Operational Analysis of Parallel Servers

Terence Kelly — *HP Labs*

Kai Shen — *U. Rochester CS*

Alex Zhang — *HP Labs*

Christopher Stewart — *HP Labs & U. Rochester CS*

Abstract

Multicore processors promise continued hardware performance improvements even as single-core performance flattens out. However they also enable increasingly complex application software that threatens to obfuscate application-level performance. This paper applies operational analysis to the problem of understanding and predicting application-level performance in parallel servers. We present operational laws that offer both insight and actionable information based on lightweight passive external observations of black-box applications. One law accurately infers queuing delays; others predict the performance implications of expanding or reducing capacity. The former enables improved monitoring and system management; the latter enable capacity planning and dynamic resource provisioning to incorporate application-level performance in a principled way. Our laws rest upon a handful of weak assumptions that are easy to test and widely satisfied in practice. We show that the laws are broadly applicable across many practical CPU scheduling policies. Experimental results on a multicore network server in an enterprise data center demonstrate the usefulness of our laws.

1 Introduction

The era of ubiquitous parallel computing has arrived. Chip multiprocessing and simultaneous multithreading have already brought us single-socket processors that present dozens of logical CPUs to operating systems and application software, and computers with scores of cores are imminent. For network servers, e.g., in enterprise data centers, today's technology trends carry profound implications. Multicore processors and virtualization will enable massive consolidation and "datacenter-on-chip" deployments of applications that are locally distributed across clusters today [16]. Modern multicore processors furthermore offer increasingly fine control over power-performance tradeoffs [14, 29]. Meanwhile, solid-state storage promises to revolutionize hardware and software architectures [12]. Unfortunately these trends, together with the growing complexity and opacity of applications, threaten to obfuscate performance in commercially important computing systems. Understanding performance remains imperative because we must balance it against other considerations such as power consumption and hardware cost. More than ever, we require performance analysis techniques that are practical, general, and accessible to real-world decision makers: They must work with black-box production applications, for which source code access, invasive instrumentation, and controlled benchmarking/profiling are seldom permitted; they must rely only on weak assumptions that are easy to test and widely satisfied in the field; and they must be easy for the average practitioner to learn and apply.

This paper presents three parallel performance laws that provide actionable insight using only lightweight passive external observations of black-box production applications. The Occupancy Law infers processor utilization and queuing delays from readily available observations of arbitrary workloads. The Capacity Expansion and Reduction Laws predict the application-level performance consequences of changing the number of processors available to an application while holding workload fixed. All three are *operational* laws because they involve only directly measurable quantities (as opposed to, e.g., probabilistic assumptions) [7]. Classical operational laws such as Little's Law [23] are the foundation of traditional computer systems performance analysis [21, 26]. Our results address the new challenges forced upon us by the multicore revolution.

The analyses that establish and characterize our operational laws are nontrivial but the laws themselves are readily accessible to nonspecialists, requiring neither esoteric assumptions nor extraordinary training. Our laws have several important uses: The Capacity Adjustment Laws allow long-term capacity planning and short-term dynamic resource allocation to incorporate application-level performance in a principled way, and the Occupancy Law enables qualitative improvements in application measurement, monitoring, and management. Although our practical discussions emphasize network servers with request/reply workloads, our results straightforwardly generalize to other contexts, and technology trends are making the formal model to which they apply increasingly relevant to real-world computing.

The remainder of this paper is organized as follows: Section 2 describes our system model and Sections 3 and 4 present our performance laws. Section 5 empirically validates our theoretical results and explores their practical application to a real multicore network server in an enterprise data center. Section 6 surveys related work and Section 7

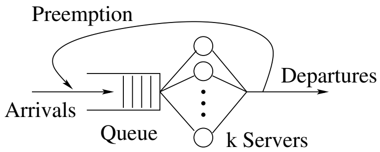


Figure 1. System model.

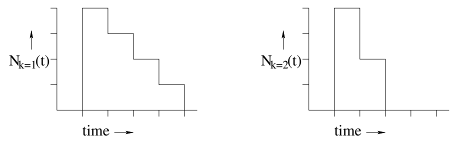


Figure 2. Occupancy curves.

concludes. A companion paper presents complementary results on processor speed scaling [35].

2 System Model

We consider a single-queue station with k servers (Figure 1) and jobs with heterogeneous service demands. Some of our results place no restrictions on job arrivals and apply equally to batch, open, closed [21], and semi-open [34] workloads; other results hold only for open arrivals. Similarly some of our results assume identical servers but others apply to heterogeneous systems. We permit but do not require preemption: the scheduler may alternately serve and enqueue a job until its service demand is satisfied. We make no assumptions about the information that guides the scheduler's decisions; e.g., we permit but do not require the scheduler to exploit offline knowledge of future arrivals. Two assumptions hold throughout this paper:

Assumption 1 *Work conservation: no server is idle unless the queue is empty.*

Assumption 2 *Serial jobs: a job occupies exactly one queue position or server at any instant.*

Our system model is reasonable for parallel network servers handling CPU-intensive request/reply workloads. Parameter k may represent the number of physical processors/cores in a computer, or the number effectively available to the application (the latter may be less than the former, e.g., due to soft concurrency limits). Assumption 1 nearly always holds in practice for CPU scheduling. Multi-level scheduling, e.g., involving virtualization, poses no inherent difficulties: our model applies regardless of whether service demands are mapped onto CPUs/cores by a conventional operating system, a virtual machine monitor, or some combination of the two. We require only that overall scheduling be work-conserving, which is true for default configurations of mainstream OSes and VMMs. Assumption 2 implies that the execution of a single job is not parallelized. This is true for request handling in most network servers.

Our model does not include blocking at auxiliary queues. Blocking can occur in today's network servers if a request must read uncached data from disk, synchronously write to non-volatile storage for durability, perform network I/O (e.g., to invoke subsidiary services via RPCs), or queue at mutexes. Technology trends have substantially reduced blocking for many applications and workloads, and these trends will accelerate as the multicore era unfolds. Large main memories accommodate the working sets of most applications, eliminating blocking storage reads in the warm steady state. Non-volatile caches in storage systems, increasingly popular as performance boosters [40], reduce the latency of synchronous durable writes below the total time cost of OS context switches [22], eliminating the need for blocking writes. Transactional memory promises to supplant locks in future applications, eliminating queueing at locks. Finally, several trends conspire to eliminate internal network I/O within complex modern applications: Today's applications locally distribute software components across hosts in a cluster, making network I/O critical to performance [39]. However the cost advantages of host consolidation coupled with virtualization technology point toward "datacenter-on-chip" app deployments on multicore processors [16], eliminating inter-component network I/O. Taken together, these trends suggest that an increasingly broad range of real computing systems will resemble the model of Figure 1 as the multicore era progresses.

