Fastrack for Taming Burstiness and Saving Power in Multi-Tiered Systems[*]

A. Caniff, L. Lu, N. Mi, L. Cherkasova, and E. Smirni
{awc, llu, esmirni}@cs.wm.edu, ningfang@ece.neu.edu, lucy.cherkasova@hp.com

Abstract

Burstiness (i.e., sudden surges) in user demands in enterprise systems that operate under the multi-tiered paradigm is a common phenomenon that leads to overprovisioning: the system is configured with excess hardware to meet peak user demands, often resulting in excessive (and unnecessary) power costs. In this paper, we present Fastrack, a parameter-free algorithm for dynamic resource provisioning that uses simple statistics to promptly distill information about changes in workload burstiness. This information, coupled with the application's end-to-end response times and system bottleneck characteristics, guide resource allocation that shows to be very effective under a broad variety of application burstiness profiles and bottleneck scenarios. Extensive simulations illustrate Fastrack's robustness for consistently meeting predefined service level objectives while minimizing power usage.

1 Introduction

Capacity planning that focuses on both performance and power is a critical task for the effectiveness and business success of today's enterprise systems. Identifying customer demands and consequent system bottlenecks is central for provisioning such systems in order to meet predefined service level objectives (SLOs). Sudden surges in user demands are a common phenomenon (and can be fueled by special events, e.g., sales), resulting in temporal loads of orders of magnitude higher than the average load. Capacity planning must cater for such phenomena as perceived user performance must always be within certain SLOs. Consequently, traditional capacity planning promotes over-provisioning, i.e., the system is always configured to meet peak user demand, resulting in systems where the cost of power can be crippling.

Resource allocation in a multi-tiered system is more challenging than in a single-tiered one. In a multi-tiered system, the tier that is the bottleneck regulates the flow of requests and usually dominates performance. Alleviating the bottleneck tier by assigning more processing power is straightforward but should be done with caution as the bottleneck may simply shift to another tier [13]. Traditional provisioning relies on workload observation that triggers resource reallocation when certain thresholds are exceeded [9, 13]. The effectiveness of such techniques depends on astute selection of their parameters. What makes resource allocation even more challenging in a multi-tiered system is the phenomenon of bottleneck switch that further exacerbates the difficulty of the problem [7, 8]. Resource allocation that also requires saving as much power as possible without compromising performance becomes a conundrum for system designers.

In this paper we offer an application-centric approach to the above difficult problem. We focus on a multi-tiered system that operates under a bursty workload and present a parameter-free algorithm called Fastrack that *quickly* tracks achievable performance and workload burstiness to self-adjust the allocation of available resources with the aim of optimizing performance while using *minimal* resources. Fastrack uses online measurements to determine whether the system experiences a *true peak* or simply variability in user arrivals and quickly determines the start of a burst, signaling the need to assign more computing resources. Correspondingly, it also detects the end of a burst, i.e., rapid returns to normal traffic intensity, signaling the need to reduce computing resources without any performance penalty. During a workload surge, the algorithm uses a "pro-active" approach to quickly identify the surge, and tames its effects by summoning new resources before performance starts to suffer. On the other hand, after the algorithm detects a quiet period, it releases resources with a slower pace such that jobs that are accumulated during a burst are flushed and the operation of the system reverts to normal. Timely identification of the start and end of bursts is the core of Fastrack and together with the system's SLOs guides when to expand/contract the number of resources to the application.

We simulate the behavior of a multi-tiered system that is built according to the classic TPC-W paradigm. Detailed experimentation demonstrates the robustness of Fastrack if the workload conditions span from mildly bursty to very bursty. Comparisons with other resource allocation policies in the literature illustrate the effectiveness of Fastrack to stay close to the SLOs while using as few resources as possible. The remainder of the paper presents our results in more detail.

2 Burstiness: Friend or Foe?

Burstiness in flows of systems has been shown detrimental for performance. Recent work in multi-tiered systems has demonstrated that burstiness may be endogenous, e.g., it may be due to database locks or caching in one of the tiers [7], or exogenous, i.e., due to bursty external arrivals[8]. Burstiness in flows results in the phenomenon of persistent "bottleneck switch" where performance measures are counter-intuitive, e.g., user SLOs are grossly violated while performance measures such as device utilizations are moderate, for more details we direct the interested reader to [7]. In [7], the authors successfully incorporated the *index of dispersion* I into new capacity planning models of multi-tier enterprise systems. In this section, we show how to use I to infer information about

[*]This work was partially supported by the National Science Foundation under grants CNS-0720699, CCF-0811417, and CCF-0937925. A. Cannif, L. Lu, and E. Smirni are with the Computer Science Department, College of William and Mary, Williamsburg, VA. N. Mi is with the Electrical and Computer Engineering department at Northeastern University, Boston, MA. L. Cherkasova is with HP Labs, Palo Alto, CA.

the patterns of upcoming workloads. In effect, we advocate using burstiness as a friend rather than a foe. We show that I can provide a simple yet powerful way for prompt identification of the *start* and the *end* of a bursty period.

Given a time series of random variables $\{X_n\}$, where $n = 0, \ldots, \infty$, the index of dispersion is defined as follows:

$$I = SCV\left(1 + 2\sum_{k=1}^{\infty} \rho_k\right), \qquad (1)$$

where the squared-coefficient of variation (SCV) quantifies the variability in this series and the lag-k autocorrelation coefficient ρ_k expresses the relationship between consecutive occurrences of the random variable with respect to its mean[1]. The mathematical definition of I in Eq.(1) shows that the index of dispersion jointly captures *variability* and *burstiness* in a single number and that the summation of autocorrelations at all lags further gives a measure of the strength of burstiness.

