, as well as doing so more consistently than either of the off-the-shelf options.

In this paper, we develop a middleware that utilizes `smart` in [23] as well as the Linux priority scheduling tools `nice` and `ionice` and further enhances their capabilities by defining a set of policies which are automatically invoked within the same application run. The policies that are automatically selected by the middleware are the following:

- **nice 0**: the background work and the foreground application are running at the same priority for both CPU and IO resources.

- **allnice**: the background work runs at the lowest priority but it is never suspended, i.e., it is executed using `nice 19` with `ionice 3`.

- **smart+**: the background work is suspended briefly if load spikes using `smart` [23]. Once load returns back to its previous level, **allnice** is used here, unlike to the policy in [23].

- **FGonly**: the background work is suspended completely if high load for an extended period (i.e., at the hour-level granularity) is detected.

Additional policies can be added according to the specific system and application scenarios. In general, more scheduling policies give finer control, but may also result in more overhead. Intermediate policies with different `nice` parameters can be used, as well as more policies between the two extremes of **nice 0** and **FGonly**. For ease of presentation and with no loss of generality, we focus here on the four policies outlined above. More details are given in Section 3.

2.2 Scheduling Background Work: Perils and Opportunities

To illustrate the ample opportunities and perils of background scheduling, we show in Figure 2 the CPU utilization and response time as a function of elapsed time for TPC-W [1], a classic multi-tiered benchmark[1] that has significant variability across time in its CPU and memory demands [22]. The figure illustrates three scenarios: one with only 10 emulated browsers (EBs) (top graph), one with 40 emulated browsers (middle graph), and one with 70 emulated browsers (bottom graph). The figure clearly shows many opportunities to schedule background jobs when there are only 10 emulated browsers: the CPU utilization is consistently low, with the exception of a few short time periods. Similarly, average response times are low across the entire experiment. The middle graph shows a different situation: with 40 EBs several bursts of short but high CPU activity that are usually clustered together, interspersed with periods of low CPU usage. The average user response time follows closely the CPU usage patterns. The bottom graph, where there are 70 EBs, shows longer periods of high utilization intermixed with periods of low utilization. The figure illustrates that there are

---

[1] For the exact description of the experimental and measurement setting see Section 4.

plenty of opportunities to schedule background tasks when there are only 10 EBs, but higher load situations require more care, lest background work is scheduled during periods of high utilization and TPC-W performance is compromised.

2.3 Monolithic Background Scheduling

Figure 3 illustrates a first proof-of-concept of the relative advantages and disadvantages of scheduling background jobs using **nice 0**, **allnice**, **smart+**, and **FGonly**. The last policy gives the norm of the ideal response time. The background work that we launch here is explained in detail in Section 4. The figure illustrates the cumulative distribution histogram (CDH) of response times for TPC-W (first row) and the throughput of background jobs (second row), presented as the number of completed iterations. The CDH figures clearly illustrate that the ranking of the various policies with respect to foreground performance are consistent for 10, 40, and 70 EBs and reflect how conservatively the background work is scheduled, ditto for the respective amount of completed background work. Yet, if there is a certain service level objective, e.g., if the 80th percentile of response time needs to be less than 600 ms, then background scheduling can be tuned to be more or less aggressive, such that it takes into account the load in the system as expressed by the number of EBs is able to guarantee better background throughput. If the system operates with 10 EBs, then **nice 0** is sufficient for performance and maximizes the completed iterations but if the system operates with 40 EBs then **allnice** can offer performance guarantees while keeping iterations at a maximum. When EBs rise to 70, then **any** background scheduling must be stopped. The figure clearly shows that effective background scheduling needs to be agile and hybrid, i.e., *continuously change its priority parameters* (e.g., switch from **nice 0** to **allnice** to **smart+** to **FGonly**) depending on the system operating conditions. In the following section we define how to develop and launch such middleware.

3. METHODOLOGY

The basic premise of the proposed middleware is that if load from the high priority (or foreground) application is light, then running background tasks with the same priority should not violate the foreground performance target. As load from the foreground application increases, the priority of background work should decrease. If the system foreground load is high then the background should be suspended until the high load period passes. We aim to consistently meet the system's foreground performance target while serving as much background work as possible. To achieve this goal, we learn the corresponding performance for different foreground load levels and monitor the latter to decide at what priority (if at all) to schedule background tasks.

