13

Virtual Machine Analysis with PDQ

13.1 Introduction

For enterprise data centers, the latest virtualization technologies afford many new opportunities for performance management. These include resource provisioning, resource consolidation, application migration, reducing electrical power, high availability (HA) and virtualized storage, just to name a few. Linux is at the forefront of this advance both as an operating system standard and as the underpinning for virtual machine managers or hypervisors.

To maximize your return on investment in virtual machine management (VMM) environments, you need to be understood certain principles of operation to achieve good performance analysis across your enterprise. In this chapter, we will cover the following points in detail:

- Virtualization spectrum for hyperthreaded multicores, hypervisors and clouds
- Review of the fair-share scheduling infrastructure
- Instrumentation and tools for VMMs
- Controlled scalability measurements

As the title suggests, the emphasis here is on enterprise scale performance management, not desktop virtualization. Cloud computing is the latest technology involving commoditization of virtual services, e.g., Amazon's EC2 (see `aws.amazon.com/ec2/`). Apart from the fact that performance management of this level of virtualization is relatively immature, a complete discussion would take us too far afield, although we will consider how they fit into the overall picture in the next section.

When it comes to hypervisors, there are many choices: Citrix XenServer, Microsoft Hyper-V, VMware ESX Server, etc. Here, we shall focus mostly on Linux and VMware ESX Server as the enterprise virtualization environment since it ubiquitous in the marketplace. This is a "bare metal" hypervisor, which has much higher performance as well as better performance and resource

N.J. Gunther, *Analyzing Computer System Performance with Perl::PDQ*,
DOI 10.1007/978-3-642-22583-3_13, © Springer-Verlag Berlin Heidelberg 2011

management tools than both the VMware Server and the workstation product, both of which are hosted hypervisors. Moreover, VMware, Inc. has done a lot of due diligence when it comes to providing performance information that is useful for performance management. In particular, the company has not just relied on sometimes poorly defined benchmarks, but has undertaken *controlled experiments* that offer the system administrator some insight into how best to do performance analysis.

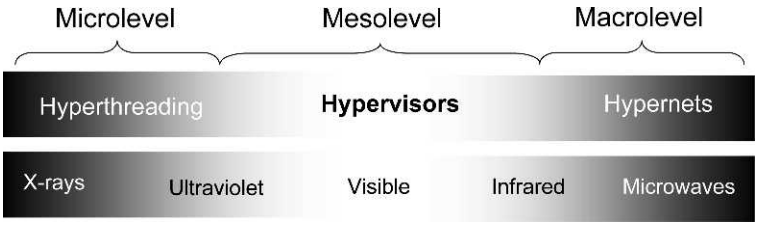


Fig. 13.1. Nominal examples of virtualization technologies organized as a discrete spectrum (*top*) in analogy to the continuous electromagnetic spectrum (*bottom*). Hypernets can include such technologies as GRIDs and Clouds. The performance attributes of meso-VMMs also tend to be the most visible to the performance analyst

13.2 The Virtual Machine Spectrum

Virtualization is about creating illusions. In particular, modern computer systems are now sufficiently powerful to present users with the illusion that one physical machine is really multiple virtual machines, each one running a separate instance of a different operating system (OS). This is one reason for the resurgence of interest in virtualization technologies.

The idea of virtual resources such as software emulators and virtual memory is not new and goes all the way back to mainframe architectures [Singh 2004]. As we shall show in this chapter, the distinctive architectural feature that modern VMs have in common is some form of *polling* mechanism to accomplish resource sharing. Polling algorithms are intrinsically stable and fair; even round-robin polling exhibits intrinsic fairness. The potential performance penalty arises from the sequential nature of polled service being worse than asynchronous service. We consider the details of polling systems throughout this chapter, starting with *hyperthreading* in Sect. 13.3. Before considering the dissimilarities of VMMs, however, we begin by examining their commonalities based on the notion of a VMM spectrum.

With polling as the discriminant, we can construct a classification the various VM manifestations by positioning each of them on the *discrete* spectrum shown in Figure 13.1, which is analogous to the more familiar *continuous*

electromagnetic (or EM) spectrum. Just as the EM spectrum can be grouped into the ultraviolet (UV), visible and infrared (IR) regions, the VM spectrum can be similarly grouped into the *micro-VMM*, *meso-VMM*, and *macro-VMM* spectral regions. The relative position of each VMM is defined by its respective polling rate or polling frequency. Examples of these polling attributes are identified in Table 13.1.