To illustrate how I as a single measure can capture burstiness, we show the counts of arrivals per 10 second windows as a function of elapsed time (total elapsed time of 90,000 seconds) for two arrival patterns, one with high variability but no burstiness and one with a clear burstiness profile, see Figures 1(a) and 1(b). Figure 1(a) shows an arrival sequence that was generated with a hyperexponential distribution and Figure 1(b) corresponds to a bursty sequence that was generated by a MAP [2] with the same mean and SCV as the hyperexponential. When there is no burstiness, the value of I exactly equals the squared-coefficient of variation because the summation of all autocorrelation coefficients ρ_k is equal to zero. In Figure 1(a), $I = SCV = 20$ because the hyperexponential that is used to generate this arrival trace has high variability but no burstiness. The MAP process that is used to generate the arrival stream in Figure 1(b) has instead $I = 3014$. We remark that arrival patterns like the one illustrated in Figure 1(b) are commonly found in real systems, see for example the arrival pattern in the classic FIFA World Cup trace where the index of dispersion for consecutive days in the arrivals to the FIFA web site reaches values as high as 8400. For more details we direct the reader to [8].

Looking at the difference in the two arrival patterns of Figure 1(a) and 1(b) one can immediately see that using simple observations such as sudden changes in the arrival rate of incoming requests to guide readjusting of resource allocation is not straight forward. Detecting changes in the arrival rate within successive time windows may be perilous: a current prediction may be incompatible with the upcoming next window, therefore suggesting too often and wasteful changes in resources. For a variable (but not bursty) workload as the one depicted in Figure 1(a) the best strategy for effective resource allocation is to rely on classic capacity planning in order to identify the ideal resource allocation to meet service and power level objectives. The pattern depicted in Figure 1(b)

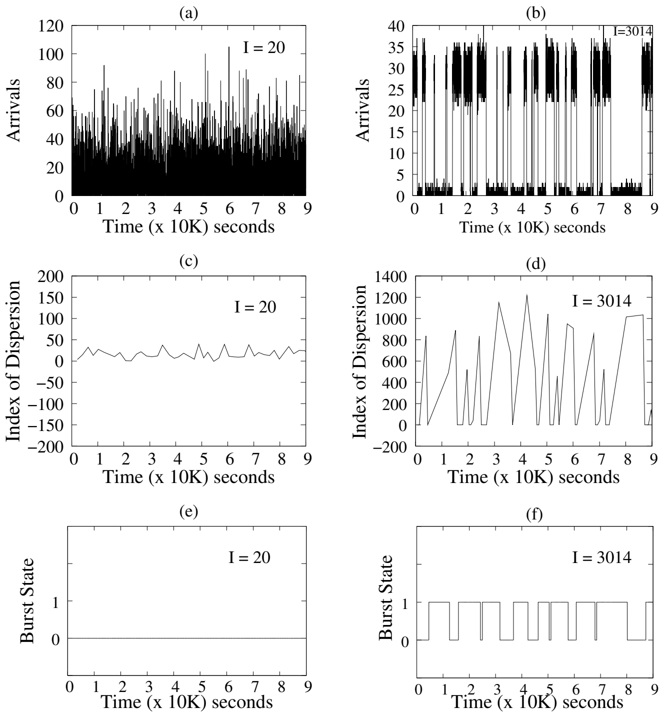

Figure 1. Illustrating the difference in two arrival processes which are driven from a hyperexponential with mean $\mu^{-1} = 1$ and $SCV = 20$ (left column of graphs), and from MAP with the same mean $\mu^{-1} = 1$ and $SCV = 20$ but strong burstiness (right column of graphs). The left column corresponds to a process with high variability but no burstiness and the right column to a process that is bursty.

however, suggests that this is a case where one allocation cannot fit well all periods. In this paper, we propose to use I to quickly detect changes in arrival intensities. An obvious disadvantage for any practical implementation of the calculation of I using Eq.(1) is the ∞ term in the summation. Here, we use a batch of $K = arrivalrate \times C$ requests, where C is a constant set to 2000[2] and because we are interested in only short-term changes in the process, we substitute the ∞ term in Eq.(1) with only the first ten percent of the batch size. Dramatic changes in I within successive batches show that there has been a significant change in the arrival rate for a given batch, and that information coupled with information about the current and previous arrival rates can show whether a burst has begun or ended. Note that comparing statistics of successive batches of jobs is superior to comparing statistic within successive windows of time, as the number of requests withing successive windows may vary widely.

The algorithm for determining the start and end of bursts is shown in Figure 2. The algorithm runs on batches of K requests. For every batch, it calculates the index of dispersion and the arrival rate for the current set of K requests, and updates the total arrival rate and SCV for all requests processed. If the absolute change in the index of dispersion between the current batch and the previous batch is greater than twice the

[1] In a time series of random variables $\{X_n\}$, where $n = 0, \ldots, \infty$, ρ_k shows the value of the correlation coefficient for different lag $k > 0$: $\rho_k = \frac{E[(X_t-\mu)(X_{t+k}-\mu)]}{\sigma^2}$, where μ is the mean and σ^2 is the common variance of $\{X_n\}$.