3.1 Foreground Load Levels Relative to the Target

Load levels are defined relative to the foreground performance target, which, without loss of generality, we define as the percentile of requests whose response time is less than a target value (e.g., 80% of foreground requests are served in less than 600 ms). The system then is said to be under high load if it closely meets the target. If the target is violated, then the system is in overload. If the load results in better performance than the target, then we consider the load to
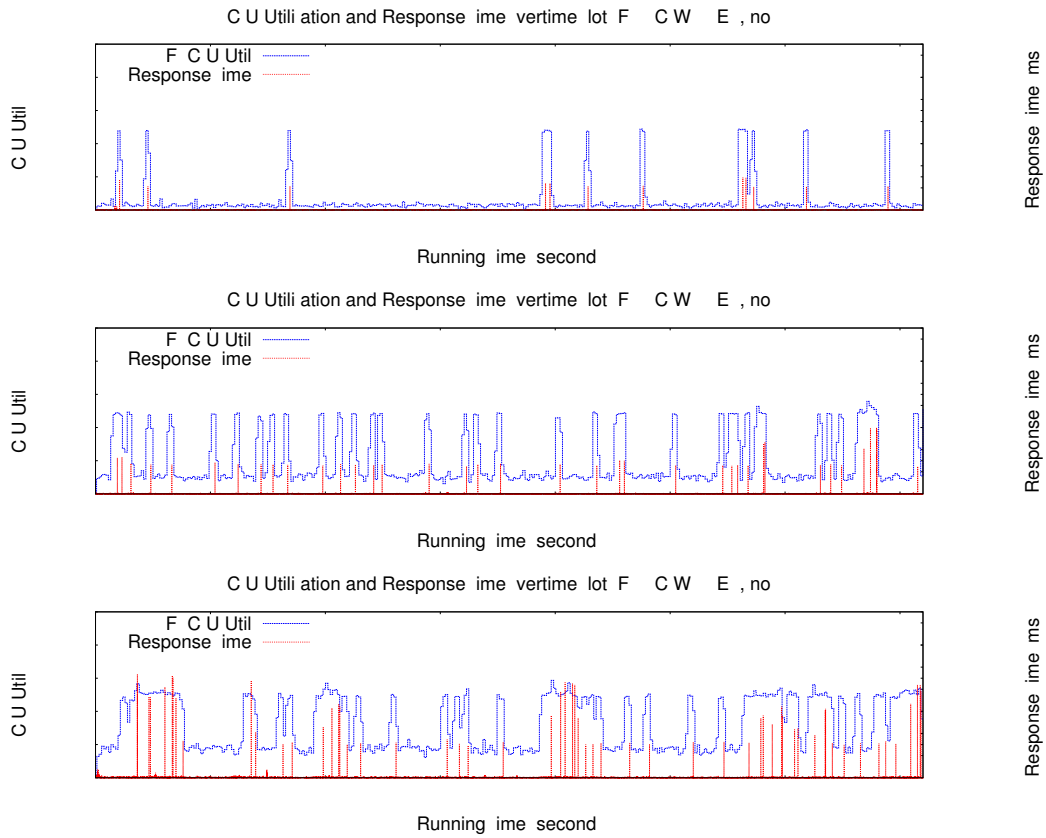
Figure 2: Overtime comparison of CPU utilization and average response times for 10, 40, and 70 emulated browsers. The duration of of this experiment is one hour.
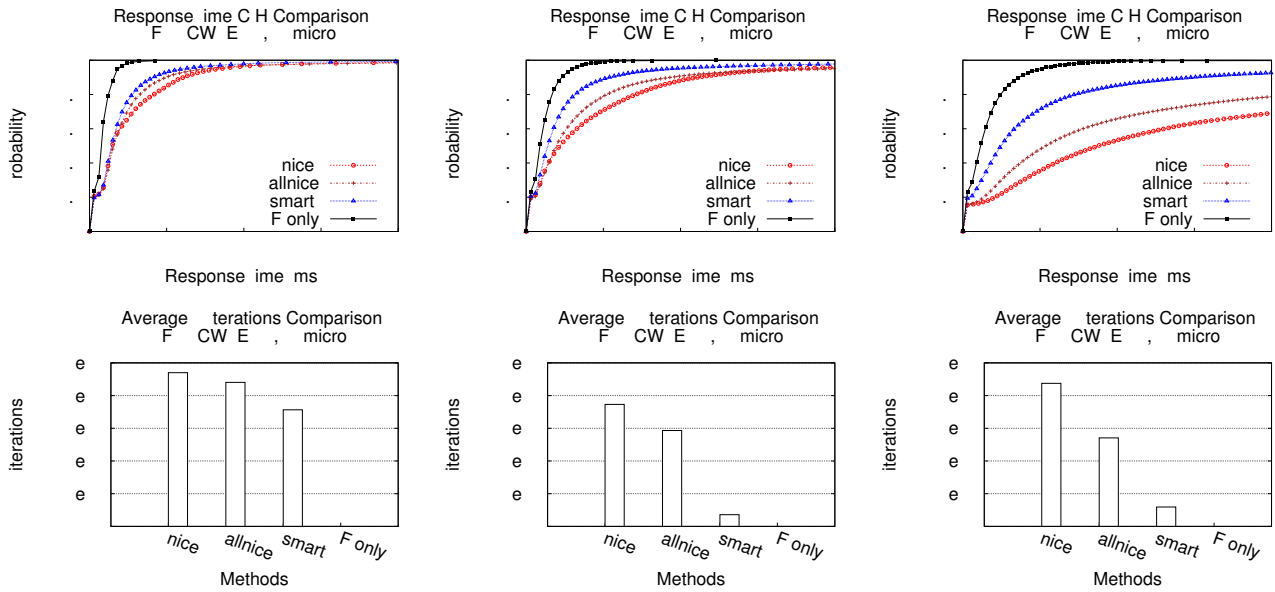

Figure 3: Performance results for TPC-W (CDH of response times) and background work completed (measured in number of iterations).

be light to moderate and background work can be scheduled without violating the target. The lighter the load, the higher the priority of the background work.

In Figure 4, we plot the cumulative distribution histogram (CDH) of TPC-W response times when load varies from light to heavy. In TPC-W the load is measured by the number of emulated browsers - EBs - (which corresponds to the number of network connections). In general, the load of a web service (which is the application type of interest given our focus on the individual nodes of a scaled-out system) can be measured similarly, although other metrics of load can be trivially defined and applied to our methodology.
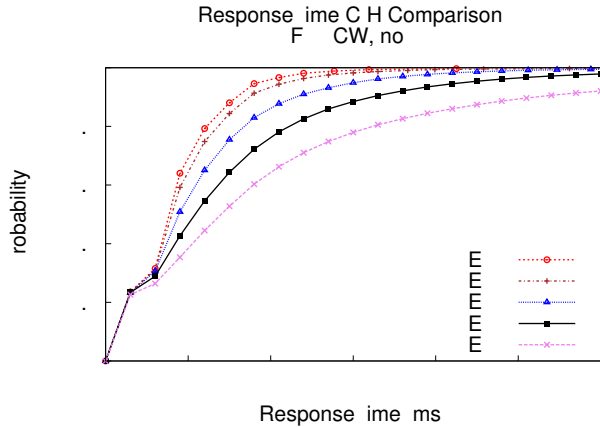

Figure 4: CDH of response times for different system load (EBs).

