Understanding, Modelling, and Improving the Performance of Web Applications in Multicore Virtualised Environments

Xi Chen, Chin Pang Ho, Rasha Osman, Peter G. Harrison, William J. Knottenbelt

Department of Computing, Imperial College London, United Kingdom, SW7 2AZ

{x.chen12, c.ho12, rosman, pgh, wjk}@imperial.ac.uk

ABSTRACT

As the computing industry enters the Cloud era, multicore architectures and virtualisation technologies are replacing traditional IT infrastructures. However, the complex relationship between applications and system resources in multicore virtualised environments is not well understood. Workloads such as web services and on-line financial applications have the requirement of high performance but benchmark analysis suggests that these applications do not optimally benefit from a higher number of cores.

In this paper, we try to understand the scalability behaviour of network/CPU intensive applications running on multicore architectures. We begin by benchmarking the Petstore web application, noting the systematic imbalance that arises with respect to per-core workload. Having identified the reason for this phenomenon, we propose a queueing model which, when appropriately parametrised, reflects the trend in our benchmark results for up to 8 cores. Key to our approach is providing a fine-grained model which incorporates the idiosyncrasies of the operating system and the multiple CPU cores. Analysis of the model suggests a straightforward way to mitigate the observed bottleneck, which can be practically realised by the deployment of multiple virtual NICs within our VM. Next we make blind predictions to forecast performance with multiple virtual NICs. The validation results show that the model is able to predict the expected performance with relative errors ranging between 8 and 26%.

Categories and Subject Descriptors

C.4 [**Computing Systems Organisation**]: Performance of Systems—*Modeling Techniques*; G.3 [**Mathematics of Computing**]: Probability and Statistics

Keywords

Benchmarking, Performance Modelling, Multicore, Virtualisation, Web Applications

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ICPE'14, March 22–26, 2014, Dublin, Ireland.
Copyright 2014 ACM 978-1-4503-2733-6/14/03 ...$15.00.
http://dx.doi.org/10.1145/2568088.2568102.

1. INTRODUCTION

Cloud computing has received intensive attention from academia and industry. Two major techniques which are heavily used in this context are virtualisation and multicore architectures. Major cloud service providers, such as Amazon or Microsoft, provide a variety of virtual machines (VMs) that offer different levels of computing power. This computing paradigm provides improved performance, reduced application design and deployment complexity, elastic handling of dynamic workloads, and lower power consumption compared to traditional IT infrastructures [10, 38, 37].

Applications running in cloud environments exhibit a high degree of diversity; hence, strategies for allocating resources to different applications and for virtual resource consolidation increasingly depend on understanding the relationship between the required performance of applications and system resources [39]. To increase resource efficiency and lower operating costs, cloud providers resort to consolidating resources, i.e. packing multiple applications into one physical machine [8]. Understanding the performance of these applications is important for cloud providers to maximise resource utilisation and augment system throughput while maintaining individual application performance targets. Performance is also important to end users, because they are keen to know their applications are provisioned with sufficient resources to cope with varying workloads. Instead of increasing or decreasing the same instances one by one [17, 23], a combination of multiple instances might be more efficient to deal with the burstiness of dynamic workloads [48, 40, 29]. To handle the resource scaling problems, a model that can appropriately express, analyse, and predict the performance of applications running on multicore VM instances is necessary.

There are at least three observations we can make in light of present research. First, not all workloads/systems benefit from multicore CPUs [13, 44] as they do not scale linearly with increasing hardware. Applications might achieve different efficiency based on their concurrency level, intensity of resource demands, and performance level objectives [12]. Second, the effects of sharing resources on system performance are inevitable but not well-understood. The increased overhead and dynamics caused by the complex interactions between the applications, workloads and virtualisation layer introduce new challenges in system management [22]. Third, modelling of low-level resources, such as CPU cores, are not generally captured by models [16, 25] or models are not comprehensive enough to support dynamic resource allocation and consolidation [35, 43].

Many benchmarking studies suggest that each individual core performs differently across the cores of one multiprocessor [24, 32, 21]. Veal et al. [45] and Hashemian et al. [21] observe a CPU single core bottleneck and suggest methods to distribute the bottleneck to achieve better performance. However, most modelling work treats each core of a multicore processor equally by using $M/M/k$ queues [7, 5], where k represents the number of cores. To the best of our knowledge, the problem of modelling the imbalance between cores and the performance of applications in multicore virtualised environment has not been adequately addressed.

This paper presents a simple performance model that captures the virtual software interrupt interference in network-intensive web applications on multicore virtualised platforms. We first conduct some benchmark experiments of a web application running across multiple cores, and then introduce a multi-class queueing model with closed form solution to characterise aspects of the observed performance. Target metrics include utilisation, average response time and throughput for a series of workloads. The key idea behind the model is to characterise the imbalance of the utilisation across all available cores, model the processing of software interrupts, and correctly identify the system bottleneck. We validate the model against direct measurements of response time, throughput and utilisation based on a real system. We take steps to alleviate the bottleneck, which turns out to involve at a practical level the deployment of multiple virtual NICs. Thereafter, we predict the performance of the system with the same workload parameters after tuning the system configuration for improved performance.

The rest of the paper is organised as follows. Section 2 provides context and background. Section 3 presents our testbed setup and performance scaling results. Section 4 introduces our performance model and validates it. Section 5 extends our model for new hardware configurations. Section 6 surveys related work and Section 7 concludes.

2. BACKGROUND

In this section, we briefly recap multicore architectures, then explain the basic steps involved in receiving/transmitting traffic from/to the network and finally discuss the overhead introduced by virtualisation. We aim to understand the most important properties of workloads and systems in order to incorporate them in an analytical model.

Multicore & Scalability: To exploit the benefits of a multicore architecture, applications need to be parallelised [31, 33, 45]. Parallelism is mainly used by operating systems at the process level to provide seamless multitasking [14]. We assume that the following two factors are inherent to web applications which scale with the number of cores: (1) the workload of a web application typically involves multiple concurrent client requests on the server and hence, is heavily parallel; (2) they can exploit the multithreading and asynchronous request services provided by modern web servers (such as Nginx). Each request is usually processed by a separate *thread* and threads can run simultaneously on different CPUs. As a result, modern web servers can efficiently utilise multiple CPU cores. However, scalability of web servers is not in practice linear as other factors, such as sharing cache between cores, communication overhead, call-stack depth, synchronization between threads, or sequential work-flows [29, 45, 8, 24] limit the performance.