[2] We have used several constants C from 100 to several thousand and we found that 2000 work effectively for our purposes for a wide variety of traces with different burstiness profiles.

```
for every batch of $K$ requests do
    current_I ← calculate I for current batch
    batch_rate ← arrival rate for current batch
    total_rate ← update average arrival rate (all requests)
    $SCV$ ← update $SCV$ (all requests)
    if (|current_I - old_I| > 2*$SCV$ ) AND
      $\left(\frac{\text{old\_I}}{\text{current\_I}} > 2 \text{ OR } \frac{\text{old\_I}}{\text{current\_I}} < 0.5\right)$
      if batch_rate > total_rate
        burst starts
      else
        burst ends
    old_I ← current_I
```

Figure 2. Algorithm for detecting bursts.

SCV, this signals the beginning (or end) of a burst [3]. Note that the direction of the change in the index of dispersion between successive batches does not indicate whether a burst has begun or ended. To determine this, we look at the arrival rate of the current batch, and compare it to the total (i.e., all requests) arrival rate into the system. A batch with arrival rate higher than the system average coupled with a change in the index of dispersion indicates the beginning of a burst. Similarly, a change in the index of dispersion coupled with a lower than average arrival rate signals the end of a burst.

Figures 1(c) and 1(d) plot the I values for successive batches as a function of elapsed time. It is interesting to see that changes in I (corresponding to spikes or dips) are reported after longer elapsed time during quiet periods, see Figure 1(d). It is also clear that successive changes in I are dramatic for the case of the bursty process, while they remain modest for the hyperexponential case. Observe also the difference in the range of the two y-axes between the two graphs.

The outcome of the algorithm is illustrated in Figures 1(e) and 1(f). Only for the bursty arrivals case the algorithm correctly detects when burstiness starts (state 1) and when it ends (state 0) and these changes follow well the actual bursts depicted in Figure 1(b). One should notice that the algorithm is slower in the detection of a quiet period. This happens because a batch of K takes longer to gather after the conclusion of a burst. However, this turns out to be beneficial for performance since the release of resources is slower and this allows for the system to flush faster the waiting queue that possibly built up during a burst. In general, comparing Figures 1(b) and 1(f), it is interesting to note that the algorithm correctly captures the beginning and end of *most* bursty periods, albeit making some errors. If a burst starts though, it tends to quickly recover, see for example the behavior at time-stamp 5 in Figures 1(b) and 1(f).

3 Allocation Algorithm: Fastrack

Our focus is on effective resource allocation in multi-tiered systems, and, to ease description, we assume an architecture that is used by the TPC-W benchmark, a standard benchmark that is routinely used for capacity planning of e-commerce systems. This multi-tier application uses a paradigm which consists of a web server, an application server, and a back-end database. The web server and the application server reside usually within the same physical server, which is called front sever. Client requests may cycle between the front and back-end (database) servers before they are returned to the client. The TPC-W benchmark implements a fixed number of emulated browsers (EBs) that is equal to the maximum number of client connections. Each EB sends requests to the system with an average think time E[Z] that represents the time between receiving a Web page and the following page download request. We further assume that there is a pool of front servers that the application can use while there is only one back-end server. The rationale for this assumption is that having multiple back-end servers requires secure and consistent database replication, which by its nature is a difficult problem that is outside the scope of this work. Consequently, we focus on how to allocate resources on the front tier only.

Under bursty workload conditions, it is tremendously important for a successful allocation algorithm to act *proactively*, i.e., to quickly detect a surge and immediately assign additional resources, that is, before queues start to build up and performance degradation becomes noticeable to the user. The algorithm presented in the previous section, see Figure 2, serves this purpose. Similarly, when the end of a burst is detected, the algorithm should reduce its resource assignment in order to save power. Independent from the detection of bursty conditions, it is also important to keep track of changes in the target performance measures and continuously compare them with those of systems SLOs, which are usually in the form of percentiles of user response times (RTs). A successful allocation algorithm should also act *reactively*. i.e., it should monitor deviations of the user performance measures from the target SLOs and quickly adjust resources aiming at minimizing these deviations. Similarly, if RT percentiles are well below SLOs, it may be desirable to reduce allocation in order to save power. Fastrack aims at meeting all of the above targets.

Fastrack first allows for a small amount of warm-up time to pass where the system simply collects statistics with the current configuration to calculate percentiles. We assume that the target SLOs are expressed in the form of percentiles[4]. After the warm-up time, the system looks at batches of K requests and collects RT percentiles for the current configuration while updating the RT percentiles of all requests into the system. Collecting percentiles for *each* allowable configuration in the system allows for a quick performance reference, essentially we keep memory of reachable performance in order to avoid unfavorable configurations. Fastrack collects other workload statistics (see Figure 2) that aid in identifying a burst.

Beyond collecting statistics across batches, it is important to identify whether the database is the system "bottleneck". This determines whether it is judicious to assign more front servers during a burst or when there is a violation in the performance SLOs. A database is deemed the bottleneck if the contribution to the end-to-end RT percentile from the front

[3]We require the computed I for the current batch to be larger than twice the SCV in order to prevent thrashing for small changes in I. By setting the threshold to be twice the SCV, we essentially require that the difference across the two batches in the sum of all autocorrelation coefficients ρ_k to be greater or equal to 0.5. This is a conservative estimation provided that the values of ρ_k by definition, see Equation [1], are between -1 and 1.

[4]SLOs could be also expressed in simpler forms such as performance averages with minimum changes into the algorithm.

servers is larger than twice the time spent on the front servers. This requirement stems from the non-linear relationship between response times and utilization, especially in moderate to high utilization levels [6]. The database bottleneck condition is critical for successful resource allocation in the two tiers. If simply the utilization levels of the two tiers are compared to decide whether the database or the front server is the bottleneck, the tier contributions to end-to-end response times are ignored. A more utilized database but a much faster one than the front tier can withstand more load without appreciable change in end-to-end response times. Correspondingly, if the bulk of end-to-end time is spent in the less loaded (but slower) front tier, it is advantageous to drastically reduce the front tier time and perhaps increase the database time. If the database is instead slow (but relatively low utilized) and contributes to the bulk of request end-to-end time, then assigning more front servers to quickly serve a burst will only exacerbate RTs at the (already slow) database. The situation is much simpler if the end of a burst is identified: the number of front servers are reduced to conserve power.