3 The Occupancy Law

The difference between a job's departure and arrival times is its response time, which is the sum of queuing time and service time. The Occupancy Law allows us to estimate aggregate queuing and service times based on operational analysis of cumulative arrivals and departures at a black-box system described in Section 2. It requires no information about scheduling within the system or jobs' individual or aggregate service demands.

Consider a system initially empty at time $t = 0$. Let $A(t)$ and $D_k(t)$ respectively denote the cumulative number of job arrivals and departures up to time t, where the subscript reminds us that $D_k(t)$ may depend on the number of servers k. Let $N_k(t) \equiv A(t) - D_k(t)$ denote the number of jobs present in the system at time t. We refer to a graphical representation of $N_k(t)$ as a *k-server occupancy curve*. Figure 2 illustrates two possible occupancy curves $N_{k=1}(t)$ (left) and $N_{k=2}(t)$ (right) for one- and two-server stations handling the same workload: four jobs, each requiring unit service, arrive at $t = 1$ and are scheduled non-preemptively. Three remarks on occupancy curves: First, the area under the occupancy curve equals the sum of response times across all jobs; this observation sometimes accompanies graphical illustrations of Little's Law [7, 23]. Second, the shape of the

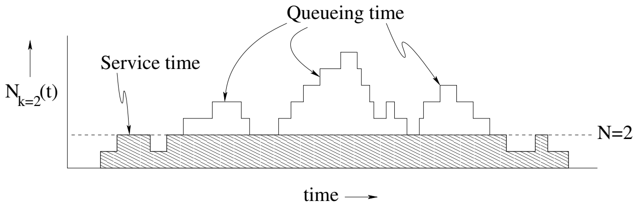


Figure 3. Occupancy Law example, $k = 2$.

occupancy curve may depend upon the queue discipline as well as the number of servers. Finally, we can compute the occupancy curve even if we cannot associate specific departures with corresponding arrivals.

Assumptions 1 and 2 imply that the number of servers busy at time t is the lesser of $N_k(t)$ and k. Therefore if we draw a horizontal line through the k-server occupancy curve at $N = k$, the area beneath both this line and the occupancy curve itself equals aggregate service time for the workload and the area above equals aggregate queueing time, as illustrated in Figure 3. The $N = k$ line furthermore separates service and queueing times *during any interval*, as summarized in our first result.

Result 1 ***The Occupancy Law***. *During any interval* $[T, T']$ *aggregate service time equals* $\int_T^{T'} \min\{N_k(t),\ k\}\,\mathrm{d}t$ *and aggregate queueing time equals* $\int_T^{T'} \max\{N_k(t) - k,\ 0\}\,\mathrm{d}t$*. The sum of the two equals* $\int_T^{T'} N_k(t)\,\mathrm{d}t$ *and is the interval's contribution to aggregate response time.*

The Occupancy Law is an *operational* law because its inputs are directly measurable quantities [7]. (By contrast, stochastic queuing models involve assumptions about the probability distributions of job arrivals and service demands.) Unlike the classical operational laws, the Occupancy Law provides the relative magnitudes of service and queuing times in a black-box system. Furthermore, unlike asymptotic and balanced-system "bounding analysis" approximations [21], it yields exact performance quantities of interest. The Occupancy Law does not assume identical servers and therefore applies to heterogeneous parallel computing systems, including heterogeneous multicore processors [20]. The Occupancy Law holds regardless of fine-scale processor phenomena, e.g., involving caching. Finally, note that even the very weak assumption of flow balance is not required to establish the Occupancy Law.

In practical terms, the Occupancy Law provides insights not readily available from conventional system- or application-level measurements. Today's system monitoring tools provide only coarse-grained aggregate resource utilization measurements at fixed, pre-specified intervals (typically 5 minutes) [5, 11, 33], but the Occupancy Law applies to arbitrary, variable-length intervals. Intervals as short as 200 ms are not unreasonable for environments such as the data center used for our experiments. Analyzing the occupancy curve in each interval of constant $N_k(t)$ and combining the results yields the distributions of server utilizations and queue lengths during any period of interest. Whereas conventional measurement tools reside on the computer being measured and incur performance overheads, the Occupancy Law allows us to infer utilizations and queueing delays via *lightweight, passive, external* observations of black-box applications.

Application-level transaction logs may record per-request response times, but these are available only after requests have completed and are inaccurate under heavy load because they do not reflect queueing delays between packet arrivals and application-level handling [36]. By contrast, the Occupancy Law estimates both utilization and queueing, and does so *even for requests that have not completed*; it is therefore better suited to real-time monitoring. Conventional data center management tools alert human operators when resource utilizations or response times exceed specified thresholds, but the former can fail to detect unresponsiveness and the latter alerts are not actionable if response times consist largely of service times. The Occupancy Law enables more sophisticated alerts based on the *relative magnitudes* of queuing and service times. Operational analysis of an occupancy curve provides accurate, high-resolution insight into both utilization and queuing in unmodified black-box legacy network servers while creating no additional load. The price of this additional insight is modest. Job arrivals and departures can easily be measured at clients, at network servers using kernel packet timestamping facilities [36], or by a network sniffer near the target machine [6]. In a cluster computing context, the same observations can be made by the job dispatcher of a cluster scheduler [30].

4 The Capacity Adjustment Laws

This section presents two operational laws that bound the performance implications of capacity expansion and reduction, i.e., increasing or decreasing the number of servers in the system depicted in Figure 1. A companion paper considers the complementary problem of predicting performance when the *speed* of the servers changes [35]. Given only a k-server occupancy curve, we wish to bound the change in aggregate queuing time that would result if a different number of servers k' handled the same workload. Predicting the performance consequences of capacity change is difficult because both scheduling and the potential for parallelism in the workload influence the outcome, but the occupancy curve seemingly contains no explicit information about either.