For a given target, we define the "high load" level based on the measurements captured in Figure 4. For example, the target of 80th percentile being at most 600 ms would result in "high-load" being 80 EBs, because it is the highest load level meeting this target. If the foreground performance target for the 80th percentile is to be at most 300 ms, then 70 EBs would be the "high load" in the system, while a load of 80 EBs would put the system into overload. For a target of 300 ms the system should serve background work alongside foreground only if the foreground load is less than 70 EBs.

The measurement data present in Figure 4 can be collected off-line in a test environment or it can be collected in the system as it comes on-line and kept up-to-date over time. Collecting such data should be possible with minimal effort, since the systems we are focusing on are provided by the general Linux distribution with an array of monitoring and logging tools.

3.2 Priority Policy Decision

The proposed middleware requires identifying the relation between current load and performance target for the foreground application (as described in Subsection 3.1), in order to identify the availability of resources to execute background tasks and set correctly the relative priority of the background tasks. As a result, similarly to the learning described in Subsection 3.1, we learn the foreground performance with a single representative background task (see more details in Section 4) treated with one of the four priority policies defined in Subsection 2.1. We again generate the distribution of foreground response times. For example, for each of the evaluated TPC-W loads and **nice 0**, we generate the same set of response time distributions as captured in Figure 4.

We learn foreground performance behavior through a number of representative cases. The background tasks that we use for *training* run concurrently with TPC-W are described in Section 4 and can be tuned to demand more or less CPU and memory resources. Specifically, during learning, we measure the system under the foreground application plus heavy background load, i.e., demanding more than 100% CPU utilization and memory, so that the impact on foreground performance would hold for *any* background task that may be served in the system. Because of these choices during the learning period, we consider the measurements conducted as a baseline that can be used reliably to guide our decision on the priority policy for a given foreground load (and its performance target) and *any* background task. Results shown in Section 4 support these choices. Our reliance on fine-grained priority scheduling done by `nice` and `ionice` adds to the robustness of our decisions.

To help visualize the data we collect during our learning process, as well as to clarify our decision-making process with regard to dynamically changing background priorities, we plot the collected data, i.e., the distribution of response times for different load levels and priority policies, as stacked bars, see Figure 5. The x-axis of Figure 5 consists of all possible (*system load, priority policy*) pairs. The y-axis in Figure 5 represents the response time percentiles of the foreground requests, measured in milliseconds. The different colors used in each bar mark a specific, i.e., 50, 70, 80, and 90th, percentile of the response time distribution for a specific pair.

The data structure visualized in Figure 5 is paired with the foreground performance target which we illustrate with a horizontal line which represents the expected response time percentile. In this figure we have marked performance targets for the 80th percentile of response times to be equal to or less than 600ms. In this case, more than 70 EBs is considered "high load", since the target is met under the **FG-only** policy only. As the foreground load decreases, the 80th percentile of foreground response times is met also by several priority policies that serve background work. For example, for 50 EBs, the 80th percentile of foreground response time is less than 600ms under the **smart+** policy, while for 40 EBs the target is still met if we schedule background work via **allnice**. At 30 EBs or less that background work can be scheduled with the same priority as foreground work via **nice-0** without violating the target. The benefit of increasing the priority of background work (from **FG-only** to **nice-0**) as foreground load reduces is to serve more background work while ensuring that the foreground performance target is met. The map in Figure 5 is used by the scheduling middleware that we propose here as the decision making engine to automatically adjust priorities as foreground load conditions change over time.

A schematic view of information interchange in our priority scheduling hybrid middleware is provided in Figure 6. We reiterate that the learning is done in such a way that it can either be complete off-line or on-line as the system comes up and can be updated overtime with more observations. As we provide more details on our prototype in Section 4, we also highlight the standard Linux utilities that we use.
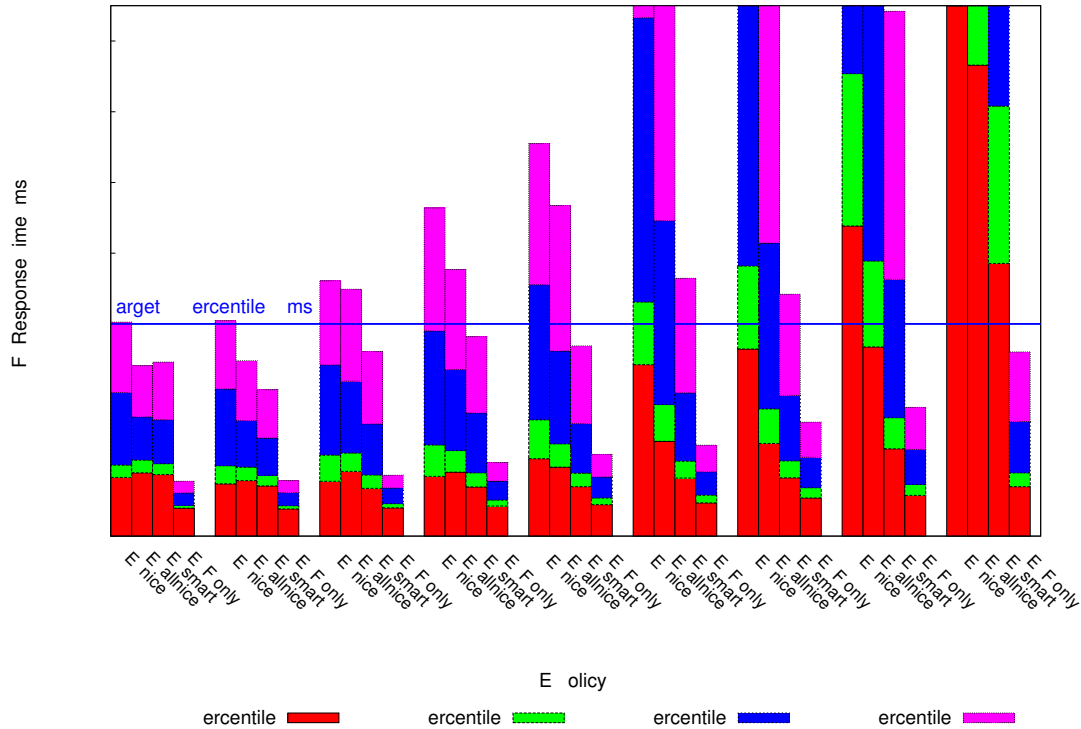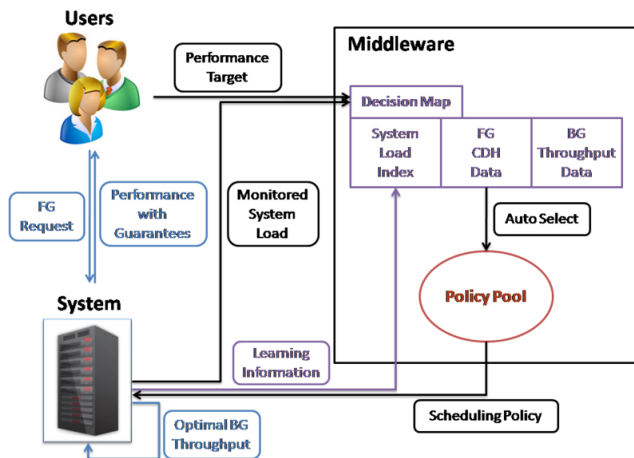
Figure 5: Decision map.