In the absence of a burst Fastrack looks at the performance of the current batch to adjust provisioning. If the RT percentiles of all requests are less than the target SLOs or if the database is the bottleneck, then Fastrack reduces the front server resources in order to save power and alleviate a very loaded database server. If however the RT percentiles in the current batch are higher than the target SLOs and the database is not the bottleneck server, then performance can improve by increasing the front servers. It is important to note that the rate of increase or decrease can be done using different functions. A conservative increase/decrease would be to add/subtract one server, a more drastic approach would be to double/halve the existing servers. In the specific instantiation of Fastrack in this paper, we add/increase servers using the double/halve rule to quickly react to changes in burstiness. Figure 3 gives the high-level description of Fastrack.

```
for every batch of $K$ requests do
    update total RT percentiles (all requests)
    update current RT percentiles (current resource configuration)
    Proactive Adjustment:
        Use Burst Detection Algorithm (see Figure 2)
    if there is a change in workload burstiness
        if burst starts   // improve performance
            if (!db_bottleneck) then increase front servers
        if burst ends
            decrease front servers   // save power
    else   // if no change in burstiness
    Reactive Adjustment: Compare RTs to target RTs (SLOs)
    if (total_RT_percentile $<$ target_RT_percentile
        OR db_bottleneck)
        decrease front servers   // save power
    else if (current_RT_percentile $>$ target_RT_percentile
        AND (!db_bottleneck) )
            increase front servers   // improve performance
```

Figure 3. Fastrack: an algorithm for adjusting resources.

4 Performance Evaluation of Fastrack

We evaluate the effectiveness of Fastrack by simulating the workload flows in a typical TPC-W 2-tier implementation (i.e., a front server and a database server). We extend the basic model to include a pool of 8 front servers that can be brought online/offline during the experiment. We focus on the effectiveness of the algorithm under a wide variety of workloads: different burstiness levels, different loads, different workloads demands (i.e., different bottlenecks), and also under the difficult case of bottleneck switch. Our purpose is to show the robustness of the algorithm under all of the above conditions.

The TPC-W defines 14 transactions, each of which can be generally classified as "browsing" or "ordering". Furthermore, TPC-W defines three transaction mixes based on the weight of each type in the particular transaction mix:

- *browsing mix*: 95% browsing and 5% ordering;
- *shopping mix*: 80% browsing and 20% ordering;
- *ordering mix*: 50% browsing and 50% ordering.

The think times of emulated browsers are modeled by using two different MAPs, each with a different burstiness profile. Consistent with the specifications of the TPC-W benchmark, both MAPs result to average user think time equal to 7 seconds but their SCV is equal to 20 (i.e., inter-arrival times into the system are very variable). Furthermore, the two MAPs have different burstiness profiles: one results in the index of dispersion $I = 41$, i.e., very low burstiness, and the other one with $I = 1,806$ which constitutes significant burstiness.

For a given experiment, we choose one of the transaction mixes, and use MAPs fitted from TPC-W experiments on a real testbed to model the service processes in the front and database servers. For more information on the fitting process that results in MAPs that accurately capture the service processes in TPC-W we direct the interested reader to [7]. For each of transaction mixes and arrival MAPs, we run an experiment for a large range of populations to gauge performance across different system loads. We gather pertinent system statistics such as user response time percentiles and power usage, and then compare these results to systems with a static number of servers. For each experiment we also report results of static configurations with 1 and 8 front servers, resulting in upper and lower performance bounds, respectively.

Power usage is measured in raw machine seconds, i.e., the sum of times that front servers are in operation. To ease comparison of power usage, all simulations run for a fixed amount of time, resulting in a range of several hundred thousand to several million requests processed on a given run. We choose to work with a constant time interval for practical reasons; if the simulations are run with a fixed request sample instead, then each simulation would run for drastically different amounts of time. Since one of the main foci of our work is on power savings, we opt to use a constant time scale, such that different runs can be judged on raw machine time units used, rather than averaging power costs across different simulation times. Traditionally, there are different ways to compute the power used by a server. Each server has a minimum power usage p_{idle} that corresponds to the idle server, and a maximum power usage p_{busy} that corresponds to 100% CPU utilization. The power used by a server is estimated as $p_{idle} + u \cdot (p_{busy} - p_{idle})$, where u is the CPU utilization of the server [12]. In this work, we use a simplified power usage accounting in raw machine seconds to explicitly reflect