Table 13.1. Typical VM spectrum polling periods and frequencies

Spectral Region	Polling	
	Period	Frequency
macro-VMM	min to day	mHz to μHz
meso-VMM	ms to min	kHz to mHz
micro-VMM	ns to μs	GHz to MHz

The so-called *visible* region on the EM-spectrum is actually an anthropocentric term because certain snakes can see in the IR (detect heat) and bees can see in UV light. Similarly, only meso-VMMs constitute a "visible" region to us in the sense that conventional management tools provide an immediate view of VMM performance. This level visibility provides some level of potential control. Conversely, micro-VMMs and macro-VMMs tend to be "invisible" to those same performance tools, so those VM resources remain largely beyond our control for performance analysis.

> `perfdynamics.blogspot.com Sunday, Feb 25, 2007:`
>
> *The VMM-spectrum concept is based on my observations that (i) disparate types of virtual machines lie on a discrete spectrum bounded by hyperthreading at one extreme and hyperservices at the other, and (ii) poll-based scheduling is the common architectural element in most VM implementations. The associated polling frequency (from GHz to μHz) positions each VMM in a region of the notional spectrum.*

From this standpoint, the distinction between VMMs according to whether they are implemented in hardware or software seems artificial—as artificial as the distinction between heat and light. Recognizing each as different manifestations of the same spectrum can lead to important insights. As we endeavor to show in the remainder of this chapter, the VMM-spectrum classification is more than mere whimsy.

An important point to note is that the relative position of each VM subtype on the VM spectrum is determined by the rate at which polling is carried out by the underlying scheduling subsystem. To make this statement more quantitative, we refer to a case where the polling periods are well documented: meso-VMM scheduling. The polling period T_p for the scheduler to associate

physical resource consumption with each active OS instance (VM guest) is $T_p = 4000$ ms, or once every 4 s. Therefore, the frequency is $f = 1/T_p = 0.25$ cycles per second or 250 mHz.

Because it is not publicly documented, we assume that the micro-VMM polling period lies in the range of nanoseconds to microseconds, since the processor has a clock frequency in the range of GHz to MHz. Macro-VMMs can take minutes or days to detect active peer horizons. These frequencies are key VM performance determinants. These polling ranges are summarized in Table 13.1.

13.3 Micro-VMM Scale: Hyperthreading

Let's start on the ground floor of Figure 13.1 where virtualization operates on the smallest size and at the highest frequency as defined in Table 13.1. Since this is the foundational level, any performance issues that arise here could influence performance in the virtualization layers above it. In today's environments, this level is a hardware layer sometimes referred to as *bare metal*. In particular, we shall focus on the microprocessor as the physical resource of interest for performance management.

Intel Corp., for example, refers to this form of processor virtualization as *symmetric multi-threading* (SMT), based on its proprietary hyperthreading (HT) technology, originally implemented in its Pentium P4 microprocessor line and now available on its Nehalem multicore Xeon product lines.

The rationale is to maximize throughput performance by utilizing idle cycles. Part of the confusion that has been associated with hyperthreaded performance stems from two possible views of what it offers. Referring to Figure 13.2:

1. **Hardware Capacity View:** Total capacity $C_{\mathrm{HW}} = 1 + \epsilon$ where ϵ is a small fractional quantity representing the possible capacity gain due to hyperthreading, e.g., 0.1 represents 10%. Intel quotes typical performance gains ranging from $\epsilon = 0.10$ to 0.30. Since, however, there is only one execution unit in actuality—which is often underutilized—by simply adding another architectural state (AS) register, it becomes possible to have another thread ready to make use of any idle cycles if the first thread becomes stalled.
2. **Software Capacity View:** Total capacity $C_{\mathrm{SW}} = 2 - \delta$ as seen by the operating system and thus any performance management tools. A single HT-enabled processor presents itself to the operating system as two virtual processors or VPUs. Using the APIC (Advanced Programmable Interrupt Controller) and CPUID IA-32 instructions, the operating system literally detects the number of VPUs by interrogating the AS registers EAX and EBX on the chip. Ideally, one would expect $\delta \simeq 0$ but, in reality, $\delta \simeq 1$. Since the performance tools indicated the presence of two VPUs, compute

cycles appear to be lost. Elsewhere, this effect has been referred to this as the "missing MIPS" problem [Gunther 2007b].