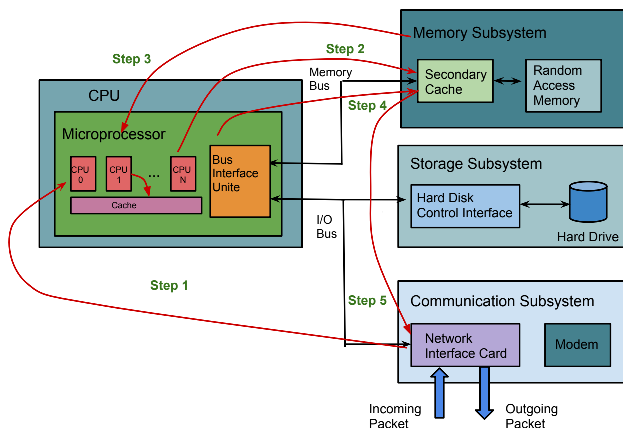


Figure 1: Context Switching Inside a Multicore Server

Linux Kernel Internals & Imbalance of Cores: Modern computer architectures are interrupt-driven. If a device, such as a network interface card (NIC) or a hard disk, requires CPU cycles, it triggers an interrupt which calls a corresponding handler function. As we look at web applications, we focus on interrupts generated by NICs. As packets from the network arrive, the NIC stores these in an internal packet queue and generates an interrupt to notify the CPU to process the packet. By default, an interrupt is handled by a single CPU (usually *CPU 0*). Figure 1 illustrates the process of passing a packet from the network to the application then sending a response back to the network (step 1 to 5). The NIC driver copies the packet to memory and generates a *hardware interrupt* to signal the kernel that a new packet is readable. A previously registered interrupt handler is called which generates a *software interrupt* to push the packet down to the appropriate protocol stack layer or application (step 2) [47]. By default, a NIC software interrupt is handled by *CPU 0 (core 0)* which induces a non-negligible load and, as processing rates increase, creates a major bottleneck for web applications (*interrupt storm*) [45].

Virtualisation & Hypervisor Overhead: Since we focus on the performance modelling of web applications running in virtualised environments, the relationship between performance cost and virtualisation overhead must be taken into account. The virtualisation overhead greatly depends on what guest workloads are doing on the host hardware [15]. With technologies like VT-x/AMD-V and nested paging, CPU-intensive guest code runs at very close to 100% native speed while I/O might take considerably longer due to virtualisation [15]. For example, Barham et al. [3] show that the CPU-intensive SPECweb99 benchmark and the I/O-intensive Open Source Database Benchmark suite (OSDB) perform differently in native Linux and XenoLinux (based on the Xen hypervisor). PostgreSQL in OSDB places considerable load on the disk resulting in many protection domain transitions which is reflected in the substantial virtualisation overhead. SPECweb99 on the other hand, does not require these transitions and hence performs 99.2% of the performance of the bare machine.

Considering the aspect of virtualisation overhead is an important factor for building accurate performance models. However, we include it into parameters in this work and will not further discuss it but rather leave this to future research.

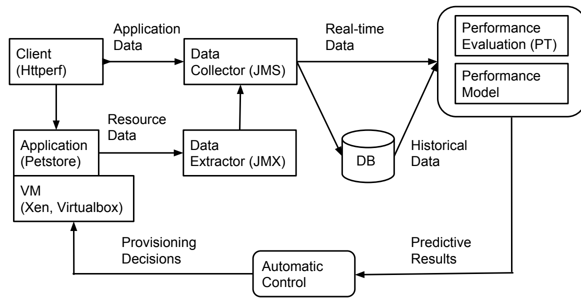
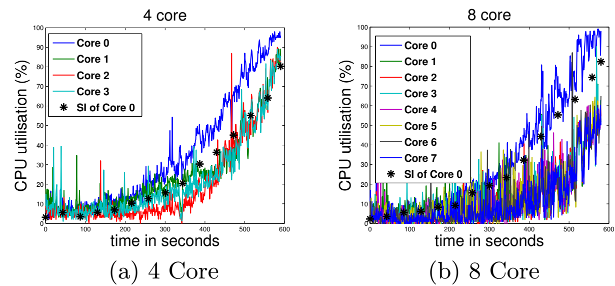
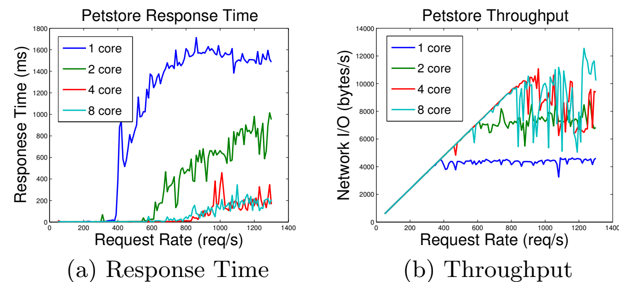


Figure 2: Testbed Architecture



(a) 4 Core (b) 8 Core

Figure 3: CPU Utilisation and Software Interrupt Generated on CPU 0 of 4 Core and 8 Core VM Running the Petstore Application on VirtualBox



(a) Response Time (b) Throughput

Figure 4: Response Time and Throughput of 1 to 8 Core VMs Running the Petstore Application on VirtualBox

3. BENCHMARKING

In this section, we conduct an initial benchmarking experiment to study the impact of multiple cores and software interrupt processing on a common HTTP-based web application. Requests do not involve database access and hence, no disk I/O is required during a response. Our application is the Oracle Java Petstore 2.0[1] which uses GlassFish[2] as the HTTP server. We run the Petstore application on VirtualBox and Xen hypervisor, respectively. The Oracle Java Petstore 2.0 workload is used to expose the VM to high HTTP request volumes which cause intensive CPU activity related to processing of input and output network packets as well as HTTP requests. Autobench[3] was deployed to generate the HTTP client workload.

Testbed Infrastructure. We set up two virtualised platforms: Xen and Virtualbox, using the default configurations. The hypervisors are running on an IBM System X 3750 M4 with four Intel Xeon E5-4650 eight-core processors at 2.70GHz to support multicore VM instances comprising 1 to 8 cores. The server has a dual-port 10 Gbps Ethernet physical network interface card (pNIC), which can operate as a virtual 1 Gbps Ethernet NIC (vNIC). The physical NIC interrupt handling is distributed across the cores, providing maximum interrupt handling performance. The machine is equipped with 48 GB memory and connected to sockets with DDR3-1333MHz channels. Other resources (e.g. disk and network bandwidth) are over-provisioned.

Testbed Setup. The system used to collect the performance data of our tests consists of several components as shown in Figure 2. The *data collector* (Java Message Service) extracts a set of application statistics, e.g. response time and throughput. This is combined with the output of the *data extractor* (Java Management Extension), which provides hardware characteristics, i.e. utilisation of each core of the VM, memory bandwidth, etc. The data collector can either feed this data directly to the *performance evaluator* or store it a database for future analysis. The performance evaluator is based on the concept of Performance Trees [42, 11], which translate the application and system characteristics into parameters that can be directly used by our performance model. The performance model is then analysed and performance indices of the system are derived and compared to actual measurements. The *automatic controller* optimises the resource configuration for specific performance targets. The system is designed for both on-line and off-line performance evaluation and resource demand estimation, which can be applied in areas such as early stage deployment and run-time management on cloud platforms. In this paper, we do not employ the *automatic controller* in our experiments.

Benchmark. Each server VM is configured with one vCPU with a number of virtual cores (from 1 core up to 8 cores for eight experiments) with 4 GB of memory and one vNIC. To mitigate the effect of physical machine thread switching and to override hypervisor scheduling, each virtual core (vCore) was pinned to an individual physical core. For each experiment, Autobench sends a fixed number of HTTP requests to the server at a specific request rate. The mean request rate incrementally increases for each experiment by 10 req/sec from 50 (e0.02)[4] to 1400 (e0.00071). Figure 3 presents the vCore utilisation for the 4 and 8 core VMs running on Virtualbox at increasing request rates for a total duration of 600s. Figure 4 shows the corresponding response time and throughput for the VM from 1, 2, 4 and 8 cores. The utilisation, response times, and throughput for the Xen hypervisor are not shown; however, they exhibit similar performance trends.