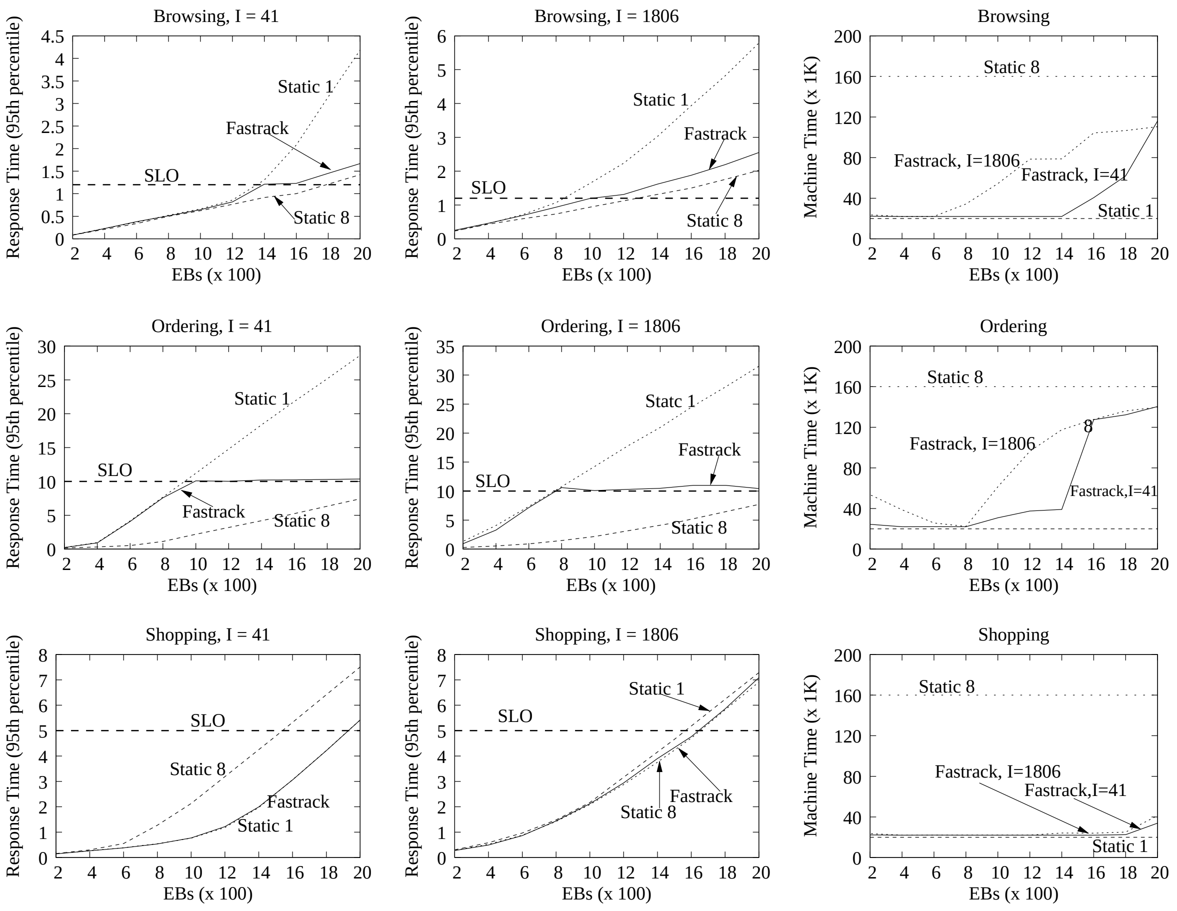

Figure 4. Illustrating 95th percentiles of response times and raw machine times under three transaction mixes and two arrival MAPs. Each row of graphs shows performance numbers for a different transaction mix. The left column is response time percentiles for the low burstiness, the middle column is the high burstiness, and the right column summarizes the power usage of each configuration.

the number of servers in the configuration used over time.

Figure 4 depicts the performance results for the various experiments. The figure, organized as a three by three grid, presents performance numbers in the form of 95th percentiles of user response times for the three TPC-W mixes when the arrival workload exhibits slight burstiness (leftmost column of graphs), high burstiness (middle column of graphs), and raw machine times units for the front tier (rightmost column). All results are presented for various populations (emulated browsers) in the system such that we show performance in low loads (low populations) and high loads (high populations). The first row of graphs corresponds to the browsing mix, the second one to the ordering mix, and the last row to the shopping mix. Each plot in Figure 4 shows the results for Fastrack, the two boundary cases with a static number of front servers equal to 1 and 8, and the target SLOs. We set a different SLO for each mix. This is a user defined input to the algorithm, and is dependent on the performance level needed for the application. We consider the three mixes to be representative of three different application types, each with a different SLO. The last column of graphs shows the machine times as a function of population for the static 1 and 8 cases (the two parallel flat lines that correspond to the two boundary static cases) as well as the machine times for Fastrack. Naturally, the closer the Fastrack machine times are to the lower flat line, the lower the power consumption. In the following subsections, we discuss the performance and power consumption under various transaction mixes.

4.1 Bottleneck Switch: Browsing Mix

The browsing mix (top row of graphs in Figure 4) exhibits bottleneck switch inherent to the service processes at the front and database tiers [7], before any burstiness is added to the arrival process. Without a constant bottleneck across time, it is difficult to efficiently provision servers to this mix, as increasing capacity at one tier does not have the same effect as if that tier were the consistent system bottleneck. Our algorithm is able to stay near the response time SLO for this mix, and is able to save significant power while doing so. In the low burstiness case, up until 1400 EBs Fastrack is able to use only 1 server most of the time (see the machine time graph) while still meeting the response time target. Beyond 1400 EBs, more servers must be assigned to the front servers to sustain good throughput and good response time. Correspondingly, more front servers translate to high power usage. With 1800 EBs, the static 8 server configuration is just slightly above our SLO target of 1.2, while Fastrack misses the target by about 20%. However, Fastrack was able to achieve that level while using an average of only 3 servers over the course of the run. Fastrack sees that the added front servers will not make a drastic difference in response time, and thus not worth the added power cost of using all 8 servers. Fastrack strikes a balance between reaching the SLO goal and achieving good power efficiency.