Figure 6: Schematic view of the middleware scheme.

4. EXPERIMENTAL EVALUATION

All experiments presented here 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. For background work we use our own micro benchmark in order to do control the experiments and ensure representative data with regard to learning.

We consider TPC-W to be a challenging workload, because it is characterized by variability in its resource de-

mands across time [5], as also shown in Figure 2. TPC-W is a web server and database performance benchmark [1] and in our prototype we use to drive the system the java distribution in [4]. We use tomcat as the application server and mysql as the database server. TPC-W provides a large number of parameters. We use the browsing mix on a 100000 items in the database.

We develop our own micro benchmark to use as background work, which is built upon the Isolation Benchmark Suite [13]. The micro-benchmark performs multiplications in a tight loop which is embedded in a larger one containing array initializations and file writes. The micro benchmark allows to experiment with a broader range of CPU, memory, and IO background demands. In the results reported in this section, we have used three different variations of the micro benchmark. In each of the three scenarios, four instances of the micro-benchmark are run concurrently, each consuming approximately 20% of memory capacity and some IO traffic. The micro benchmarks parameters are scaled to change the CPU demand across as shown below:

- micro1: consumes approximately 100% of the system's total CPU resource.

- micro2: consumes approximately 45% of the system's total CPU resource.

- micro3: consumes approximately 160% of the system's total CPU resource (i.e., uses almost both cores).

In order to provide a simple and easily portable implementation, our monitoring and scheduling algorithms are implemented entirely in user space, making use of readily available Linux commands (e.g., `pidstat` and `kill`). For

monitoring, we launch a shell script to call `pidstat` every 10 seconds and extract the CPU utilization for all running processes, classifying the results into three main categories: foreground (TPC-W related) processes, background (micro benchmark related) processes, and other system processes. The coarse granularity of these intervals differs from the fine-grained handling generally used in real-time scheduling algorithms in the literature, but we emphasize that we delegate the fine-grained decisions to `nice` and `ionice`.

To control the execution of background work, we use the STOP and CONT signals and pass them to process by the `kill` command to "pause" and "resume" the background tasks. 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. We stress all these native system tools make our method easy to deploy and with low overhead.

4.1 Results

Initially, we evaluate our hybrid scheduling middleware by running the TPC-W as the foreground task and four micros for a total of 100% additional CPU utilization (i.e., variant *micro1* above) as background tasks for different foreground performance targets. We choose two scenarios to present here. Scenario 1's performance target is the 80th percentile to be equal or smaller than 600 ms and Scenario 2's performance target is the 90th percentile to be equal or smaller than 650 ms. Based on the decision map of Figure 5 the scheduling strategy is devised and summarized in Table 1. The policy transition parameters in Table 1 are obtained from our off-line learning. As robustness of this learning approach is key to our evaluation, we run all our tests for 14 hours, during which the foreground load varies from 0 to 80 EBs, including load levels that were not used in learning. For those cases, the decision is done based on the next higher load tested.

We evaluate our hybrid priority scheduling middleware by comparing it with the monolithic scheduling methods for the same scenario. Our experiments are run 14 hours long to ensure enough instances of foreground workload changes occur to demonstrate the robustness of our hybrid middleware. We plot the results for Scenario 1 and Scenario 2 in Figures 7 and 8, respectively. In each figure, we plot the system load measured by both CPU utilization and number of EBs (see top plot). As load varies over time, so do the opportunities to schedule background work. For each hour, we report in the top plot of Figures 7 and 8 along the x-axis, the decision of our hybrid middleware based on the parameters devised in Table 1 and monitoring of foreground load levels. Figures 7 and 8 also plot the CDH and CCDF of the response times for the different monolithic policies and our hybrid middleware (see the second row).

As expected, the **FG-only** and **nice-0** achieve the best and worst foreground performance, respectively, because **FG-only** suspends background work while **nice-0** treats foreground and background work the same. The other two policies, **allnice** and **smart+** maintain better foreground performance at the cost of background throughput (bottom right plot in Figures 7 and 8). The bottom left plot in Figures 7 and 8 shows how hybrid meets the foreground performance target (see vertical line) while achieving highest background throughout among all policies that do meet the performance target (**FG-only**, **smart+**, and **hybrid**

for Scenario 1 and **FG-only** and **hybrid** for Scenario 2). (see right plot in the bottom row).