From Figure 3(a) and 3(b), we observe that the utilisation of vCore 0 reaches 90% and 98% at 500 secs (corresponding to 1200 req/sec) for 4 and 8 vCore servers respectively, while the utilisation of the other vCores are under 80% and 60% for the same setup. Figure 4(a) shows that the sys-

[1] `http://www.oracle.com/technetwork/java/index-136650.html`
[2] `https://glassfish.java.net/`
[3] `http://www.xenoclast.org/autobench/`
[4] e0.02 refers to an exponential distribution with a mean interarrival time of 0.02s, `http://www.hpl.hp.com/research/linux/httperf/httperf-man-0.9.txt`

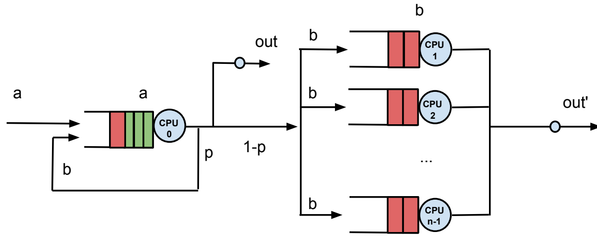


Figure 5: Modelling a Multicore Server Using A Network of Queues

tem becomes overloaded at 400 req/s for a single vCore and at 600 req/s for a dual core. The saturation points for 4 vCores (800 req/s) and 8 vCores (900 req/s) do not reflect the doubling of vCPU capacity. Figure 4 also shows that for the single and dual core cases, the improvement of system throughput asymptotically flattens with a higher request rate and finally saturates around 4000+ bytes/sec and 7000+ bytes/sec. However, the capacity of the VM servers does not increase linearly when the number of vCores changes from 4 to 8 vCores.

When investigating the imbalance of vCore utilisation and lack of scalability across vCores, we have observed that the software interrupt processing causes 90% of the *vCore 0* utilisation, as shown in Figure 3. This saturates *vCore 0* as network throughput increases and it becomes the bottleneck of the system. This bottleneck has also been observed in network-intensive web applications executing on non-virtualised multicore servers [21].

In summary, Figures 3 and 4 show that, when using the default configurations of VirtualBox, the multicore VM server exhibits poor performance *scalability* across the number of cores for network intensive workloads. Additionally, the *utilisation* of each vCore behaves differently across the cores and as vCore 0 deals with *software interrupts*, it saturates and becomes the bottleneck of the system.

4. PROPOSED MODEL

This section describes our proposed model for the performance of a web application running in a multicore virtualised environment. We first give the specification of the model and then present an approximate analytical solution followed by the description of our method to estimate the model parameters. Finally, we validate our model with the testbed from Section 3. Here we refer to vCore 0, ..., vCore $n-1$ as CPU 0, ..., CPU $n-1$.

4.1 Model Specification

Consider a web application running on an n-core VM with a single NIC (eth0), as in our set-up in Section 3.

Modelling Multiple Cores: We model the symmetric multicore processor using a network of queues where each queue (CPU 0, ..., CPU $n-1$) represents a single core (Figure 5). The interrupts generated by eth0 are handled by CPU 0. In a Linux system, one can see that CPU 0 serves an order of magnitude more interrupts than any other core in */proc/interrupts*. We assume that two classes are served under processor sharing (PS) queueing policy in CPU 0; the other queues are M/M/1-PS with single class, which reflects the scheduling policy in most operating systems (e.g. Linux CPU time sharing policy).

When a request arrives from the network:

1. eth0 detects the associated packet and generates an interrupt, which is represented by job class a for CPU 0 (see Figure 5).

2. The interrupt is processed and the packet is forwarded to the application which reads the request. From the model perspective, a class a job turns into a class b job, which reflects that the interrupt triggers the scheduling of a request process.

3. Jobs of class b are either scheduled to CPU 0 with probability p or to one of the remaining CPUs with probability $1-p$. Class a and b jobs are served at service rate μ_1 and μ_2, respectively.

4. After a class b job has been processed, the response is sent back to the client. Note that we naturally capture output NIC interrupts by including them into the service time of class a jobs.

In our model, the arrival of jobs is a Poisson process with arrival rate λ and job service times are exponentially distributed. The system has a maximum number of jobs that it can process as shown in Figure 4, which is also very common for computer systems. For each experiment, an arrival is dropped by the system if the total number of jobs in the system has reached a specified maximum value N.

The preemptive multitasking scheme of an operating system, such as Windows NT, Linux 2.6, Solaris 2.0 etc., utilises the interrupt mechanism, which suspends the currently executing process and invokes the kernel scheduler to reschedule the interrupted process to another core. Otherwise, when a class a job arrives, a class b job executing in CPU 0 could be blocked. However, in a multicore architecture, the blocked processes could experience a timely return to execution by a completely fair scheduler, shortest remaining time scheduler, or some other CPU load-balancing mechanism. To simplify the model, class a and class b jobs are processed separately with a processor sharing policy in CPU 0.

4.2 CPU 0

The proposed queueing model in Figure 5 abstracts the process of serving web requests on a multicore architecture. In this model, CPU 1 to CPU $n-1$ are modelled as standard M/M/1-PS queues, the arrivals to which emanate at CPU 0 as class b jobs. An M/M/1-PS queue is one of the common queue types in the literature [30]. The nontrivial part of the model, however, is CPU 0. CPU 0 processes two classes of jobs, a and b, and the number of jobs can be described as a two dimensional Markov chain $X = (i, j)$, where i is the number of class a job and j is the number of class b job. Figure 6 illustrates the state transitions corresponding to the generator matrix of its stochastic process, $\mathbf{Q}$.

One can compute the stationary distribution numerically by solving the normalised left zero eigenvector of $\mathbf{Q}$. However, as the capacity of the system, N, is a very large number in the real system, the size of $\mathbf{Q}$, is combinatorially large and hence, computing the zero eigenvector becomes infeasible. In the next section, we obtain the stationary distribution of the Markov chain.