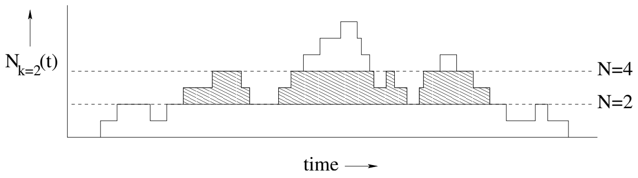


Figure 4. Example, $\mathbf{k = 2}$ and $\mathbf{k' = 4}$.

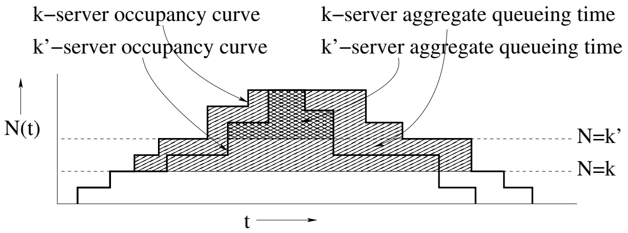


Figure 5. Graphical derivation.

4.1 The Capacity Expansion Law

Consider the example in Figure 4. It is tempting to conclude (falsely) that the shaded area under the occupancy curve $N_k(t)$ between horizontal lines $N = k = 2$ and $N = k' = 4$ equals the reduction in aggregate queueing time that would result from increasing the number of servers from $k = 2$ to $k' = 4$. However, it is easy to generate counterexamples showing that the reduction in aggregate queueing time is sometimes strictly greater. It turns out that the area beneath the occupancy curve and between the $N = k$ and $N = k'$ lines can *bound* the change in queueing time. We first consider capacity expansion ($k' > k$) and begin by introducing an additional assumption.

Assumption 3 *Completion-monotonic scheduling: if $k' > k$, then $D_{k'}(t) \geq D_k(t)$ for all times t (increasing capacity does not reduce cumulative job completions).*

Assumption 3 states that additional servers "do no harm." Below we state and derive the Capacity Expansion Law (which requires Assumption 3). We then show that several widely used scheduling policies are completion-monotonic, i.e., they satisfy Assumption 3, under two additional assumptions.

Result 2 ***The Capacity Expansion Law***. *If the number of servers increases from k to k' ($k' > k$), then aggregate queueing time during the interval $[0,T]$ decreases by at least $\int_0^T \max\{\min\{N_k(t),\ k'\} - k,\ 0\}\,\mathrm{d}t$ (the area beneath the k-server occupancy curve and between the horizontal lines $N = k$ and $N = k'$).*

Derivation When the number of servers increases from k to k', by the Occupancy Law the reduction of aggregate queueing time is $\int_0^T \max\{N_k(t) - k,\ 0\}\,\mathrm{d}t - \int_0^T \max\{N_{k'}(t) - k',\ 0\}\,\mathrm{d}t$. Since server scheduling is completion-monotonic (Assumption 3), we have $N_k(t) \geq N_{k'}(t)$ for every time t—the k'-server occupancy curve is never higher than the k-server occupancy curve. Therefore:

$$
\begin{aligned}
&\int_0^T \max\{N_k(t) - k,\ 0\}\,\mathrm{d}t - \int_0^T \max\{N_{k'}(t) - k',\ 0\}\,\mathrm{d}t \\
\geq &\int_0^T \max\{N_k(t) - k,\ 0\}\,\mathrm{d}t - \int_0^T \max\{N_k(t) - k',\ 0\}\,\mathrm{d}t \\
= &\int_0^T \max\{\min\{N_k(t),\ k'\} - k,\ 0\}\,\mathrm{d}t \qquad \blacksquare
\end{aligned}
$$

The Capacity Expansion Law also admits a graphical derivation shown in Figure 5. Aggregate queueing time with k servers is the area beneath the k-server occupancy curve but above the horizontal line $N = k$ (light shading). With k' servers, aggregate queueing time is the area beneath the k'-server occupancy curve but above $N = k'$ (heavy shading). Since the k'-server occupancy curve is never higher than the k-server occupancy curve (by Assumption 3), aggregate queueing time with k' servers is no greater than the area beneath the k-server occupancy curve but above the $N = k'$ line. Thus, the reduction in aggregate queueing time when the number of servers increases from k to k' is bounded from below by the area beneath the k-server occupancy curve and between the horizontal lines $N = k$ and $N = k'$.

4.2 Completion-Monotonic Scheduling

We now return to completion monotonicity (Assumption 3). This property is intuitive for schedulers that strive to make effective use of resources and it holds for many common schedulers. In this paper we show that all static priority-based preemptive schedulers (including First-Come-First-Served and Shortest-Job-First) are completion monotonic (Theorem 1), as is fine-grained Processor Sharing or Round Robin with infinitesimal timeslices (Theorem 2). In both cases, we show that every job finishes at the same time or earlier after the capacity expansion, which is a sufficient (but not necessary) condition for completion monotonicity. Two additional assumptions formalize the notion that changing the number of servers does not change the workload and restrict attention to homogeneous systems with identical servers.

Assumption 4 *Open arrivals: Jobs arrive according to an open arrival process.*

Assumption 5 *Fixed service demands: Job service demands are fixed (though not necessarily identical). A job's service demand is independent of the number, nature, and location of other jobs in the system, the number of servers, the server(s) on which the job runs, and the scheduler's decisions.*

Theorem 1 ***Completion monotonicity under Priority scheduling.*** *If scheduler S assigns a static priority to each job (where the priorities form a total order) and S preemptively schedules jobs by priority, then S is completion-monotonic.*

Proof of Theorem 1: Consider a workload with n jobs ordered by priority in scheduler S: T_1 (highest priority), T_2, $\cdots$, T_n (lowest priority). Under scheduler S, we show by induction that no jobs finish later when the number of servers increases from k to k'. Note that under open arrivals (Assumption 4), every job arrives at the same time on k and k' servers. Regardless of the number of servers, job T_1 always starts as soon as it arrives and runs without interruption. Following Assumption 5, T_1 finishes at the same time on k and k' servers. If jobs T_1, T_2, $\cdots$, T_i do not finish later when the number of servers increases from k to k', we show that job T_{i+1} also does not finish later. Under priority scheduling, job T_{i+1} runs when the number of arrived but not yet completed higher-priority jobs is less than the number of servers. Therefore all runnable time periods for job T_{i+1} on k servers must also be runnable on k' servers. From Assumption 5, we know that job T_{i+1} finishes, on k' servers, no later than it does on k servers. ■