We also evaluate the resiliency of our hybrid middleware to the learning methodology. Recall that learning was done with background tasks adding up to 100% CPU utilization. We run the same 14 hours test for Scenario 1 and Scenario 2, but now the background work follows the variant micro2 (45% total CPU demand) and micro3 (total 160% CPU demand). For micro2 background workload, we show the respective results for Scenario 1 and Scenario 2 in Figures 9 and 10. For micro3 background workload, we show respective results for Scenario 1 and Scenario 2 in Figure 11 and Figure 12. The decisions on policy transitions are done according to Table 1 for both cases.

These experiments confirm that the hybrid middleware meets the foreground performance target under all these different combinations of foreground performance targets and background workload and that the learning approach is effective. The reason behind this is that the low priority workload is only scheduled during low system load periods, where the foreground impact is well controlled. In addition, we are always conservative by approximating the untrained foreground intensity to the higher nearest intensity entry in the decision map. Another critical aspect that contributes to the resiliency of our hybrid middleware is the fact that the scheduling policies used in our scheme are based on `nice` and `ionice`, which control priorities at very fine granularities.

5. RELATED WORK

There has been a large volume of related work, that can be roughly classified as scheduling that requires kernel modification, application modification, real time scheduling, or scheduling for quality of service. Yet, to the best of our knowledge, there is no mechanism that is available at the user space as the one proposed here, that relies on automatic usage of pause/resume of background execution as well as automatic usage of the various `nice` and `ionice` options, or `renice` thereof.

Traditional work on real-time scheduling relies on strictly or semi-strictly predictable periodic tasks, such as media players, and requires kernel modification, changes to application code in order to take advantage of the scheduling, and keeping track of specific deadline information for every task [19, 17, 15]. Cucinotta et. al. focus on meeting acceptable throughput for "soft real-time" applications, specifically media streaming [7]. 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 [14]. Our work does know rely on periodicity or any modifications to kernel or application code.

Scheduling that provides quality is service to individual customers has been developed in [26], 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 without again requiring any kernel modification. Indeed, the proposed mechanism to background task

Table 1: Scheduling Computed from Decision Map for different Scenarios.

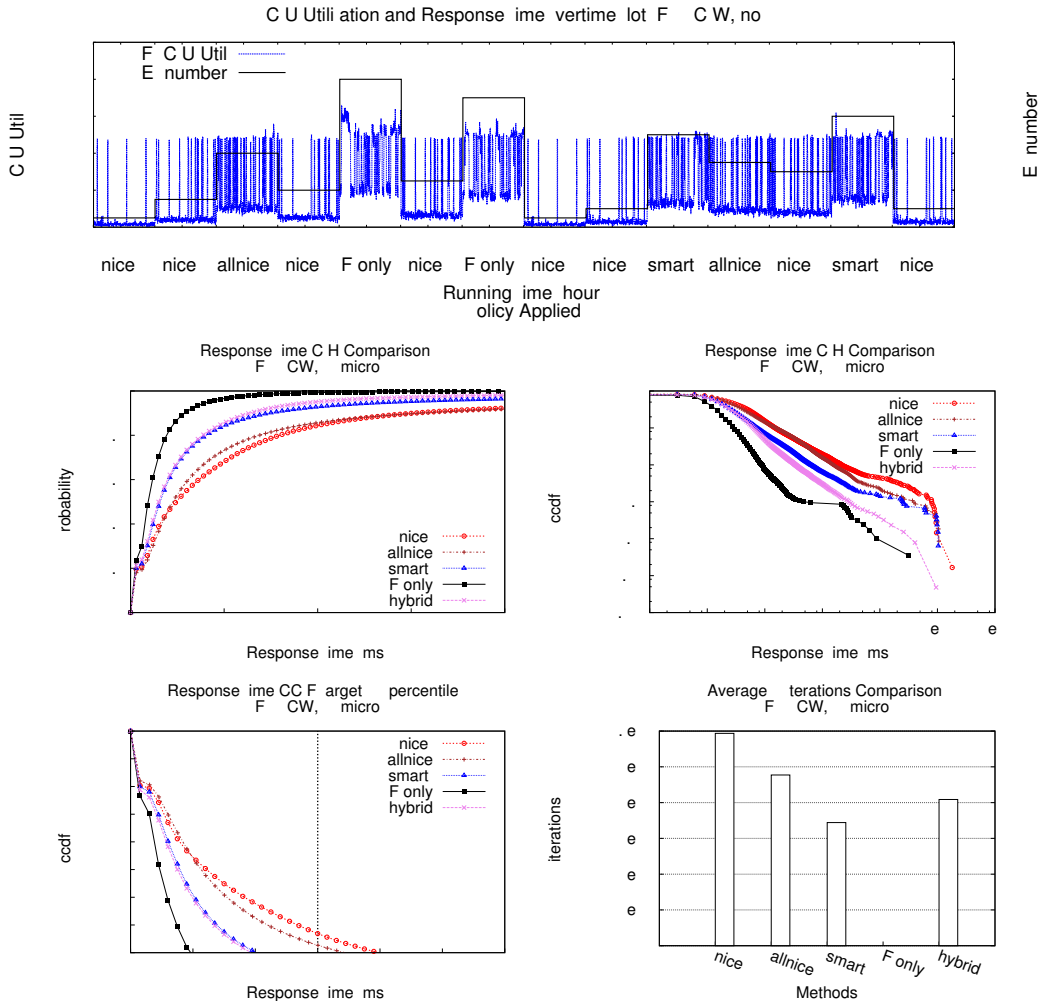| Scenario 1: Target: $Prt(80\%) <= 600ms$			Scenario 2: Target: $Prt(90\%) <= 650ms$		
System Load	EB Range	Policy Selection	System Load	EB Range	Policy Selection
low	0-30	nice0	low	0-10	nice0
medium	31-40	allnice	medium	11-19	allnice
high	41-60	smart+	high	20-40	smart+
extreme	61-80	FGonly	extreme	40-80	FGonly

Figure 7: Scenario 1 - BG: CPU total demand: 100% .

management could be combined with QoS differentiation schemes by using different thresholds to protect higher QoS processes.