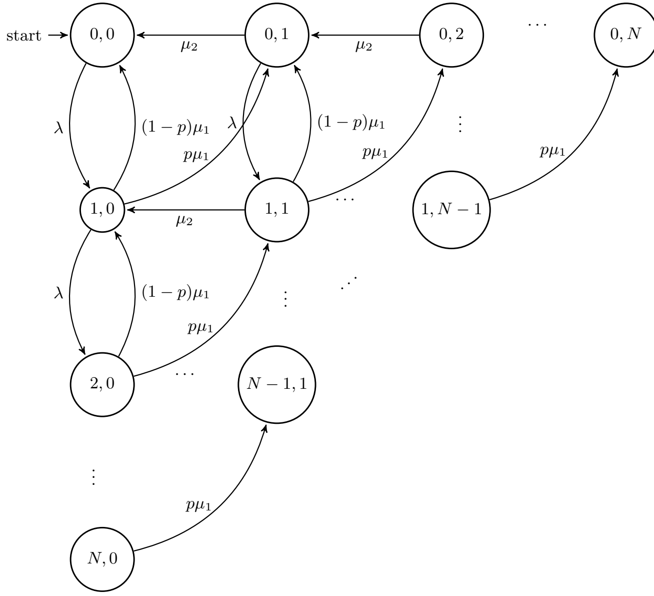


Figure 6: State Transition Diagram for CPU 0

4.2.1 Two-class Markov Chain and its Stationary Distribution of CPU 0

The model specification given in Section 4.1 and the state transition diagram of Figure 6 make the approximating assumption that the total service rate for each class (a and b) does not degrade as the population of the network increases, remaining at the constant values μ_1 and μ_2. Therefore the classes behave independently and the modelled behaviour of CPU 0 is equivalent to a tandem pair of single-class PS queues with rates μ_1 (for class a) and μ_2 (for class b) respectively. The arrival rate at the first queue is λ and at the second $p\lambda$ (since we are considering only CPU 0). This is a standard BCMP network [30] with a population constraint and so has the product-form given in equation (2)[5]. Moreover, the result is a trivial application of the Reversed Compound Agent Theorem (RCAT), see for example [19] [20]. The normalising constant can be obtained as a double sum of finite geometric series and gives the value of $\pi_{0,0}$ shown in equation (1).

We therefore have the following product-form solution:

Proposition 1. *Assuming that a steady state exists, let the steady-state probability of state (i, j) in Figure 6 be denoted $\pi_{i,j}$. Then,*

$$\pi_{0,0} = \frac{(\alpha-1)(\alpha-\beta)(\beta-1)}{\alpha^{N+2}(\beta-1)+\beta^{N+2}(1-\alpha)+\alpha-\beta}, \quad (1)$$

and

$$\pi_{i,j} = \alpha^i \beta^j \pi_{0,0}, \quad (2)$$

where

$$\alpha := \frac{\lambda}{\mu_1} \quad \text{and} \quad \beta := \frac{p\lambda}{\mu_2}. \quad (3)$$

[5] We thank a referee for pointing out that the result was first derived in [41]

Proof. By the proceeding argument, the BCMP Theorem yields,

$$\pi_{i,j} = C\pi_1(i)\pi_2(j).$$

where C is a normalising constant. The marginal probabilities are,

$$\pi_1(k) = \alpha^k \pi_1(0), \pi_2(k) = \beta^k \pi_2(0) \quad \forall k = 0, 1, \ldots, N.$$

Therefore,

$$\pi_{i,j} = C\pi_1(i)\pi_2(j) = C\alpha^i\beta^j\pi_1(0)\pi_2(0) = \alpha^i\beta^j\pi_{0,0}.$$

Normalizing, we have

$$\sum_{i,j}\pi_{i,j} = 1$$

$$\sum_{i,j}\alpha^i\beta^j\pi_{0,0} = 1$$

$$\pi_{0,0}\sum_{i=0}^{N}\sum_{j=0}^{N-i}\alpha^i\beta^j = 1$$

Since

$$\sum_{i=0}^{N}\sum_{j=0}^{N-i}\alpha^i\beta^j = \frac{\alpha^{N+2}(\beta-1)+\beta^{N+2}(1-\alpha)+\alpha-\beta}{(\alpha-1)(\alpha-\beta)(\beta-1)},$$

we obtain

$$\pi_{0,0} = \frac{(\alpha-1)(\alpha-\beta)(\beta-1)}{\alpha^{N+2}(\beta-1)+\beta^{N+2}(1-\alpha)+\alpha-\beta}.$$

□

4.2.2 Average Sojourn Time

Proposition 1 provides the stationary distribution of the Markov chain associated with CPU 0. With that information, we can find the average number of jobs in the system.

Proposition 2. *Let the random variable k denote the total number of jobs at CPU 0. Then,*

$$E(k) = \frac{g(\alpha,\beta)-g(\beta,\alpha)+(\beta-\alpha)(2\alpha\beta-\alpha-\beta)}{[\alpha^{N+2}(\beta-1)+\beta^{N+2}(1-\alpha)+\alpha-\beta](\alpha-1)(\beta-1)}, \quad (4)$$

where $g(x, y) := x^{N+2}(y-1)^2(xN-N-1)$.

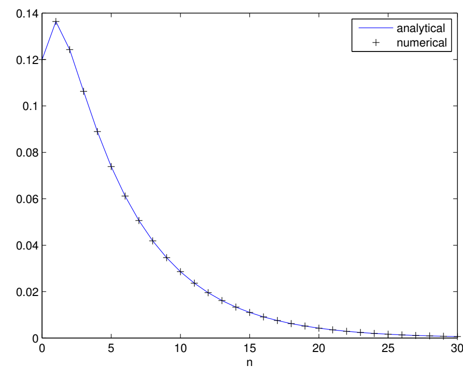


Figure 7: Comparing Numerical and Analytical Solution of E(k)

The proof of Proposition 2 can be found in Appendix A. Figure 7 plots the value of $E(k)$ against N.

Consider again CPU 0 with two job classes a and b. Arrivals will be blocked if the total number of jobs reaches N. The probability function of the total number of jobs at CPU 0 can be calculated as,

$$P_N = P[n_a + n_b = N] = \sum_{i,j}^{i+j=N} \pi_{i,j}$$

Using Proposition 2, a job's expected sojourn time $E(T)$ can be calculated from the long-term average effective arrival rate λ and the average number of jobs $E(k)$, using *Little's Law* for the system as follows:

$$E(T) = \frac{E(k)}{\lambda(1 - P_N)}$$

4.2.3 Average Service Time and Utilisation

PROPOSITION 3. *Let T_s be the random variable denoting the service time of a job γ entering service. The expected service time is*

$$E(T_s) = \frac{1}{\mu_1} n_a^0 + \frac{1}{\mu_1}\frac{\lambda}{\lambda + p\mu_1}(1 - n_a^0) + \frac{1}{\mu_2}\frac{p\mu_1}{\lambda + p\mu_1}(1 - n_a^0), \quad (5)$$

where

$$n_a^0 = \pi_{0,0}\frac{1 - \beta^{N+1}}{1 - \beta}.$$

The proof of the proposition is provided in Appendix B. With the result above, the utilisation of a single core can be derived by the *Utilisation Law*,

$$U = \lambda E(T_s)$$

4.3 Likelihood for Estimating Parameters

The stationary distribution π of the Markov process in Figure 6 with generator matrix $\mathbf{Q}$ and the expected number of jobs $E(k)$ are given in Propositions 1 and 2. There are three corresponding parameters, μ_1, μ_2, and p. We assume that the average response time for a certain request arrival rate λ_i can be estimated from real system measurements. From our previous observations, for example, when a one core system receives 100 req/sec, on average, 2.9% of the CPU utilisation are spent for processing software interrupts while for 200 req/sec, this amount increases to 7.2%. We can obtain μ_1 from utilisation law,

$$\frac{\bar{\lambda}}{\mu_1} = \bar{U}_{si} \quad (6)$$

where $\bar{U}_{si}$ denotes the average utilisation of software interrupts (si) processed by CPU 0 during a monitoring window of size t and $\bar{\lambda}$ is the average λ_i during t. Then the reciprocal of μ_1 is the mean service time of CPU 0 handling si. Note that by using the average utilisation for software interrupts to calculate μ_1, the service time for a class a job includes the service time for *all* software interrupts involved to successfully process the corresponding class b job (see Section 4.1). Here, we find μ_1 to be 3301 req/sec. In the single core case, p is 1. However, for multiple core cases, p can be obtained by the inverse proportion of the utilisation as a load balancing across multiple cores.