Theorem 2 ***Completion monotonicity under Processor Sharing.*** *Let k be the number of servers in the system. Over any time period of length p with a constant number of active jobs n, scheduler S assigns* $\min\{p \cdot \frac{k}{n}, p\}$ *service time to each of the n active jobs. Then any change in the number of servers is always completion-monotonic under S.*

Let $L_t^k(T)$ denote the amount of service job T has received up to time t when k is the number of servers. Let the total service demand of job T be l (recall that by Assumption 5 a job's service demand is independent of the number of servers and the particular ones that serve it). Therefore $L_t^k(T) = l$ if job T has completed by time t. We define a capacity increase from k to k' servers to be *per-job completion-monotonic* at time t if every job has received no less service by time t in a system with k' servers than with k servers (i.e., $L_t^k(T) \leq L_t^{k'}(T)$ for all jobs T). We introduce a lemma before proving Theorem 2. Recall that by Assumption 4 every job arrives at the same time on k and k' servers.

Lemma 1 *If the capacity increase is per-job completion-monotonic at time t_{pstart} and there is no new job arrival after t_{pstart} but before t_{pend}, then the increase is per-job completion-monotonic at any time t in $[t_{pstart}, t_{pend}]$ under the processor-sharing scheduling S.*

Proof of Lemma 1: When the system has k servers, those jobs that have not completed by t_{pstart} but will complete by t_{pend} are: $T_1, T_2, \cdots, T_m$ (in order of their completion times). Let $t_1 \leq t_2 \leq \cdots \leq t_m$ be their corresponding completion times. Let $T_{m+1}, T_{m+2}, \cdots, T_n$ be remaining active jobs at t_{pstart} (which will stay active at t_{pend}). We now prove Lemma 1 by induction. First we show that the capacity increase is per-job completion-monotonic at any time t in $[t_{\text{pstart}}, t_1]$ under scheduler S. Since there are n jobs during the time period $[t_{\text{pstart}}, t_1]$ when the system has k servers, for any job T and any time t in this period, we have:

$$L_t^k(T) = L_{t_{\text{pstart}}}^k(T) + \min\{(t - t_{\text{pstart}}) \cdot \frac{k}{n}, t - t_{\text{pstart}}\}$$

Since there are no more than n jobs during $[t_{\text{pstart}}, t_1]$ with k' servers, for any job T, we have:

$$\begin{aligned} L_t^{k'}(T) &\geq L_{t_{\text{pstart}}}^{k'}(T) + \min\{(t - t_{\text{pstart}}) \cdot \frac{k'}{n}, t - t_{\text{pstart}}\} \\ &\geq L_{t_{\text{pstart}}}^{k}(T) + \min\{(t - t_{\text{pstart}}) \cdot \frac{k}{n}, t - t_{\text{pstart}}\} = L_t^k(T) \end{aligned}$$

If the capacity increase is per-job completion-monotonic at any time t in $[t_{\text{pstart}}, t_i]$ (here $1 \leq i \leq m$), below we show that it is also per-job completion-monotonic at any time t in $[t_{\text{pstart}}, t_{i+1}]$ (or $[t_{\text{pstart}}, t_{\text{pend}}]$ when $i = m$). Since there are $n - i$ jobs during the time period $[t_i, t_{i+1}]$ when the system has k servers, for any job T and any time t in this period, we have:

$$L_t^k(T) = L_{t_i}^k(T) + \min\{(t - t_i) \cdot \frac{k}{n-i}, t - t_i\}$$

Since the capacity increase is per-job completion-monotonic at time t_i, at least i jobs have completed by t_i when the system has k' servers. Consequently there are no more than $n - i$ jobs during the time period $[t_i, t_{i+1}]$ when the system has k' servers. For any job T, we have:

$$\begin{aligned} L_t^{k'}(T) &\geq L_{t_i}^{k'}(T) + \min\{(t - t_i) \cdot \frac{k'}{n-i}, t - t_i\} \\ &\geq L_{t_i}^{k}(T) + \min\{(t - t_i) \cdot \frac{k}{n-i}, t - t_i\} = L_t^k(T) \quad ■ \end{aligned}$$

Proof of Theorem 2: Let $t_1 \leq t_2 \leq \cdots \leq t_n$ be the arrival times of all jobs. At t_1, no job has made any progress regardless of the number of servers in the system so a capacity increase is per-job completion-monotonic at time t_1. According to Lemma 1, we can show that it is also per-job completion-monotonic at any time up to t_2. Step by step, we can further show that a capacity increase is per-job completion-monotonic at any time up to $t_3, t_4, \cdots$. Consequently we know that a capacity increase is per-job completion-monotonic at any time instant under the processor-sharing scheduling S. This also means every job finishes on k' servers no later than it does on k servers. ■

There exist scheduling policies that do *not* satisfy completion monotonicity. Some are contrived pathological policies, e.g., the policy that employs Shortest Job First when k

servers are available and Longest Job First for $k' > k$. However a non-deterministic scheduler that employs randomization, for example, may violate the completion-monotonicity property. We speculate that most commonly used deterministic scheduling policies have the intuitive "do no harm" property of completion monotonicity. We leave the proofs for additional schedulers to future work.

4.3 Tightness of the Bound

The Capacity Expansion Law defines a lower bound R_{LB} on the reduction in aggregate queuing time when the number of servers increases. How tight is this bound? In the absence of restrictions on problem parameters, it is possible to construct examples in which the ratio between the actual reduction in queuing time and R_{LB} is arbitrarily high. However if the number of jobs in the system is bounded—i.e., if $N_k(t) \le \hat{N}$ at all times t—then the ratio between the actual queueing time change and R_{LB} is limited to $\frac{\hat{N}-k}{k'-k}$. This is easy to show because 1) the reduction of aggregate queueing time is no more than the total k-server aggregate queueing time, and 2) the area beneath any occupancy curve and between two horizontal lines $N = y$ and $N = y+1$ is monotonically non-increasing when y increases. This result suggests that the bound is tighter when the system is less congested. Bounding $N_k(t)$ is not a restrictive assumption; if the number of jobs in the system grows without bound, the system is simply oversaturated. Like the Occupancy Law, the Capacity Expansion Law does not require the assumption of flow balance, but flow balance ensures tighter bounds.