Recent scheduling research has often focused on the particular problems of scheduling jobs on multicore machines and computing clusters [10, 25]. 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 [25, 11]. The individual characteristics of particular tasks are often taken into account for scheduling purposes, for instance to save energy during periods of low utilization [21] or to spread out intensive tasks to prevent thermal damage to a machine [6]. In some cases the non-linear interaction of different co-located jobs is taken into account [12]. 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 [16, 24].

Virtual machines (VMs) can also be used to isolate high priority tasks [18]. VM management is not straightforward either and requires significant overhead to manage, monitor, and adjust resource allocation. In contract to traditional VM managing solutions our approach is simpler to use as it does not require the deployment of any additional software. We provide a more precise sharing of resources since it adjusts based on percentage of total CPU usage and may
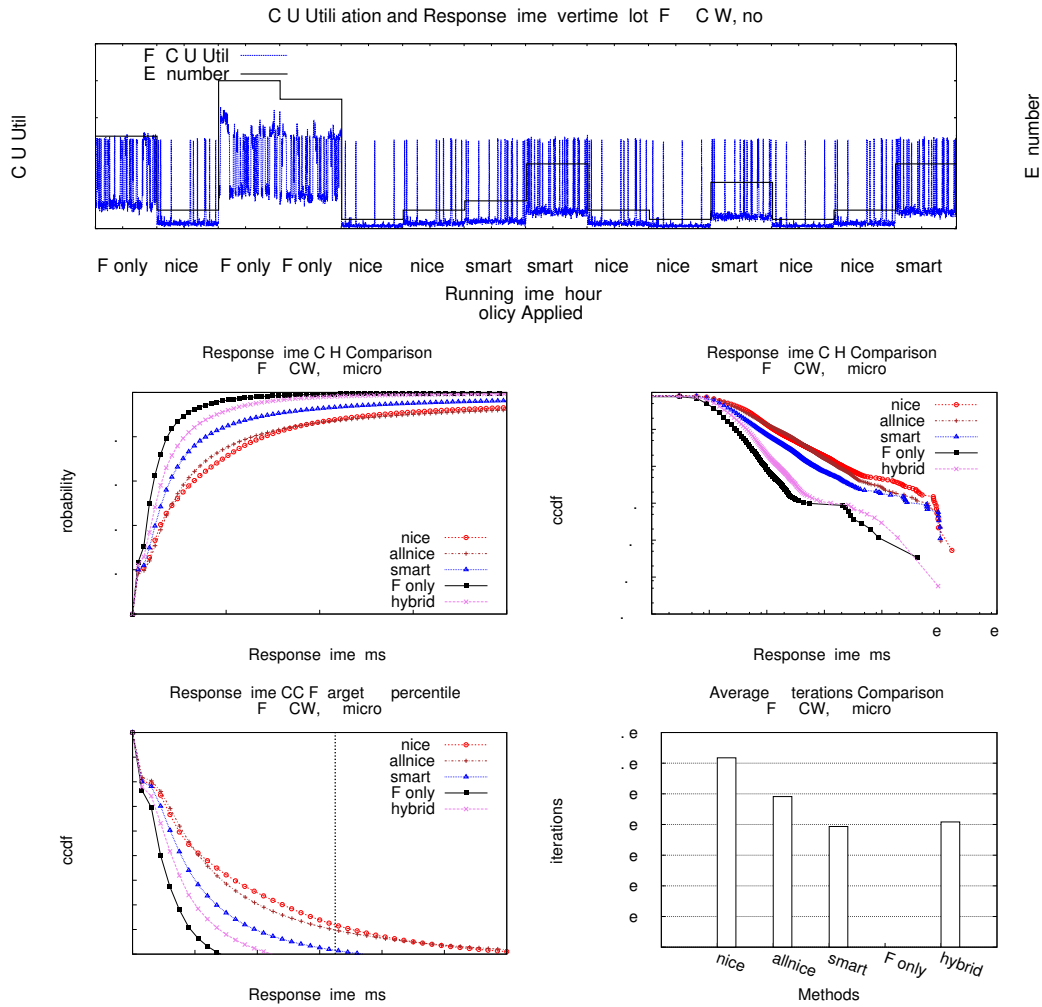
Figure 8: Scenario 2 - BG: CPU total demand: 100% .

allow the high and low priority processes to share cores, whereas the virtual machine approach generally assigns a whole number of cores to each virtual machine, although the exact number may change dynamically [18].

Other researchers have focused on the progress rate of applications to determine appropriate resource sharing between them [8, 9]. 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 [9]. Douceur and Bolosky share our goal more clearly, identifying very low priority tasks that should not be allowed to impact the foreground task [8]. 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 foreground application themselves.

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 [3]. 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. They 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.