Let T_i be the average response time estimated for a certain arrival rate from the model and T_i' be the average time from the real system measurements when the arrival rate is $\lambda_i, i = 1, \ldots, m$. Since the estimated response time T' is the mean of samples, it is approximately a normally distributed random variable with mean T and variance $\frac{\sigma_T^2}{n}$ when the number of samples n is very large [6]. Hence μ_2 can be estimated by maximising the log-likelihood function,

$$\log \prod_{i=1}^{m} \frac{1}{\sqrt{2\pi\sigma_i^2/n_i}} \exp\left[\frac{(T'_i - T_i)^2}{2\pi\sigma_i^2/n_i}\right] \quad (7)$$

Maximising the log-likelihood function above is equivalent to minimising the weighted sum of squared errors:

$$\sum_{i=1}^{m} \frac{(T'_i - T_i)^2}{2\pi\sigma_i^2/n_i} \quad (8)$$

Now the problem of finding the parameters becomes an optimisation problem,

$$\mu_2 = \arg\min_{\mu_2} \sum_{i=1}^{m} \frac{(T'_i - T_i)^2}{2\pi\sigma_i^2/n_i} \quad (9)$$

The optimisation problem can be solved in different ways, such as steepest descent and truncated Newton [6]. We carried out the experiments in the single core case with λ varying from 10 req/s to 500 req/s. For each λ we sent requests from 300 to 30 000 req/s and measured the mean response time and the corresponding standard deviation.

4.4 Combined Model

In the previous section, we analysed the properties of CPU 0, which gives us a better understanding of how its performance is affected by interrupts. To build the entire model, we will combine the previous results of CPU 0 and the results of CPU 1 to CPU $n-1$ given in [30].

For K jobs arriving in the system, we expect Kp of them will stay in CPU 0 and $K(1-p)$ of them will be sent to CPU 1, ..., CPU $n-1$. Given request arrival rate λ, we approximate the arrival rate of jobs at CPU 1, ..., CPU $n-1$ as $\lambda(1-p)$. We further assume that those jobs are uniformly assigned to different cores and so for CPU i, the corresponding (class b) job arrival rate is $\lambda_i = \lambda(1-p)/(n-1)$. Given the service rate of class b jobs is μ_2, the expected number of jobs at these CPUs is $\lambda_i/(\mu_2 - \lambda_i)$, $\forall i = 1, \ldots, n-1$.

	CPU 0	*CPU 1, ..., CPU n-1*
Arrival Rate	λ $p\mu_1$(a $\to$ b)	$\lambda(1-p)$
Service Rate	μ_1 (a) μ_2 (b)	μ_2
Mean Jobs	Proposition 2	$\lambda_i/(\mu_2 - \lambda_i)$

Table 1: Summary of Key Model Parameters

Table 1 gives the brief summary of key model parameters. Let k_i denote the number of jobs in the queue of CPU i; then by Little's Law, the expected sojourn time of a request in the whole system is,

$$E(T_{\text{sys}}) \approx \frac{E(k_0 + k_1 + \cdots + k_{n-1})}{\lambda} = \frac{E(k_0) + E(k_1) + \cdots + E(k_{n-1})}{\lambda}.$$

Values of μ_2 (req/sec)			
1 core	2 core	4 core	8 core
367	345	300	277

Table 2: Likelihood Estimation of the Mean Service Time for Class b Job

4.5 Validation

We validate our model against real system measurements of response time and throughput, focusing on benchmarks running on the VirtualBox hypervisor and using the system set-up of Section 3.

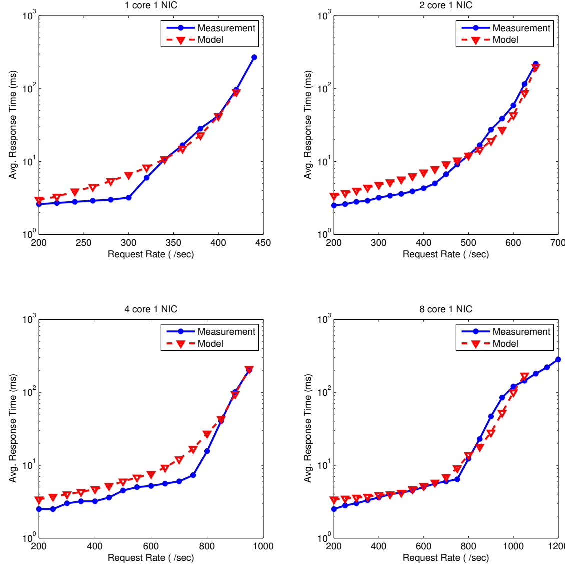


Figure 8: Response Time Validation

Prior to validation, we conducted baseline runs of the benchmark in our test-bed system. For each run, we varied the number of cores and collected information about workload and response time for the parameter estimation (see Section 4.3). The parameters we obtained for class b decrease from 1 core to 8 cores as shown in Table 2. The decreasing μ_2 captures the fact that the web server scales poorly on multiple cores because of (i) the virtualisation overhead; (ii) the inherent problem of multicore, such as context switching overhead. Figure 8 shows the validation of response time.

5. SCALABILITY AND MODEL ENHANCEMENT

In this section, we first describe a set of complementary techniques of system and hardware configurations aimed to prevent the single core bottleneck discussed in Section 3. We apply one of the techniques to increase parallelism and improve performance for multicore web applications. Second, we derive our model for performance under an improved configuration. We then validate our model under the new configurations and show that the results fit with the obtained performance improvements.

5.1 Scalability Enhancement

Multiple network interfaces can provide high network bandwidth and high availability [24, 21, 45]. Enforcing CPU affinity for interrupts and network processing has been shown to be beneficial for SMP systems [45] and the same benefits should apply to virtualised multicore systems. Combining multiple NICs and CPU affinity allows us to distribute the software interrupts for different NICs to different cores and hence mitigate load imbalance. In real systems, installing multiple network interfaces might cause space and power issues; however, in virtualised environments, this can be trivially achieved by using virtual NICs. For our enhanced configuration, we configure multiple vNICs as follows:

- Fix the number of web server threads to the number of cores and assign each web server thread to a dedicated core to avoid the context switching overhead between two or more threads [21].
- Distribute the NIC interrupts to multiple cores by assigning multiple virtual NICs, i.e. vboxnet, to the VM.

5.2 Model Enhancement

Since we model the imbalance of multicore system by distinguishing two different types of queues, we can derive the model for the new configuration by increasing the number of leading two-class queues to match the number of cores m which deal with NIC interrupts. Recall that our baseline model assumes a single core (queue) handling NIC interrupts (job a). Consider the situation when job a comes to m two-class queues (equals to m CPU 0), in which m represents the number of cores that handle NIC interrupts. Then, a class a job transfers into a class b job and either returns to the queue with probability p or proceeds to CPU m, ..., CPU $n-1$ with probability $1-p$.