An obvious *upper* bound on the reduction in aggregate queueing time is to reduce the queueing time to zero. We have not yet established tighter general upper bounds, but we know that there does not exist any upper bound in the form of a constant times R_{LB}. This follows the pessimistic result on the general tightness of using R_{LB} as a lower bound.

In practical terms, the Capacity Expansion Law sometimes assures you that you *definitely should expand* capacity. Sometimes—e.g., because the performance improvement bound is not tight—it does not recommend any action. Our next result, the Capacity Reduction Law, is symmetric: it sometimes warns you that you *definitely should not reduce* capacity.

4.4 The Capacity Reduction Law

We conclude this section by stating the Capacity Reduction Law, which closely resembles the Capacity Expansion Law in both definition and derivation.

Result 3 ***Capacity Reduction Law***. *If the number of servers decreases from k to k' ($k > k'$), then aggregate queueing time during the interval $[0,T]$ increases by at least $\int_0^T \max\{\min\{N_k(t),\ k\} - k',\ 0\}\,\mathrm{d}t$ (the area beneath the k-server occupancy curve and between two horizontal lines of $N = k$ and $N = k'$).*

Derivation When the number of servers decreases from k to k', by the Occupancy Law the increase of aggregate queueing time is $\int_0^T \max\{N_{k'}(t) - k',\ 0\}\,\mathrm{d}t - \int_0^T \max\{N_k(t) - k,\ 0\}\,\mathrm{d}t$. Since scheduling is completion-monotonic (Assumption 3), we have $N_k(t) \le N_{k'}(t)$ for every time t—the k'-server occupancy curve is never lower than the k-server occupancy curve. Therefore:

$$\begin{aligned}
&\int_0^T \max\{N_{k'}(t) - k',\ 0\}\,\mathrm{d}t - \int_0^T \max\{N_k(t) - k,\ 0\}\,\mathrm{d}t \\
\ge &\int_0^T \max\{N_k(t) - k',\ 0\}\,\mathrm{d}t - \int_0^T \max\{N_k(t) - k,\ 0\}\,\mathrm{d}t \\
= &\int_0^T \max\{\min\{N_k(t),\ k\} - k',\ 0\}\,\mathrm{d}t \qquad \blacksquare
\end{aligned}$$

5 Experiments

We conducted experiments on a real network server in an HP data center to verify the practicality of the Occupancy Law and test the tightness of the Capacity Adjustment Laws' bounds. Our client and server machines are identical HP ProLiant BL460c blades housed together in an HP BladeSystem c7000 enclosure communicating via a Cisco Catalyst Blade Gb Switch 3020. Each blade contains two dual-core Intel Xeon 5160 3 GHz CPUs (i.e, $k = 4$) with 64 KB L1 cache, 4 MB L2 cache, and 8 GB of 667 MHz RAM; both blades run 64-bit SMP Linux 2.6.9-42. The server application is a CPU-bound program invoked through the CGI interface of Apache 2.0.52. We measured request start and end times with a bespoke client workload generator. Our measurements are similar to those that would be collected by a network sniffer located near the Apache machine [6]. Network load and client CPU load were negligible; queueing at the client did not distort measurements. Client-server `ping` RTT is 81 μs and the client application latency of a null request is 1.35 ms; these RTTs are far less than the CPU demands of requests.

5.1 Externally Estimating Queuing Delay

Our first test employed a semi-open workload: each client session is a closed generator and the sessions themselves arrive in an open fashion. This is arguably the most realistic kind of workload for a user-facing network application [34]. Each session generates eight requests with a server CPU demand drawn from $\mathscr{U}[1,2]$ sec and think times drawn from $\mathscr{U}[3,6]$ sec, where $\mathscr{U}[a,b]$ denotes the uniform distribution in the range $[a,b]$. The per-request server CPU

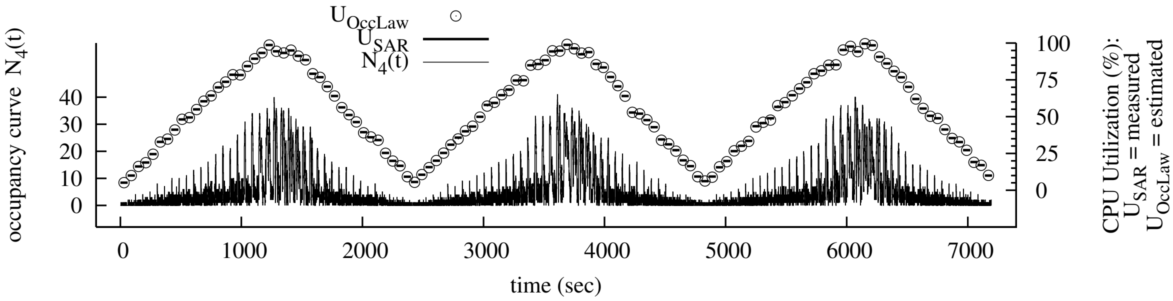
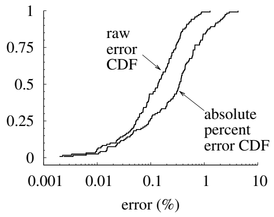


Figure 6. Occupancy curve and server utilization.

demands are comparable to those of today's enterprise applications [37]. Session arrivals are bursty and are constructed such that average server CPU demand follows a sawtooth pattern to mimic diurnal cycles. We ran the test for two hours and measured CPU utilization at the server in 1-minute intervals using the `sar` utility [11]. We measure utilization rather than queueing delay because conventional tools do not report the latter. For our present purposes, either is sufficient to test the accuracy of Occupancy Law estimates. Accurate utilization estimates translate directly into accurate queueing time estimates because estimated service times can simply be subtracted from our externally measured response times.