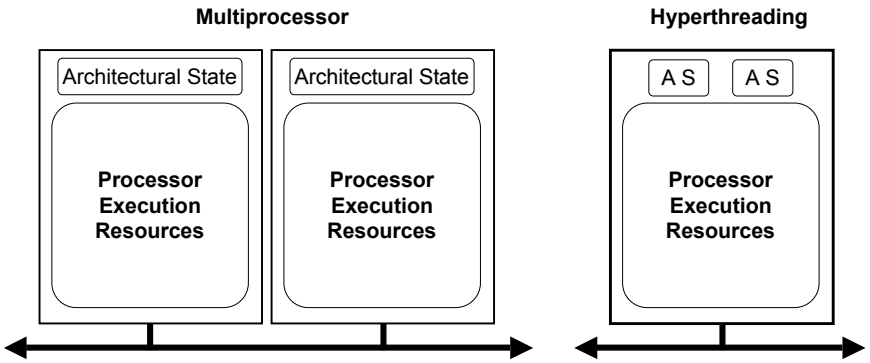


Fig. 13.2. Simple block diagram comparing a two-way symmetric multiprocessor architecture (*left*) with an Intel SMT hyperthreaded processor (*right*). Both types of microprocessor present two architectural state (AS) registers to the operating system and performance tools

The relationship between these two views can be summarized simply as $\delta = 1 - \epsilon$. In the subsequent discussion, we take the second or software view because it best represents the source of confusion for many performance analysts. In fact, for fully optimized CPU-intensive workloads, whether controlled benchmarks or certain types of production workloads, the correct starting point is close to $\delta = \epsilon = 0.5$. On the other hand, for less CPU-intensive workloads, $\epsilon \ll 0.5$ or what amounts to the same thing, $\delta \gg 0.5$, and this results in an effective capacity $C_{\mathrm{SW}} \ll 1.5$ VPUs.

Hyperthreading can also be combined with *multicore* technology, where multiple physical CPUs are interconnected like a symmetric multiprocessor (SMP) but on the same VLSI die. Sun Microsystems, for example, refers to this as *chip multi-threading* (CMT) and offers it with the UltraSPARC T2 processors comprising eight cores with eight threads per core for a total of 64-way VPUs (See `www.t2-project.org/about.html`). Canonical has an Ubuntu Linux update for the T2 (See `www.ubuntu.com/partners/sun`)

To further clarify the distinction between physical CPUs and VPUs, we employ the concept of *polled queues*. Polling systems are not new. Token-ring networks use this principle, and they are in common use for high-speed data network switches and routers. The performance characteristics of polling systems are notoriously difficult to solve analytically and one often has to resort to simulations to determine the performance characteristics.

A polling system schematic is shown in Figure 13.3, where multiple queues or buffers are multiplexed into a common server or execution unit. Prima facie, this seems like a ludicrous design choice for a hyperthreaded architecture.

After all, elementary queueing analysis tells us that the waiting time associated with a single buffer can present significant delays [Gunther 2007a], so those delays can only gets worse when more buffers are added to the same server. To underscore this point, consider a grocery store checkout aisle. Each aisle acts like a single waiting line or buffer with a single server—the cashier. A grocery store that is designed like a polling system would have the same number of aisles but only one cashier to service *all* of them! Sounds like a grocery store that you would want to avoid at all costs. However, this *polling grocery store* can be made more efficient by imposing certain conditions on how it operates. For example:

1. Cashier has to be very athletic
2. Cashier runs between the cash registers on each aisle
3. Cashier circulates in round-robin order
4. Customers are only permitted to buy one or two items

In other words, the overall performance of the store can be made acceptable if the service time for each customer is kept small (like a thread) and the cashier circulates quickly between aisles (buffers or thread registers). In reality, there are many complicated variations on this basic polling theme, but we need not be concerned with them here. The simple polling model just described has not been widely recognized in the context of hyperthreading. However, as I mentioned earlier, it will be familiar to many readers in the guise of a Token Ring network, where the rotating token determines when each successive packet queue receives routing service. The token is simply the analog of our sprinting cashier. Although the hyperthreaded processor does not employ a literal token, some overhead and delay will be incurred due to the necessity of switching between the two thread registers.