5.3 Blind Prediction with Previous Parameters and Model Limitations

We apply the model for the enhanced configurations with the same parameters as shown in Table 2. The revalidation results are shown in Figure 9. The results show that the performance of the application improves with the new configurations, and exhibits better scalability. For example, with 4 cores and 1 NIC, the knee-bend in system performance occurs at around 800 req/sec; using 2 NICs this increases to around 1000 req/sec and for 4 NICs to around 1200 req/sec.

The summary of the error found in all validation results of Figure 8 and Figure 9 are shown in Table 3. The average relative modelling error is around 15%. This shows a tendency to decrease with an increasing number of NICs. We see a relative error of e.g. 7.9% and 7.4%, for a 4 core machine with 2 NICs and a 4 core machine with 4 NICs, respectively. Since distributing the NIC interrupts in the real system causes extra context switching overhead, the response time of relatively low intensity workloads (i.e. 200 to 600 req/sec) is round 10-20% higher than that for the default configuration.

We identify several factors that affect model accuracy:

1. The *routing probability* p: we use a simple load balancing policy as we discussed in Section 4.3, which cannot represent the Linux kernel scheduling algorithm used in our testbed, which is a completely fair scheduler. More advanced scheduling policies like O^2 [46] can also not be described with this simple model.

	1 NIC				2 NICs			4 NICs		Overall
Num. of Core	1	2	4	8	2	4	8	4	8	
Response Time	23.8	23.2	25.8	11.3	19.4	7.9	10.3	7.4	14.2	15.9
Throughput	14.1	12.9	13.4	16.5	14.5	11.9	15.6	10.6	16.7	14.02
Util. Core 0 to m-1	10.5	7.9	8.4	9.8	8.4	8.9	12.9	11.4	13.4	10.2
Util. Core m to N-1	-	-9.4	-14.6	-23.7	-	-10.4	- 16.7	-	-17.8	-15.4

Table 3: Relative Error between Model and Measurements (%)

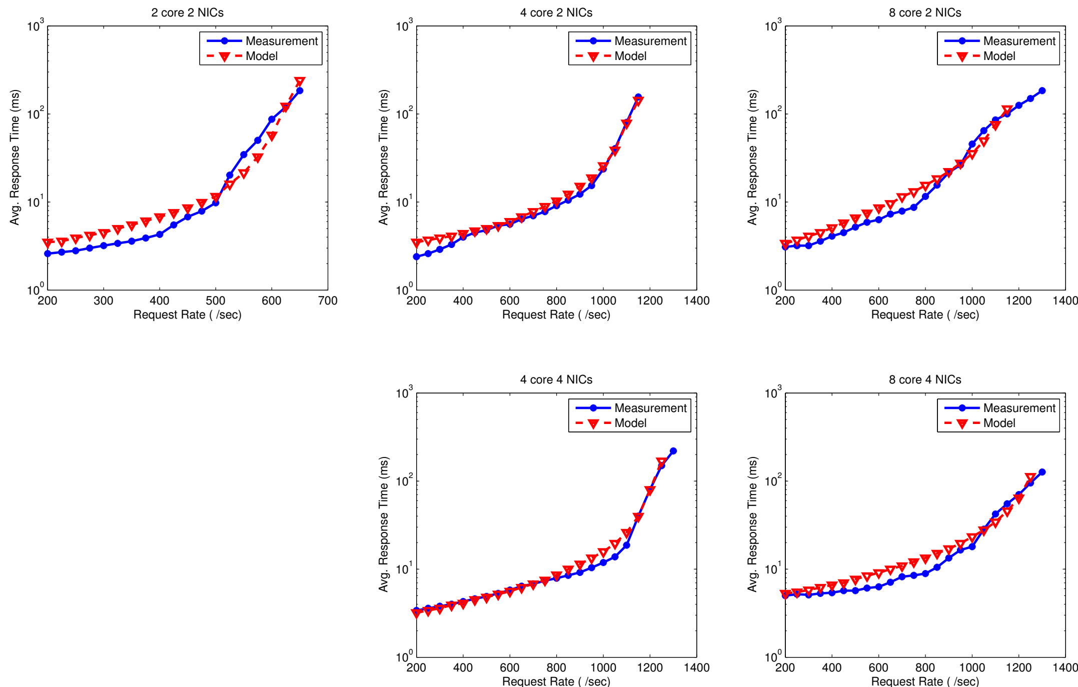


Figure 9: Revalidation of Response Time with Multiple NICs

2. *Interrupt priority*: in general, NIC interrupts (job a) have higher priority than system and user processes (job b). In the single core case, job b is blocked when a new job a arrives. However, in the multicore case, the scheduler will assign it to another core. To simplify the model, we do not consider priorities and interference between job classes a and b.

3. *Context switching overhead*: an operating system executes a context switch by loading a new context, i.e. registers, memory mappings, etc., in one CPU. Though we try to reduce the context switching overhead by assigning each web server thread statically to a distinct core, other context switches, such as register, task, and stack, need to be considered.

4. *Hypervisor overhead*: our model implicitly considers virtualisation overhead, e.g. via the decrease of service rate with increasing number of cores. However, how the overhead of processing requests at the different virtualisation layers has yet to be accounted for.

6. RELATED WORK

Multicore Benchmarking. Veal and Foong [45] identified that scaling of web applications on multicore systems requires distributing the affinity of NICs to different cores. Harji et al. [18] examined in-memory and disk I/O static web application performance on a quad-core system. Their experiments reveal that "the implementation and tuning of web servers is perhaps more important than server architecture". Hashemian et al. [21] characterised the performance of dynamic and static network intensive Web applications on a two quad-core system. The authors have shown that achieving efficient scaling behaviour entails application specific configurations to achieve high utilisation on multiple cores. In addition, the authors observed the CPU's single core bottleneck caused by the default configuration of the NIC interface. We have now identified that the NIC interrupt bottleneck is present in default virtual machine configurations. Peternier et al. [31] profile the execution of parallel multi-threaded benchmarks on multicore systems and use the collected parallel profile to predict the wall time execution of the benchmarks for a target number of cores.

Virtual Machine Performance. Cherkasova and Gardner [9] examined the performance of web applications on the Xen VM monitor on a single processor system, identifying the effect of network interrupts on physical CPU utilisation. Pu et al. [34] measured the performance of co-located web applications on virtualised clouds. Most of the work related to virtual network interfaces is concerned with optimising the packet delivery between the physical NIC and the hosted virtual machines, e.g. [4, 36]. VM migration optimisation and performance has been studied using a variety of approaches including analytical models [1, 28], regression-based models [22, 27] and benchmarking [26].

Multicore Modelling. Most queueing network models represent k-core processors as M/M/k queues. M/M/k models have also been used when modelling virtualised applications running on multicore architectures. Cerotti et al. [7] benchmark and model the performance of virtualised applications on a multicore environment using an M/M/k queue. Brosig et al. [5] model the overhead of virtualised applications using multi-server queueing Petri-nets similar to an M/M/k queue with additional scheduling mechanisms for overlapping resource usage. The authors assume that the VM-specific CPU demands and the VM-specific overhead in terms of induced CPU demand on Domain-0 are known. Brosig et al. reported accurate prediction of server utilisation; however, large errors were present for response time calculations for multiple guest VMs. Bardhan et al. [2] developed an approximate two-level single-class Queueing Network model to predict the execution time of applications on multicore systems. The model captures the memory contention caused by multiple cores and incorporates it into an application-level model.