Figure 5 shows the system throughput (completed requests per time unit) as a function of the number of EBs for browsing when $I = 41$ (low arrival burstiness) and illustrates the strengths of dynamic server provisioning. Up to 1200 EBs, the static 1 server and static 8 server configurations result in the same throughput, as the load is light enough that the extra servers do not help appreciably. However, at 1400 EBs and beyond, throughput begins to level off for the static 1 arrangement but it continues to grow for the static 8 arrangement. Fastrack exploits the positives in both static configurations. In low load, i.e., for less than 1400 EBs, it behaves as the static 1 policy (see also the rightmost graph in Figure 4 for browsing, where the machine times clearly suggest that 1 server is only used if EBs are less than 1400). For more than 1400 EBs, Fastrack tends to assign more front servers as the EBs increase, and in doing so maintains the same throughput as the static 8 policy (see again the the rightmost graph in Figure 4 for browsing, where the raw time is commensurate with the increase of EBs).

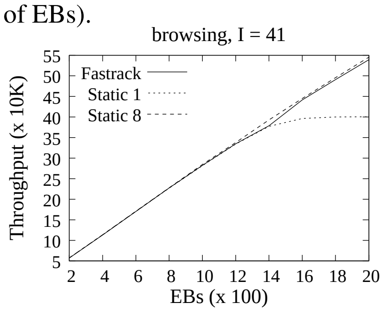

Figure 5. Throughput (served requests per time unit) for the browsing mix, low burstiness.

In the high burstiness case, (see middle graph and rightmost graphs on the top row of Figure 4) spikes in arrivals cause raw machine times to increase under lighter load, and the SLO is violated. As in the low burstiness case, Fastrack provisions servers efficiently, staying near the much more power intensive static 8 case, while saving significant power. Past 1400 EBs, even the static 8 case surpasses the SLO threshold, but Fastrack continues to maintain a balance between good response times and efficient power usage.

4.2 Persistent Bottleneck: Ordering Mix

The ordering mix (see middle row of graphs in Figure 4) poses a difficult problem for effective capacity planning. By observing the end-to-end response time of requests across different EB levels, we see that the database contribution to the end-to-end times is much smaller in comparison to the front server. As a result, adding more front servers can significantly improve the responsiveness of the system before the database is unable to handle the increased arrival intensity from the front server and becomes truly the bottleneck that regulates the flow in the system. This mix clearly shows one of the perils of over-provisioning; although a static 8 arrangement of servers can maintain excellent response times, it is well below the SLO, thus there is plenty of room for reducing power consumption by reducing front server resources. Fastrack uses minimal power until the SLO threshold is reached. As the load increases, Fastrack only increases server usage enough to just maintain the SLO, (see machine time graph for the ordering mix, rightmost column of graphs in Figure 4).

4.3 Persistent Bottleneck: Shopping Mix

The shopping mix (see last row of graphs in Figure 4) provides a nice contrast to the ordering mix. Here, even in the simple case of 1 front server only, the database is the system bottleneck. Its relative speed compared to that of the front server tier is such that it dominates end-to-end response times. Provisioning additional front servers can actually have a drastically negative impact on the system performance as the static 8 experiment sees worse end-to-end response times than the more power efficient configurations. In the low burstiness case, extra server provisioning is detrimental to the system, as it completely overloads the database tier. This mix shows the other danger of over-provisioning, and makes a great case against naively adding servers without careful capacity planning. In the case of high burstiness, it is still not beneficial to add more servers, although the bursts in traffic cause some overloading at the front server tier, which brings the high power solution back closer to the low power ones. Although additional provisioning at the front server would initially improve response time percentiles when a burst occurs, the net effect would be negative, as it would still end up overloading the database. When loads grow large enough to violate the SLO, our algorithm correctly identifies the database as the system bottleneck, and does not increase capacity at the front server tier even in the presence of bursts, achieving excellent power savings without compromising performance.

4.4 Transient Behavior

To show the algorithm's responsiveness to burstiness, we illustrate how Fastrack deploys servers for a representative case. We have selected here the ordering mix and EBs=1000. Figure 6 shows the request arrivals of low and high burstiness to the front server. Arrivals are plotted across time. The reciprocal graphs (second row in Figure 6) show the number of deployed front servers for the two experiments across time. At 1000 EBs, the ordering mix cannot maintain the SLO with just 1 server. Throughout the run, Fastrack deploys a second server during bursts, and then removes it again when the burst recedes to maintain good power efficiency. In the high burstiness case (right column of graphs in Figure 6), traffic bursts often cause Fastrack increase front tier servers to 8. This mitigates the effects of the bursts and good response times are maintained. Despite needing 8 servers at some times, the algorithms follows bursts well enough and the majority of time is still spent in low power configurations while still maintaining the SLO.

5 Comparisons

In this section we evaluate the effectiveness of Fastrack in comparison with another dynamic assignment algorithm for multi-tier systems [13]. The algorithm presented in [13]

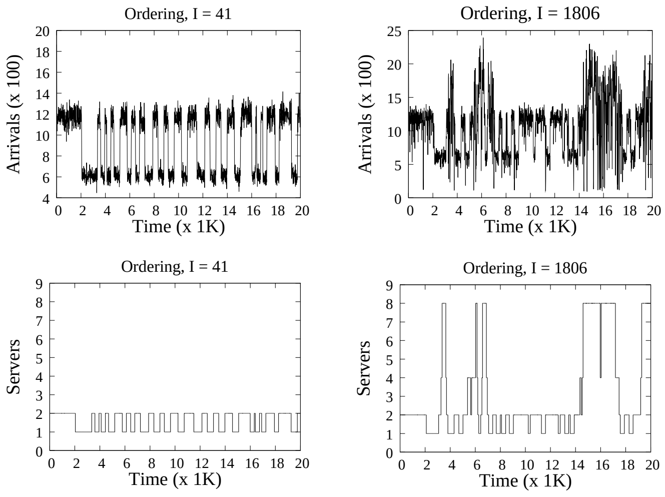

Figure 6. Comparison of transient server allocations of Fastrack for the ordering mix, EBs=1000, for low and high burstiness levels.