To sum up, the proposed middleware for background 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
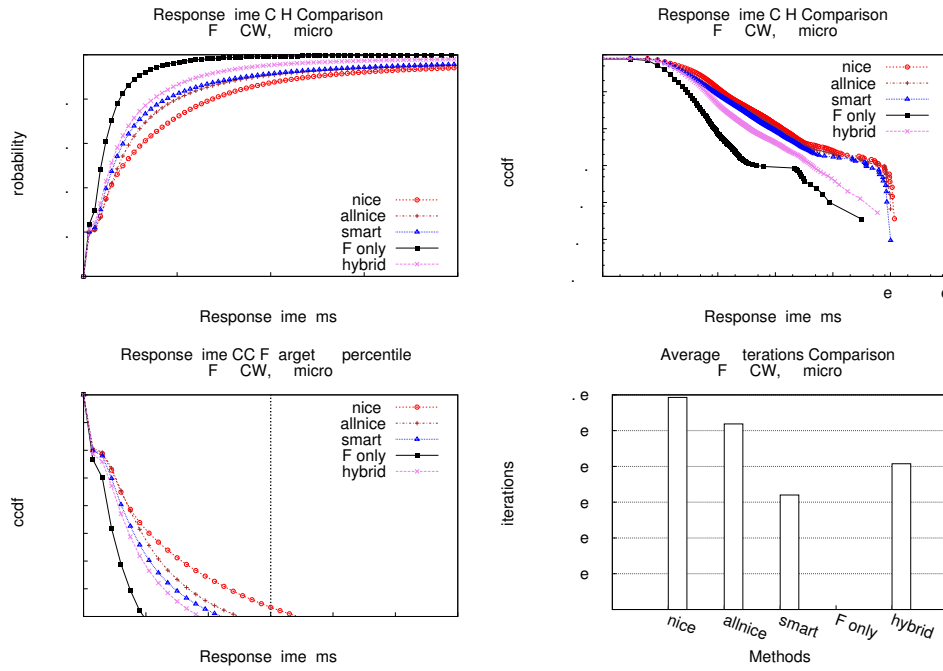
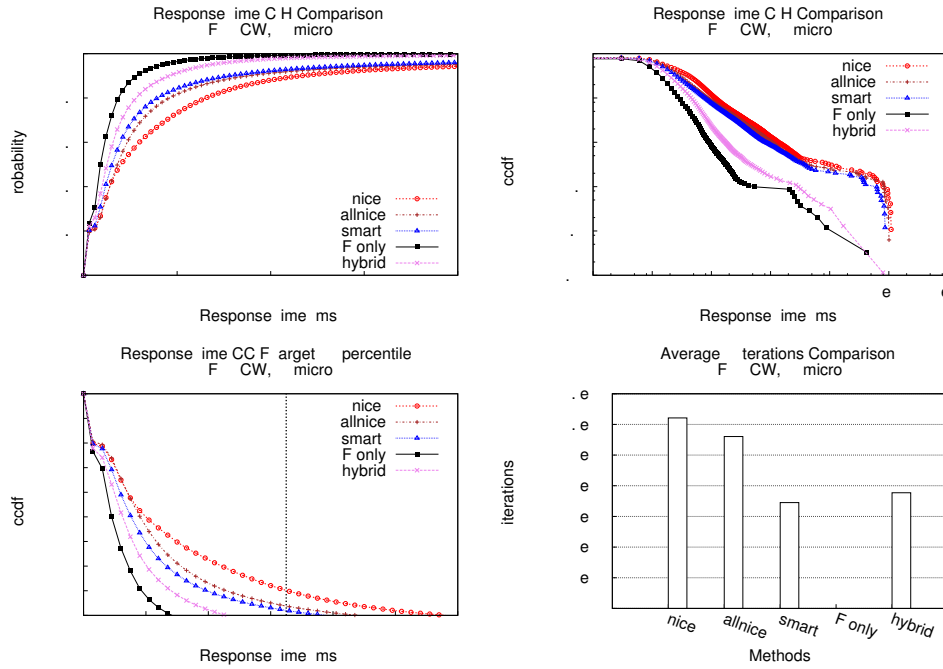**Figure 9: Scenario 1 - BG: CPU total demand: 45% .**

Figure 10: Scenario 2 - BG: CPU total demand: 45% .

interrupting the current services. To deploy it, a learning phase is required to characterize the statistical distribution of the foreground traffic's busy periods to determine the optimal periods to suspend the background job execution, and based on this information it launches *different* background job scheduling policies that can best fit the current system conditions. Therefore, it is lightweight, portable, and flexi-

ble as it manages to take advantage of the benefits of several monolithic background scheduling policies while minimizing their respective shortcomings.

6. CONCLUSIONS

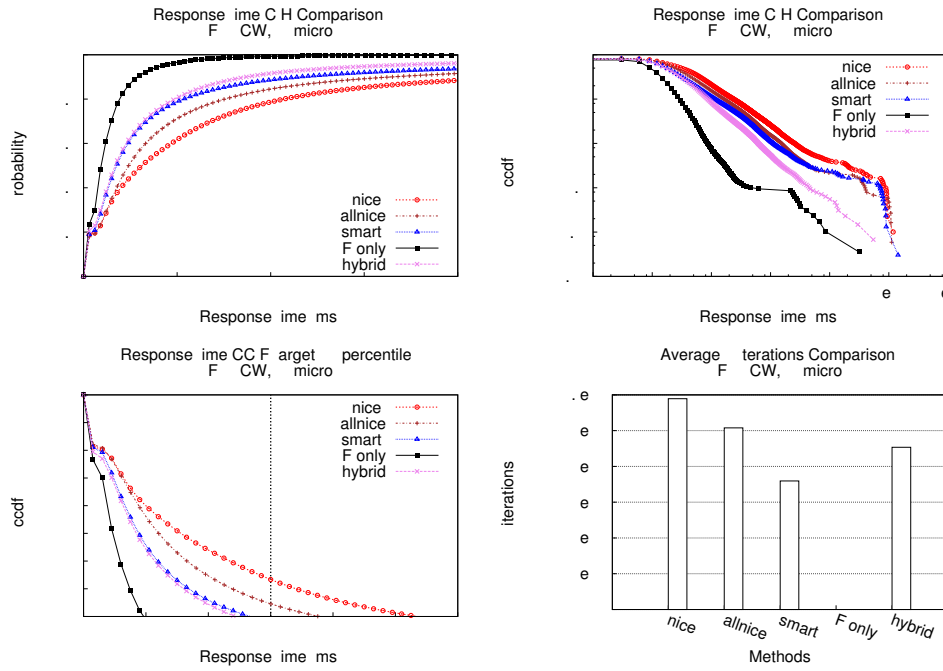In this paper, we proposed a middleware scheme that remedies the shortcomings of monolithic background schedul-