7. CONCLUSION

This paper has presented a performance model for web applications deployed in multicore virtualised environments. The model is general enough to capture the performance of web applications deployed on multicore VMs and can account for hardware idiosyncrasies such as CPU bottlenecks and interrupt influences. We gave an approximate analytical solution and validated our model in our testbed using an open-source web application running on multicore VMs. In addition, we presented a simple approach to achieve better scalability for multicore web servers through use of virtual hardware. We also demonstrated the applicability of our model in the enhanced configurations.

In future, we will refine our model to overcome the approach limitations we discussed and extend our model to multi-granularity multiple VM instances. We also plan to investigate how the model fits multiple applications (e.g. I/O-intensive) deployed on heterogeneous VM instances. Another direction is to develop a multi-objective optimisation policy to support more comprehensive resource management.

Acknowledgements

We would like to thank Lukas Rupprecht and the anonymous reviewers for insightful comments. We greatly appreciated the help with test-bed setup from Duncan White, Lloyd Kamara, Thomas Joseph and other CSG team members. Thanks also to Eva Kalyvianaki and members of AESOP group. Xi Chen is supported by an Imperial Faculty of Engineering International Scholarship.

8. REFERENCES

[1] A. Aldhalaan and D. A. Menascé. Analytic performance modeling and optimization of live VM migration. *Proc. EPEW*, pages 28–42, 2013.

[2] S. Bardhan and D. A. Menascé. Analytic performance models of applications in multi-core computer. *Proc. MASCOTS*, 2013.

[3] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization categories and subject descriptors. *Proc. SOSP*, 2003.

[4] M. Bourguiba, K. Haddadou, and G. Pujolle. Packet aggregation based network I/O virtualization for cloud computing. *Proc. Computer Communications*, pages 309–319, Feb. 2012.

[5] F. Brosig, F. Gorsler, N. Huber, and S. Kounev. Evaluating approaches for performance prediction in virtualized environments. *Proc. MASCOTS*, 2013.

[6] J. Cao, M. Andersson, C. Nyberg, and M. Kihl. Web server performance modeling using an M/G/1/K*PS queue. *Proc. Telecommunications*, 2:1501–1506, 2003.

[7] D. Cerotti, M. Gribaudo, P. Piazzolla, and G. Serazzi. End-to-End performance of multi-core systems in cloud environments. *Proc. EPEW*, pages 221–235, 2013.

[8] L. Y. Chen, G. Serazzi, D. Ansaloni, E. Smirni, and W. Binder. What to expect when you are consolidating: effective prediction models of application performance on multicores. *Proc. Cluster Computing*, May 2013.

[9] L. Cherkasova and R. Gardner. Measuring CPU overhead for I/O processing in the Xen virtual machine monitor. *Proc. USENIX ATEC*, pages 387–390, 2005.

[10] J. D. Deng and M. K. Purvis. Multi-core application performance optimization using a constrained tandem queueing model. *Journal of Network and Computer Applications*, 34(6):1990–1996, Nov. 2011.

[11] N. J. Dingle, W. J. Knottenbelt, and T. Suto. PIPE2: A tool for the performance evaluation of generalised stochastic Petri nets. *Proc. ACM SIGMETRICS*, 2009.

[12] H. Esmaeilzadeh, E. Blem, R. St. Amant, K. Sankaralingam, and D. Burger. Dark silicon and the end of multicore scaling. *Proc. ISCA*, pages 365–376, 2011.

[13] M. Ferdman, A. Adileh, and O. Kocberber. Clearing the clouds: a study of emerging scale-out workloads on modern hardware. *Proc. ASPLOS 2012*, pages 1–11, 2012.

[14] P. Gepner and M. Kowalik. Multi-Core processors: new way to achieve high system performance. *Proc. PARELEC*, pages 9–13, 2006.

[15] D. Gupta, L. Cherkasova, R. Gardner, and A. Vahdat. Enforcing performance isolation across virtual machines in Xen. *Proc. Middleware*, 2006.

[16] V. Gupta, R. Nathuji, and K. Schwan. An analysis of power reduction in datacenters using heterogeneous chip multiprocessors. *Proc. ACM SIGMETRICS*, pages 87–91, 2011.

[17] R. Han, L. Guo, M. M. Ghanem, and Y. Guo. Lightweight resource scaling for cloud applications. *Proc. CCGrid*, pages 644–651, May 2012.

[18] A. S. Harji, P. A. Buhr, and T. Brecht. Comparing high-performance multi-core web-server architectures. *Proc. SYSTOR*, pages 1–12, 2012.

[19] P. G. Harrison. Turning back time in Markovian process algebra. *Journal of Theoretical Computer Science*, 290:1947–1986, Jan. 2003.

[20] P. G. Harrison, C. M. Lladó, and R. Puigjaner. A unified approach to modelling the performance of concurrent systems. *Journal of Simulation Modelling Practice and Theory*, 17:1445–1456, Oct. 2009.

[21] R. Hashemian, D. Krishnamurthy, M. Arlitt, and N. Carlsson. Improving the scalability of a multi-core web server. *Proc. ACM/SPEC ICPE*, pages 161–172, 2013.

[22] N. Huber, M. V. Quast, M. Hauck, and S. Kounev. Evaluating and modeling virtualization performance overhead for cloud environments. *Journal of CLOSER*, pages 563–573, 2011.

[23] W. Iqbal, M. Dailey, and D. Carrera. SLA-driven adaptive resource management for web applications on a heterogeneous compute cloud. *Proc. CloudCom*, pages 243–253, 2009.

[24] H. C. Jang and H. W. Jin. MiAMI: Multi-core aware processor affinity for TCP/IP over multiple network interfaces. *Proc. HPI*, pages 73–82, Aug. 2009.

[25] N. Khanyile, J. Tapamo, and E. Dube. An analytic model for predicting the performance of distributed applications on multicore clusters. *Proc. IAENG*, 2012.

[26] S. Kikuchi and Y. Matsumoto. Performance modeling of concurrent live migration operations in cloud computing systems using PRISM probabilistic model checker. *Proc. Cloud Computing*, pages 49–56, 2011.

[27] H. Liu, H. Jin, C. Z. Xu, and X. Liao. Performance and energy modeling for live migration of virtual machines. *Proc. HPDC*, pages 249–264, Dec. 2011.

[28] D. A. Menascé. Virtualization: concept, application, and peformance modeling. *Proc. CMG conference*, 2005.

[29] Q. Noorshams, D. Bruhn, S. Kounev, and R. Reussner. Predictive performance modeling of virtualized storage systems using optimized statistical regression techniques categories and subject descriptors. *Proc. ACM/SPEC ICPE*, pages 283–294, 2013.

[30] Peter G. Harrison, Nareth M. Patel. *Performance modeling of communication networks and computer architecture*. Addison-Wesley, 1992.

[31] A. Peternier, W. Binder, A. Yokokawa, and L. Chen. Parallelism profiling and wall-time prediction for multi-threaded applications. *Proc. ACM/SPEC ICPE*, pages 211–216, 2013.

[32] R. Prasad, M. Jain, and C. Dovrolis. Effects of interrupt coalescence on network measurements. *Passive and active network measurement*, pages 247–256, 2004.