Figure 6 presents the client-measured occupancy curve $N_4(t)$ (lower series, left scale) and server CPU utilization (upper series, right scale) estimated by the Occupancy Law (circles) and measured by `sar` (short bars). The Occupancy Law's estimate of server CPU utilization is remarkably accurate at all utilization levels, from roughly 5% to over 99.5%.

The figure at right confirms this impression. It presents cumulative distributions over our 120 measurement intervals of two error metrics of the difference between measured utilization $U_{\mathtt{sar}}$ and utilization estimated via the Occupancy Law U_{OccLaw}: raw error $|U_{\mathtt{sar}} - U_{\text{OccLaw}}|$, where both utilizations are expressed as percentages, and normalized error $100 \times \frac{\text{raw error}}{U_{\mathtt{sar}}}$. By either error metric, the Occupancy Law allows us to estimate CPU utilization to within 1% using only lightweight passive external measurements; again, these estimates translate directly into accurate queueing delay estimates. (A similar experiment involving much shorter per-request CPU demands drawn from $\mathscr{U}[100, 500]$ ms yields qualitatively similar conclusions, with median and 97th percentile normalized errors of 0.64% and 4.91%, respectively.)



5.2 Capacity Adjustment & Performance

The Capacity Adjustment Laws offer actionable information to complement the insights provided by the Occupancy Law. By bounding the application-level performance consequences of changing the number of processors/cores available to an application, the Capacity Adjustment Laws provide a principled basis for decisions ranging from processor selection and capacity planning ("how many cores do I need in my next hardware generation to meet my QoS requirements?") to dynamic resource allocation ("is it worth the energy cost to wake up a dormant core?").

We quantify the tightness of the Capacity Adjustment Laws' bounds by replaying a fixed open workload to our network server, varying the number of cores available to the application using the `sched_setaffinity()` system call. The CPU demands of individual requests are drawn from $\mathscr{U}[1, 2]$ sec. We use the same pseudo-random number generator seed on each run and ensure that the workload seen by the server machine is identical on all runs. Over the course of each 90-minute run, arrival rates vary and server CPU utilization fluctuates as shown at right. Note that peak utilization exceeds 25% and therefore the server machine is briefly overloaded when only one of its four cores is available to the application.

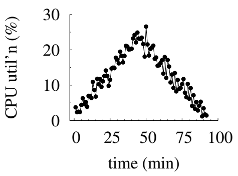


Table 1 presents raw results from our four experimental runs. The table shows $\int_0^T \min\{N_k(t), k'\}dt$, the area under both the occupancy curve and the $N = k'$ horizontal line, for all combinations of k and k'. The $k' = \infty$ column contains aggregate response time, from which we subtract the entry in the $k = k'$ column to obtain the aggregate queuing time shown in the rightmost column. Table 2 shows the change in aggregate queuing time that results from changing the number of cores available to the application from k to k' for all (k, k') pairs. Negative quantities indicate reduction in queueing time and positive quantities indicate increase. For example, Table 1 shows that aggregate queuing delays for $k = 2$ and $k = 3$ are respectively 304.00 sec and 47.28 sec, for a difference of 256.72 sec,

# cores enabled	$k'=1$	$k'=2$	$k'=3$	$k'=4$	$k'=\infty$	aggregate queueing
$k=1$	2721.15	4360.84	5431.77	6176.27	7862.46	5141.31
$k=2$	2139.77	2766.31	2950.09	3014.17	3070.30	304.00
$k=3$	2122.40	2698.98	2841.18	2870.96	2888.45	47.28
$k=4$	2104.71	2654.67	2759.34	2776.92	2782.19	5.27

Table 1. Area (sec) under both occupancy curve and $\mathbf{N = k'}$.

	$k'=1$	$k'=2$	$k'=3$	$k'=4$
$k=1$		−4837.31 (−1639.69)	−5094.04 (−2710.61)	−5136.04 (−3455.12)
$k=2$	4837.31 (626.53)		−256.72 (−183.79)	−298.73 (−247.87)
$k=3$	5094.04 (718.78)	256.72 (142.20)		−42.01 (−29.78)
$k=4$	5136.04 (672.21)	298.73 (122.25)	42.01 (17.58)	

Table 2. Actual change in aggregate queueing delay from capacity adjustment (Capacity Adjustment Law bounds in parentheses).

as shown in the $(k=2, k'=3)$ and $(k=3, k'=2)$ cells of Table 2. Beneath the actual changes in aggregate queuing delay, Table 2 also shows in parentheses the bounds obtained from the Capacity Adjustment Laws. For example, from Table 1 we compute the Capacity Expansion Law bound on the reduction in queuing delay resulting from increasing the number of available cores from $k=1$ to $k'=4$ as $6176.27 - 2721.15 = 3455.12$ sec. The actual reduction in queuing delay is 5136.04 sec, or roughly 27% greater. In general, as we expect, the bounds that we obtain from the Capacity Adjustment Laws are conservative. Furthermore the bounds that we obtain in this experiment are far better than the those guaranteed by the theoretical bounds-tightness results of Section 4.3.

5.3 Improved Scheduling & Tighter Bounds

Our experiments so far have employed the default Linux scheduler, which for our CPU-bound tasks is essentially Round Robin with 100 ms timeslices [4]. As noted in Section 4.3, shorter queues imply tighter Capacity Adjustment Law bounds, so it is reasonable to suspect that the bounds would be tighter if the scheduler and workload were better suited to one another. We conducted an additional experiment to explore this issue.

It is well known that Shortest Remaining Processing Time First (SRPT) scheduling minimizes queuing delay, and Harchol-Balter *et al.* have shown that SRPT scheduling of *network connections* can sometimes significantly reduce average client-perceived latency without unduly penalizing large transfers [13]. We made two changes to our experimental testbed to implement SRPT: we set the CPU demand of all requests to 1.5 sec, and we set the CPU scheduling policy to real-time FIFO (i.e., non-preemptive First-Come First-Served) using the Linux `sched_setscheduler()` system call. The net effect of non-preemptive FCFS scheduling on identical jobs is that the scheduler mimics an SRPT scheduler. In all other respects our experiment was identical to the one that yielded the data in Table 1.