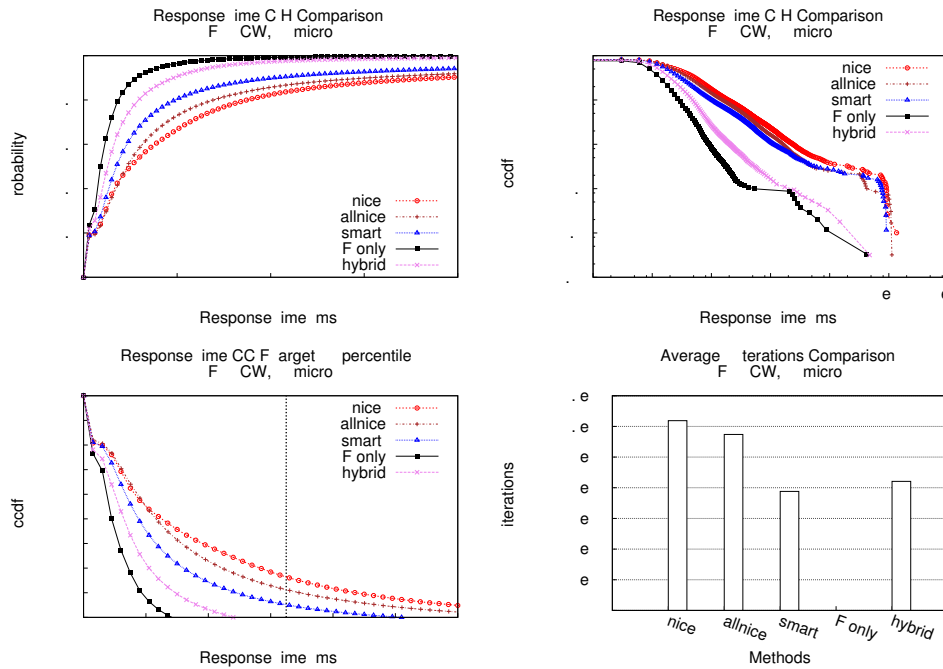**Figure 11: Scenario 1 - BG: CPU total demand: 160% .**

Figure 12: Scenario 2 - BG: CPU total demand: 160% .

ing and provides strong performance guarantees on foreground work. Our middleware scheme learns the foreground resource requirements and stores such information in a compact way, in the form of a cumulative data histogram. This learning allows the scheme to determine the appropriate scheduling policy based on pre-specified performance targets and current system load levels. The scheduling middleware is built above standard system tools, ensuring that is portable, with low overhead, and that can be deployed easily at a node level within large scaled-out systems. Detailed experimental results verified its effectiveness and robustness. In the future, we plan to add more policies and experiment with a wider array of different applications. We also plan to explore the case of meeting background work targets (e.g.,

close to a deadline) but still with minimum foreground performance impact.

7. ACKNOWLEDGMENTS

8. REFERENCES

[1] TPC-W. http://www.tpc.org/tpcw/.

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

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

[4] 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.

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

[6] 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*, pages 169–180, 2009.

[7] T. Cucinotta, F. Checconi, L. Abeni, L. Palopoli, and L. Palopoli. Self-tuning schedulers for legacy real-time applications. In *EuroSys*, pages 55–68, 2010.

[8] J. R. Douceur, W. J. Bolosky, and W. J. Bolosky. Progress-based regulation of low-importance processes. In *SOSP*, pages 247–260, 1999.

[9] A. D. Ferguson, P. Bodak, S. Kandula, E. Boutin, R. Fonseca, and R. Fonseca. Jockey: guaranteed job latency in data parallel clusters. In *EuroSys*, pages 99–112, 2012.

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

[11] C. Krasic, M. Saubhasik, A. Sinha, and A. Goel. Fair and timely scheduling via cooperative polling. In *EuroSys*, pages 103–116, 2009.

[12] 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*, pages 271–282, 2012.

[13] J. N. Matthews, W. Hu, M. Hapuarachchi, T. Deshane, D. Dimatos, G. Hamilton, M. McCabe, and J. Owens. Quantifying the performance isolation properties of virtualization systems. In *Experimental Computer Science*, page 6, 2007.

[14] 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.

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

[16] 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*, pages 205–216, 2009.

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

[18] 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*, pages 13–26, 2009.

[19] L. Sha, T. F. Abdelzaher, K.-E. ÃĚrzÃľn, 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. pages 101–155, 2004.

[20] M. Stonebraker. The case for shared nothing. *IEEE Database Eng. Bull.*, 9(1):4–9, 1986.

[21] E. Thereska, A. Donnelly, D. Narayanan, and D. Narayanan. Sierra: practical power-proportionality for data center storage. In *EuroSys*, pages 169–182, 2011.

[22] 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*, ICAC '12, pages 33–42, New York, NY, USA, 2012. ACM.

[23] F. Yan, S. Hughes, A. Riska, and E. Smirni. Overcoming limitations of off-the-shelf priority schedulers in dynamic environments. In *Proceedings of the 21st International Symposium on Modeling, Analysis, and Simulation On Computer and Telecommunication Systems*, MASCOTS 13, 2013.

[24] F. Yan, A. Riska, and E. Smirni. Fast eventual consistency with performance guarantees for distributed storage. In *ICDCS Workshops*, pages 23–28, 2012.

[25] 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*, pages 265–278, 2010.

[26] 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*, pages 96–105, 2003.