(henceforth referred to as the *umass* algorithm) is based on a multi-tier application model that can be summarized as follows: The peak arrival rate λ_i to the system is expressed as:

$$\lambda_i \geq \left[s_i + \frac{\sigma_a^2 + \sigma_b^2}{2 \cdot (d_i - s_i)} \right]^{-1},$$

where σ_a^2 and σ_b^2 are the variance of inter-arrival time and the variance of service time, respectively, d_i is the mean response time at tier i and s_i is the mean service time at tier i.

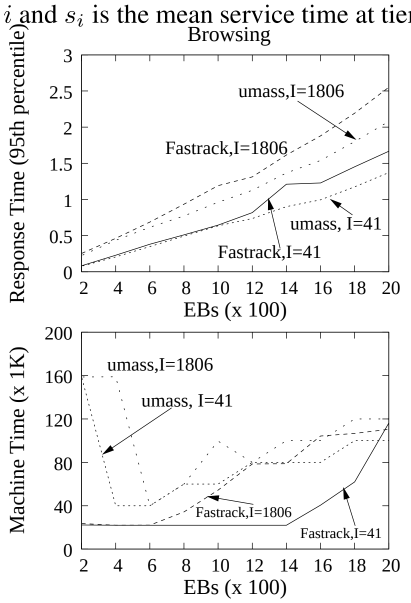

Figure 7. Comparison of performance and power savings of Fastrack and the *umass* algorithm.

A lower bound of λ_i is then substituted into Little's Law to find the number of servers needed to accommodate a given arrival rate.

$$\eta_i = \left\lceil \frac{\beta_i \lambda_\tau}{\lambda_i Z} \right\rceil, \quad (2)$$

where $\frac{\lambda_\tau}{Z}$ is the request arrival rate and β_i is a tier-specific constant. The reactive portion of the *umass* algorithm is a comparison of predicted arrival rate to the observed arrival rate. Data from a long running system are used to determine a predicted arrival rate based on time of day, day of week. For the purpose of our experiments, we assume that the "predicted" arrival rates are already known a priori. For a reactive adjustment to be made, the ratio of the observed arrival rate to the predicted rate must be larger than a threshold:

$$\frac{\lambda_{obs}}{\lambda_{pred}} > \tau$$

We chose $\tau = 1.5$, as it seemed a reasonable level to warrant addition provisioning. When this threshold is surpassed, Eq.2 is reevaluated with the new observed arrival rate, and a new provisioning is determined. An additional important parameter for provisions is β_i in Eq.2. To allow for a favorable implementation of the *umass* algorithm, we chose a β_i such that the initial number of servers were at a good level.

We conducted all experiments presented in Section 4 but we only report here on the browsing experiments due to lack of space. We selected the browsing mix to present because this is the most challenging one for Fastrack, see first column of graphs in Figure 4 that compare Fastrack with the SLOs. By design, we anticipate that Fastrack nets better power efficiency while the *umass* algorithm does better in performance. This is because the *umass* algorithm, more servers are added as increases in the arrival rate are detected, but these servers are not taken away from the application. This inevitably results in better power savings for Fastrack. Figure 7 presents the performance and power savings for the two algorithms as a function of the system load (population) and confirms that indeed Fastrack is superior to the *umass* algorithm for power savings while it does not achieve as good response times. To improve the clarity of the figure, we did not add the SLO targets, but these lines can be seen in Figure 4. The *umass* algorithm detects the first burst and immediately assigns more processors. Bursts are more pronounced in the light load cases (low number of EBs) , so it is not surprising that a burst occurred causes a very high number of servers to be provisioned.

Results for the ordering and shopping mixes are more favorable for Fastrack and are not presented here due to lack of space. In the case of the ordering mix, there are again instances when traffic bursts caused a large number of servers to be brought on. The shopping mix case shows much of the same behavior, but response time does not improve much at all with the addition of servers here, and so this is the clearest case of wasted power for the *umass* algorithm.

6 Related Work

Dynamic resource management and power consumption are emerging as key challenges in data center environments. In response to the increased importance of this problem, there has been a large body of research work on enterprise power management. Many papers in this area, e.g., [3, 9], apply server turn on/off mechanism for power management. These early papers consider a simple single-tier Internet applications that are deployed using web server clusters. In [9], the authors design an algorithm which periodically evaluates whether nodes should be added/removed from the cluster, based on threshold driven policy and the expected performance. The paper [3] considers resource allocation and power management problem in hosting centers by using market-based policies. The proposed framework enables adaptive resource provisioning by dynamic trade-offs of service quality and cost. The authors approximate the resource demand on cluster nodes from the observed historic information. In [4], the authors address power management while meeting the application response SLAs. They employ both techniques for power management: shutting down the servers and voltage scaling (DVS). The authors propose a solution that integrates proactive and reactive mechanisms. The proactive mechanism predicts load for the near future, and uses this information in a stochastic queuing model for server provisioning task. The reactive mechanism is based on feedback control and uses dynamic voltage scaling. The authors present a simple prediction technique based on S-ARMA model. Our approach also relies on proactive and reactive mechanisms, but the proposed proactive mechanism is based on an insightful workload model that significantly improves the accuracy of load prediction.