	$k'=1$	$k'=2$
$k=1$		−2104.406 (−1244.616)
$k=2$	2104.406 (611.797)	

The table at right presents our results in the same format as Table 2; $k>2$ cases are omitted because queueing was negligibly small for these runs. Comparing the two tables, we see that the Capacity Adjustment Law bounds are considerably tighter under SRPT. While this experiment does not attempt to imitate real-world network server workloads or CPU scheduling policies, the results are consistent with our expectation that shorter queues due to better scheduling tighten the Capacity Adjustment Law bounds.

6 Related Work

Computer system performance models have long facilitated the exploration of design alternatives [17] and capacity planning [26]. New applications continue to arise, e.g., dynamic resource provisioning [8, 42], performance anomaly detection [19], and server consolidation decision support [37].

Most existing approaches cluster at opposite ends of a complexity/fidelity spectrum. At one extreme, stochastic queuing models [3] yield supremely detailed insight—specifically, the full distributions of performance measures of interest. Stochastic models, however, can require considerable skill to apply and can be brittle with respect to their detailed underlying assumptions [41].

Denning & Buzen popularized the other end of the spectrum: operational analysis, which involves only directly measurable quantities (as opposed to probabilistic assumptions) [7]. Elementary results include the well-known classical operational laws, e.g., Little's Law [23]. Arguably the most powerful and versatile result in all of performance modeling [26], Little's Law is furthermore the foundation of more sophisticated methods including Mean Value Analysis (MVA) [21]. MVA is useful for modeling modern multi-tiered network server applications [24, 42] and has spawned its own extensions and generalizations. For example, Rolia & Sevcik [31] and Menascé [25, 27] generalize MVA to account for queueing at "soft" resources such as mutexes and concurrency limits.

Our contributions represent an intermediate point on the complexity/fidelity spectrum. Our laws are easier to learn and apply than either MVA or stochastic models and they yield more detailed insight than the classical operational laws. All of our results rest upon a handful of straightforward assumptions that are easy to test and that are satisfied in many practical systems of interest. Of course, there is no free lunch: Our laws do not yield all of the insights of the more sophisticated approaches. The former do not supplant the latter but rather complement them by offering a useful new point on the spectrum of performance modeling techniques.

Performance model calibration, e.g., service demand estimation, is a difficult problem in itself. Many proposed methods emphasize enhanced runtime measurement [2] and application profiling via controlled benchmarking [18, 39, 43]. Unfortunately, both are sometimes forbidden in production environments. Recent research emphasizes calibration via lightweight passive observation coupled with sophisticated analyses [37, 38]. The present paper represents another step in the same direction by further reducing the inputs required for the modeling exercise, restricting attention to lightweight passive *external* observations of black-box applications.

Energy efficiency is an increasingly important concern. Fan *et al.* argue that computing systems should consume power in proportion to the useful work they perform [10]. At the level of microprocessors, power/performance tradeoffs involve clock speed adjustment or per-core hibernation [14, 29]. We consider performance vs. number of active processors/cores in this paper and processor speed scaling in a companion paper [35]. Irani & Pruhs survey algorithmic problems involving processor speed scaling and powering down idle processors [15], and Augustine *et al.* present online-optimal power-down algorithms for a single idle processor with multiple sleep states [1]. These contributions differ from ours both in objective and approach: they focus on power savings alone whereas we consider queueing delays, and they employ competitive analysis in contrast to our operational analysis. Rybczynski *et al.* consider energy-conservation strategies that actively alter disk workloads [32]. Our models are more appropriate to CPUs, and our approach does not reshape workloads.

Eager *et al.* explore the tradeoff between speedup and efficiency (average overall utilization) as functions of the number of processors serving a single compound job with known service demand, bounding the extent to which speedup and efficiency can both be poor [9]. Our work is similar in that we too consider the resource/performance tradeoffs inherent in a workload. However we consider multiple jobs and we do not assume that service demands are given.

Our Occupancy Law builds on the rule that the number of busy servers at any instant is the lesser of the number of servers k and the number of jobs in the system N_k, if scheduling is work-conserving. This rule appears in prior stochastic modeling work (e.g., multi-server analysis of the Rate Conservation Law [28]). However, we are not aware of any prior result that provides a precise, fine-grained separation of request waiting time and service time as established by our Occupancy Law. To the best of our knowledge the Capacity Adjustment Laws (and our complementary results on processor speed scaling [35]) are entirely novel.

7 Conclusions

This paper presents three operational laws well suited to the new challenges thrust upon us by the multicore revolution and other current technology trends. Our laws require only readily available inputs and usefully illuminate performance in opaque applications that satisfy a handful of simple assumptions. Our experiments confirm that the Occupancy Law accurately estimates queueing delays and the Capacity Adjustment Laws bound the performance consequences of capacity changes in a real parallel network server. Technology trends promise to improve application-level performance but also threaten to obscure it. Operational analysis can help to preserve our understanding and control of performance as these trends overtake us.

Acknowledgments

We thank Arif Merchant and Ward Whitt for helpful discussions and suggestions, and we thank Hernan Laffitte, Eric Wu, and Krishnan Narayan for technical support that made our experiments possible. This work was supported in part by National Science Foundation grants CCF-0448413, CNS-0615045, and CCF-0621472. Finally, we thank the anonymous reviewers for useful suggestions and pointers.

References

[1] J. Augustine, S. Irani, and C. Swamy. Optimal power-down strategies. *SIAM J. Comput.*, 37(5):1499–1516, 2008.

[2] P. Barham, A. Donnelly, R. Isaacs, and R. Mortier. Using Magpie for request extraction and workload modelling. In *OSDI*, pages 259–272, Dec. 2004.

[3] G. Bolch, S. Greiner, H. de Meer, and K. S. Trivedi. *Queueing Networks and Markov Chains*. John Wiley & Sons, 1998.

[4] D. P. Bovet and M. Cesati. *Understanding the Linux Kernel*. O'Reilly, third edition, Nov. 2005.

[5] H.-P. Corp. HP Performance Manager software, Jan. 2008.

[6] H.-P. Corp. HP Real User Monitor, Jan. 2008. Search for "Real User Monitor" at `http://www.hp.com/`.

[7] P. J. Denning and J. P. Buzen. The operational analysis of queueing network models. *ACM Computing Surveys*, 10(3):225–261, Sept. 1978.