[33] G. Prinslow and R. Jain. Overview of performance measurement and analytical modeling techniques for multi-core processors, 2011. `http://www.cse.wustl.edu/~jain/cse567-11/ftp/multcore/`.

[34] X. Pu, L. Liu, Y. Mei, S. Sivathanu, Y. Koh, C. Pu, and Y. Cao. Who is your neighbor: Net I/O performance interference in virtualized clouds. *Proc. Services Computing*, pages 314–329, 2012.

[35] A. Rai, R. Bhagwan, and S. Guha. Generalized resource allocation for the cloud. *Proc. ACM SoCC*, pages 1–12, 2012.

[36] K. K. Ram, J. R. Santos, Y. Turner, A. L. Cox, and S. Rixner. Achieving 10 Gb/s using safe and transparent network interface virtualization. *Proc. ACM SIGPLAN/SIGOPS VEE*, 2009.

[37] C. Reiss, A. Tumanov, and G. Ganger. Heterogeneity and dynamicity of clouds at scale: Google trace analysis. *Proc. SoCC*, 2012.

[38] G. Shanmuganathan, A. Gulati, and P. Varman. Defragmenting the cloud using demand-based resource allocation categories and subject descriptors. *Proc. ACM SIGMETRICS*, pages 67–80, 2013.

[39] A. Sharifi and S. Srikantaiah. Mete: meeting end-to-end qos in multicores through system-wide resource management. *Proc. ACM SIGMETRICS*, pages 13–24, 2011.

[40] U. Sharma, P. Shenoy, and D. F. Towsley. Provisioning multi-tier cloud applications using statistical bounds on sojourn time. *Proc. ICAC*, pages 43–52, 2012.

[41] S.S.Lam. Queuing Networks with Population Size Contraints. *IBM Journal of Research and Development*, pages pp 370–378, July, 1977.

[42] T. Suto, J. Bradley, and W. Knottenbelt. Performance trees: A new approach to quantitative performance specification. *Proc. MASCOTS*, pages 303–313, 2006.

[43] B. M. Tudor and Y. M. Teo. On understanding the energy consumption of ARM-based multicore servers. *Proc. ACM SIGMETRICS*, pages 267–278, 2013.

[44] A. Tumanov and J. Cipar. alsched: algebraic scheduling of mixed workloads in heterogeneous clouds. *Proc. SoCC*, 2012.

[45] B. Veal and A. Foong. Performance scalability of a multi-core web server. *Proc. ANCS*, pages 57–66, 2007.

[46] D. Wentzlaff, K. Modzelewski, and J. Miller. An operating system for multicore and clouds : mechanisms and implementation categories and subject descriptors. *Proc. SoCC*, 2010.

[47] W. Wu, M. Crawford, and M. Bowden. The performance analysis of Linux networking - packet receiving. *Proc. International Journal of Computer Communications*, 2006.

[48] F. Wuhib, R. Stadler, and H. Lindgren. Dynamic resource allocation with management objectives implementation for an OpenStack cloud. *Proc. CNSM*, pages 309–315, 2012.

APPENDIX

A. PROOF OF PROPOSITION 2

PROOF. By definition, the expected number of jobs is

$$E(k) = \sum_{i,j}(i+j)\pi_{i,j}.$$

Using results from Proposition 1, we have

$$\begin{aligned} E(k) &= \sum_{i,j}(i+j)\pi_{i,j}, \\ &= \pi_{0,0}\sum_{i,j}(i+j)\alpha^i\beta^j, \\ &= \pi_{0,0}\sum_{i=0}^{N}\sum_{j=0}^{N-i}(i+j)\alpha^i\beta^j, \\ &= \pi_{0,0}\frac{g(\alpha,\beta)-g(\beta,\alpha)+(\beta-\alpha)(2\alpha\beta-\alpha-\beta)}{(\alpha-1)^2(\alpha-\beta)(\beta-1)^2}, \\ &= \frac{g(\alpha,\beta)-g(\beta,\alpha)+(\beta-\alpha)(2\alpha\beta-\alpha-\beta)}{[\alpha^{N+2}(\beta-1)+\beta^{N+2}(1-\alpha)+\alpha-\beta](\alpha-1)(\beta-1)}, \end{aligned}$$

where $g(x,y) := x^{N+2}(y-1)^2(xN-N-1)$. □

B. PROOF OF PROPOSITION 3

PROOF. Let n_a be the current number of class a job in the system, we have

$$\begin{aligned} E(T_s) &= E(T_s|\gamma \text{ is job a})P(\gamma \text{ is job a}) \\ &\quad + E(T_s|\gamma \text{ is job b})P(\gamma \text{ is job b}) \\ &= \frac{1}{\mu_1}P(\gamma \text{ is job a}) + \frac{1}{\mu_2}P(\gamma \text{ is job b}) \\ &= \frac{1}{\mu_1}P(\gamma \text{ is job a}|n_a=0)P(n_a=0) \\ &\quad + \frac{1}{\mu_1}P(\gamma \text{ is job a}|n_a>0)P(n_a>0) \\ &\quad + \frac{1}{\mu_2}P(\gamma \text{ is job b}|n_a=0)P(n_a=0) \\ &\quad + \frac{1}{\mu_2}P(\gamma \text{ is job b}|n_a>0)P(n_a>0). \end{aligned}$$

Since

$$P(\gamma \text{ is job b}|n_a=0)=0, \qquad P(\gamma \text{ is job a}|n_a=0)=1,$$

we have

$$\begin{aligned} E(T_s) &= \frac{1}{\mu_1}P(n_a=0) \\ &\quad + \frac{1}{\mu_1}P(\gamma \text{ is job a}|n_a>0)P(n_a>0) \\ &\quad + \frac{1}{\mu_2}P(\gamma \text{ is job b}|n_a>0)P(n_a>0). \\ &= \frac{1}{\mu_1}P(n_a=0) + \frac{1}{\mu_1}\frac{\lambda}{\lambda+p\mu_1}P(n_a>0) \\ &\quad + \frac{1}{\mu_2}\frac{p\mu_1}{\lambda+p\mu_1}P(n_a>0) \\ &= \frac{1}{\mu_1}P(n_a=0) + \frac{1}{\mu_1}\frac{\lambda}{\lambda+p\mu_1}(1-P(n_a=0)) \\ &\quad + \frac{1}{\mu_2}\frac{p\mu_1}{\lambda+p\mu_1}(1-P(n_a=0)). \end{aligned}$$

Notice that from previous results,

$$\begin{aligned} P(n_a=0) &= \sum_{j=0}^{N}\pi_{0,j} \\ &= \pi_{0,0}\sum_{j=0}^{N}\alpha^0\beta^j \\ &= \pi_{0,0}\frac{1-\beta^{N+1}}{1-\beta}. \end{aligned}$$

Therefore,

$$E(T_s) = \frac{1}{\mu_1}n_a^0 + \frac{1}{\mu_1}\frac{\lambda}{\lambda+p\mu_1}(1-n_a^0) + \frac{1}{\mu_2}\frac{p\mu_1}{\lambda+p\mu_1}(1-n_a^0),$$

where

$$n_a^0 = \pi_{0,0}\frac{1-\beta^{N+1}}{1-\beta}.$$

□