There is a large body of research (e.g., [5, 14]) devoted to the problem of server consolidation problem with the objective of minimizing the number of servers supporting workloads, and as a result naturally leads to a more efficient power usage. Most of the papers on shared resource pool management employ the "black-box" approach, and hence, are not application-centric. Commonly available resource demand traces are the basis for such management system. However, there are many research papers, e. g., [11, 13, 1], which design dynamic provisioning systems for targeted classes of applications, e. g., multi-tier applications. In [11], the authors discuss *system-workload* context (system and workload characteristics) that impact energy saving policies. Especially interesting are key workload characteristics used by the authors, such as the *load ratio* (defined as the ratio of the average load to the peak load) and the *rate of change in load* in relation to the current load. These characteristics aim to capture the load profile and serve as a very simple characterization of burstiness. In [13], the authors argue that dynamic resource provisioning of multi-tier applications is very different from provisioning of single tier applications(we share the same position). The authors design an analytical model of multi-tier application that practically reflects the required capacity at different tiers for a given workload. The authors employ a combination of predictive models and reactive techniques at different time scales for dynamic provisioning of multi-tier applications. The work presented in this paper is different from the above works in that it focuses on solving the performance/power problem in the more difficult case of bursty workloads. Burstiness *precludes* the use of analytical models that are used in [14] as these models may result in predictions that underestimate the effects of burstiness [7]. The algorithm that we propose here does not depend on specific workload thresholds that trigger allocation/deallocation of resources. In addition to identifying when to allocate *more* resources, we are also effective in the prompt release of under-utilized resources in contrast to other works that release resources only when they are needed by *other* applications [13].

7 Conclusions

Effective resource allocation in multi-tiered system that aims at meeting predefined SLOs while using minimal resources is a challenging problem that is exacerbated by workload burstiness, uneven demands in the various tiers, and the phenomenon of bottleneck switch. We presented Fastrack, a self-adaptive, parameter-free algorithm that detects the start and end of bursty periods and appropriately adjust its configuration parameters to meet SLOs while restaining the usage of additional resources. Extensive experimentation has revealed that Fastrack can effectively operate under various workload conditions and levels of burstiness in the arrivals. Our results uniformly show that Fastrack is a robust, parameter free algorithm that can operate seamlessly in a variety of settings.

In our future work we intend to work on estimating the duration bursts and of quiet periods, to be able to reach better allocation decisions faster. Furthermore, we are also considering the mechanisms to power down/power up a server (versus simply leaving the server idle as we do in this paper) and incorporate them into Fastrack.

References

[1] D. Carrera, M. Steinder, I. Whalley, J. Torres, E. Ayguade. Enabling Resource Sharing Between Transactional and Batch Workloads Using Dynamic Application Placement. Proc. of the 9th ACM/IFIP/USENIX Int. Conf. on Middleware, 2008.

[2] G. Casale, E. Zhang, E. Smirni. KPC Toolbox: Simple Fitting Using Markovian Arrival Processes. Proc. of the 5th International Conf. on Quantitative Evaluation of Systems (QEST 2008), 2008.

[3] J. Chase et al. Managing energy and server resources in hosting centers. In the 18th Symp. on Operating Systems Principles (SOSP), 2001.

[4] Y. Chen et al. Managing server energy and operational costs in hosting centers. Proc. of ACM SIGMETRICS, 2005.

[5] D. Gmach, J. Rolia, L. Cherkasova, G. Belrose, T. Turicchi, and A. Kemper: An Integrated Approach to Resource Pool Management: Policies, Efficiency and Quality Metrics. Proc. of the Intl. Conf. on Dependable Systems and Networks, (DSN), 2008.

[6] D. A. Menasce, L. W. Dowdy, and V. A. F. Almeida. *Performance by Design : Computer Capacity Planning By Example.* Prentice Hall PTR, 2004.

[7] N. Mi, G. Casale, L. Cherkasova, and E. Smirni. Burstiness in multi-tier applications: Symptoms, causes, and new models. In ACM/IFIP/USENIX International Middleware'08, 2008.

[8] N. Mi, G. Casale, L. Cherkasova, and E. Smirni. Injecting realistic burstiness to a traditional client-server benchmark. Proc. of the 6th Intl. Conference on Autonomic Computing (ICAC), 2009.

[9] E. Pinheiro et al. Load balancing and unbalancing for power and performance in cluster-based systems. In Proc. of the Workshop on Compilers and Operating Systems for Low Power (COLP), 2001.

[10] R. Raghavendra, P. Ranganathan, V. Talwar, Z. Wang, X. Zhu. No Power Struggles: A Unified Multi-level Power Management Architecture for the Data Center. In Intl. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS), 2008.

[11] K. Rajamani and C. Lefurgy. On Evaluating Request-Distribution Schemes for Saving Energy in Server Clusters. In the IEEE Intl. Symp. on Performance Analysis of Systems and Software (ISPASS), 2003.

[12] D. Economou, S. Rivoire, C. Kozyrakis, P. Ranganathan. Full-System Power Analysis and Modeling for Server Environments. In Workshop on Modeling, Benchmarking, and Simulation (MoBS), 2006.

[13] B. Urgaonkar, P. Shenoy, A. Chandra, P. Goyal, and T. Wood. Agile Dynamic Provisioning of Multi-tier Internet Applications. ACM Transactions on Adaptive and Autonomous Systems (TAAS), Vol. 3, No. 1, March 2008.

[14] B. Urgaonkar, P. Shenoy, and T. Roscoe. Resource Overbooking and Application Profiling in a Shared Internet Hosting Platform ACM Transactions on Internet Technologies (TOIT), vol 9, N. 1, 2009.