[8] R. P. Doyle, J. S. Chase, O. M. Asad, W. Jin, and A. M. Vahdat. Model-based resource provisioning in a Web service utility. In *Proc. USENIX Symposium on Internet Technologies and Systems (USITS)*, Mar. 2003.

[9] D. K. Eager, J. Zahorjan, and E. D. Lazowska. Speedup versus efficiency in parallel systems. *IEEE Trans. Computers*, 38(3):408–423, Mar. 1989.

[10] X. Fan, W.-D. Weber, and L. A. Barroso. Power provisioning for a warehouse-sized computer. In *ISCA*, pages 13–23, 2007.

[11] S. Godard. `sysstat` utilities for Linux version 8.0.4, Jan. 2008. `http://pagesperso-orange.fr/sebastien.godard/`.

[12] G. Graefe. The five-minute rule 20 years later, and how flash memory changes the rules. In *Proc. Workshop on Data Management on New Hardware (DaMoN)*, June 2007.

[13] M. Harchol-Balter, B. Schroeder, N. Bansal, and M. Agrawal. Size-based scheduling to improve Web performance. *ACM Transactions on Computing Systems*, 21(2):207–233, 2003.

[14] HP, Intel, Microsoft, Phoenix, and Toshiba. Advanced configuration & power interface (ACPI) specification, Oct. 2006. `http://www.acpi.info/spec.htm`.

[15] S. Irani and K. R. Pruhs. Algorithmic problems in power management. *ACM SIGACT News*, 36(2):63–76, June 2005.

[16] R. Iyer et al. Datacenter-on-chip architectures: Terascale opportunities and challenges. *Intel Technical Journal*, 11(3):227–238, Aug. 2007.

[17] R. Jain. *The Art of Computer Systems Performance Analysis*. John Wiley & Sons, 1991.

[18] N. Joukov, A. Traeger, R. Iyer, C. P. Wright, and E. Zadok. Operating system profiling via latency analysis. In *OSDI*, Nov. 2006.

[19] T. Kelly. Detecting performance anomalies in global applications. In *USENIX Workshop on Real, Large Distributed Systems (WORLDS)*, Dec. 2005.

[20] R. Kumar, D. M. Tullsen, P. Ranganathan, N. P. Jouppi, and K. I. Farkas. Single-ISA heterogeneous multi-core architectures for multithreaded workload performance. In *ISCA*, pages 64–75, 2004.

[21] E. D. Lazowska, J. Zahorjan, G. S. Graham, and K. C. Sevcik. *Quantitative System Performance: Computer System Analysis Using Queueing Network Models*. Prentice-Hall, 1984. ISBN 0-13-746975-6.

[22] C. Li, C. Ding, and K. Shen. Quantifying the cost of context switch. In *Experimental Comp. Sci.*, June 2007.

[23] J. D. Little. A Proof of the Queueing Formula: $L = \lambda W$. *Operations Research*, 9(3):383–387, May 1961.

[24] X. Liu, J. Heo, and L. Sha. Modeling 3-tiered web applications. In *MASCOTS*, Sept. 2005.

[25] D. A. Menascé. Two-level iterative queuing modeling for software contention. In *MASCOTS*, Oct. 2002.

[26] D. A. Menascé and V. A. F. Almeida. *Scaling for E-Business: Technologies, Models, Performance, and Capacity Planning*. Prentice Hall, May 2000.

[27] D. A. Menascé and M. Bennani. Analytic performance models for single class and multiple class multithreaded software servers. In *Computer Measurement Group Conf.*, Dec. 2006.

[28] M. Miyazawa. The derivation of invariance relations in complex queueing systems with stationary inputs. *Advances in Applied Probability*, 15(4):874–885, Dec. 1983.

[29] A. Naveh et al. Power and Thermal Management in the Intel Core Duo Processor. *Intel Tech. J.*, 10(2):109–122, May 2006.

[30] Platform Computing. LSF Scheduler. `http://www.platform.com/Products/platform-lsf-family/`.

[31] J. A. Rolia and K. C. Sevcik. The method of layers. *IEEE Transactions on Software Engineering*, 21(8):689–700, Aug. 1995.

[32] J. P. Rybczynski, D. D. Long, and A. Amer. Adapting predictions and workloads for power management. In *MASCOTS*, pages 3–12, Sept. 2006.

[33] R. F. Sauers, C. P. Ruemmler, and P. S. Weygant. *HP-UX 11i Tuning and Performance*. Prentice Hall, 2004.

[34] B. Schroeder, A. Wierman, and M. Harchol-Balter. Open versus closed: A cautionary tale. In *NSDI*, pages 239–252, May 2006.

[35] K. Shen, A. Zhang, T. Kelly, and C. Stewart. Operational analysis of processor speed scaling. In *SPAA*, June 2008. Short paper.

[36] N. Spring, L. Peterson, A. Bavier, and V. S. Pai. Using PlanetLab for network research: Myths, realities, and best practices. In *Proc. USENIX Workshop on Real, Large Distributed Systems (WORLDS)*, pages 67–72, Dec. 2005.

[37] C. Stewart, T. Kelly, and A. Zhang. Exploiting nonstationarity for performance prediction. In *Proc. EuroSys*, Mar. 2007. Also available as HP Labs Tech Report 2007-64.

[38] C. Stewart, T. Kelly, A. Zhang, and K. Shen. A dollar from 15 cents: Cross-platform management for Internet services. In *USENIX Annual Tech*, June 2008.

[39] C. Stewart and K. Shen. Performance modeling and system management for multi-component online services. In *NSDI*, pages 71–84, May 2005.

[40] Texas Memory Systems. RamSan-400 Solid State Disk, Jan. 2008. `http://www.superssd.com/products/ramsan-400/`.

[41] E. Thereska and G. R. Ganger. IRONModel: Robust Performance Models in the Wild. In *SIGMETRICS*, June 2008.

[42] B. Urgaonkar et al. An analytical model for multi-tier Internet services and its applications. In *Proc. ACM SIGMETRICS*, pages 291–302, June 2005.

[43] B. Urgaonkar, P. Shenoy, and T. Roscoe. Resource overbooking and application profiling in shared hosting platforms. In *OSDI*, Dec. 2002.