In the case of Xeon-class microprocessors there are just two queues corresponding to single-entry thread buffers, i.e., the AS registers in Figure 13.2. It is the state of these buffers that are monitored by the OS scheduler. Threads are taken off the run-queue and placed in the next empty AS buffer. On the microprocessor chip, each thread buffer is serviced by the single execution unit in some order, e.g., round-robin for fairness, although the exact protocol is in fact very complex and proprietary. It should be noted that a constant polling delay (on the order of nanoseconds in Table 13.1) is not responsible for the missing MIPS problem mentioned earlier. That extension to the polling model is developed next.

13.3.1 Controlled Measurements

Carefully controlled measurements of hyperthreaded processors indicate that execution times can depend significantly on the type of applications being run. While hyperthreading improves performance in many instances, some tests suggest that thread processing times are dependent on the load being executed by the thread scheduler. Moreover, there are internal complexities

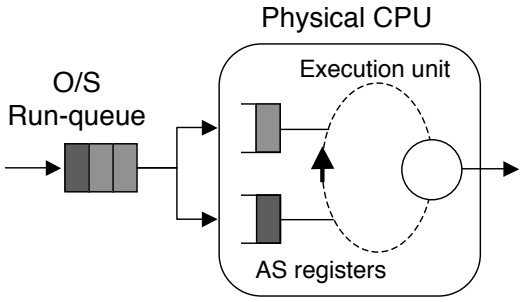


Fig. 13.3. Queueing model of the SMT hyperthreaded core in Figure 13.2 showing how the two thread registers (VPUs) are polled for the single execution unit

such as how context switching is handled, whether L1 caches are shared as in Intel processors or independent L1 caches per core as in Sun's T2, and so on. These are invisible contributions to variability in processing times can lead to erroneous capacity forecasting without an appropriate performance model.

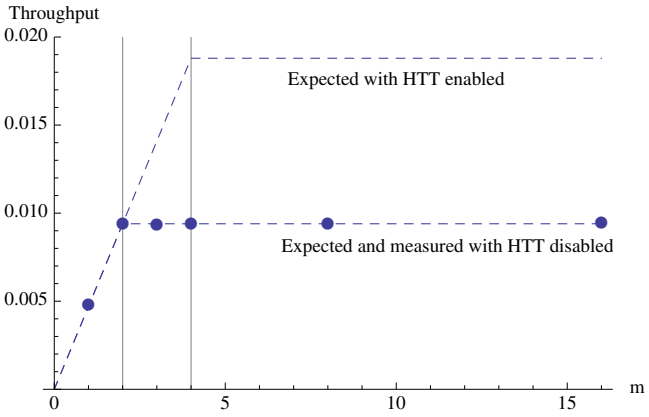
TeamQuest Corporation (`www.teamquest.com`) created a test program to consume all available CPU cycles by configuring the number of executing threads [Johnson 2003]. The test platform comprised a dual-processor Compaq ML530 equipped with 2.4 GHz Intel Xeon processors. A BIOS utility provided the ability to enable and disable hyperthreading. The elapsed time was measured at 1 second resolution using the `time()` function. System and user processing time were measured using `GetProcessTimes()` at 100 ns resolution, which included the activity of all process threads.

The objective was to determine the system throughput as a function of the number of active threads m during the elapsed time T_m of the test program. This thread-dependent throughput X_m can be calculated from the definition:

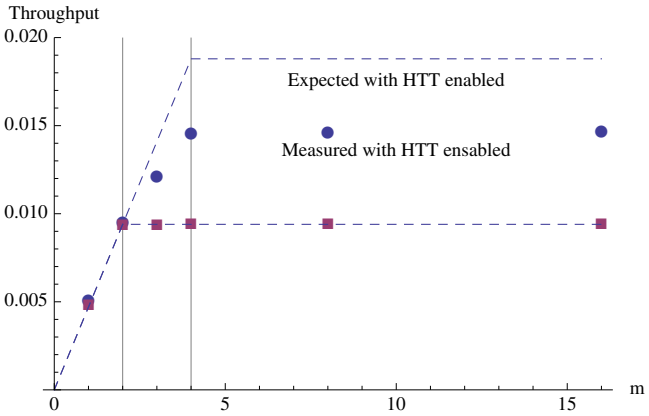
$$X_m = \frac{m}{T_m} \tag{13.1}$$

with $m = 1, 2, \ldots, 16$ threads. If $m = 1$, then $X_1 = 1/T_1$. If $m = 2$, then $X_2 = 2/T_2$, and so on. If complete thread parallelism can be maintained, the runtime will be the same for any number of threads, i.e., $T_1 = T_2 = \ldots$, etc. and the throughput will simply increase in proportion to the number of threads. At some value of m, however, only a certain number of threads may be able to execute simultaneously due to hardware limitations. Beyond that point, the throughput will remain the same, irrespective of the number of threads that have been spawned. See Figure 13.4.

Figure 13.4 shows these measured throughputs compared with the throughput predicted by PDQ. With hyperthreading disabled (lower dashed curve in Figure 13.4(a)), measurements and prediction are almost identical. A knee occurs at $m = 2$ and that also corresponds to two CPUs. With hyperthreading enabled, i.e., four VPUs, the PDQ model correctly predicts that the knee



(a) Predicted throughput (*dashed lines*) and measurements (*dots*) with hyperthreading disabled. The modeled performance is realized in the measurements



(b) With hyperthreading enabled the measured throughput does not achieve the expected limit shown by the upper dashed line. A gap persists between the actual throughput (upper dots) and the expected throughput (upper line)

Fig. 13.4. Comparison of measured throughput and PDQ modeled throughput with hyperthreading (a) disabled and (b) enabled. The anticipated doubling in throughput is not realized

occurs at $m = 4$ threads (upper dashed curve in Figure 13.4(a)). However, the data in Figure 13.4(b) reveals that the knee occurs earlier than expected and the maximum throughput is only about 75% of that predicted by the PDQ model. To understand what is going on, let's look at the latency measurements and PDQ polling model predictions.

When it comes to latency curves like those in Figure 13.5, lower is better. With hyperthreading disabled, i.e., two physical CPUs, the data fall on the upper dashed curve in 13.5(a) with the expected knee (or heel) occurring at $m = 2$. This reflects exactly what we see in Figure 13.4(a). With hyperthreading enabled and therefore four VPUs, the PDQ model predicts that the data should have much lower latency and therefore lie on the lower dashed curve. The measurements in 13.5(b), however, reveal that the data points lie *above* the predicted lower curve. Latency is worse than expected by about 30%.

13.3.2 PDQ Model of Micro-VMM

The PDQ model assumes that the average service time per thread, S_0, is a constant and independent of the number of threads. In Figure 13.5 the runtime R at the VPU with $m \leq 4$ threads, is given by

$$R = S_0. \tag{13.2}$$

This is just the constant service time S_0 seen at the foot of the "hockey stick," and signifies that a VPU is always available to service threads and thus, no waiting time is incurred. When $m > 4$, however, all VPUs become saturated and threads must wait at either the AS registers or the run-queue as depicted in Figure 13.3. The saturation residence time R_s above $m = 4$ is given by

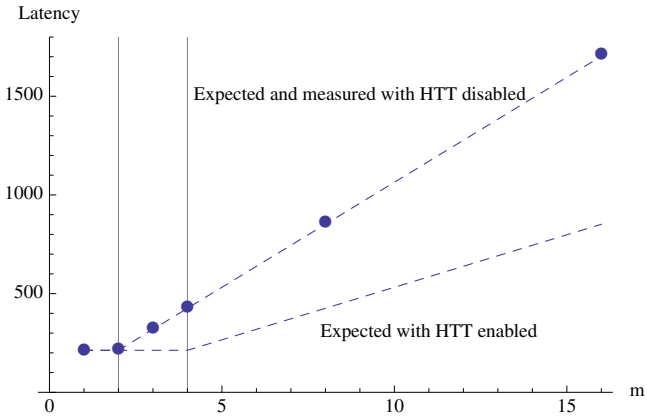
$$R = mS_0. \tag{13.3}$$

The latency increases linearly with m as shown in Figure 13.5(b)). The usual conclusion is that mS_0 is responsible for the observed increase in latency R, but this cannot be correct. The data points in Figure 13.5(b) are increasing in a *super-linear* fashion, i.e., latency is growing faster than expected when HT is enabled. How can this happen?

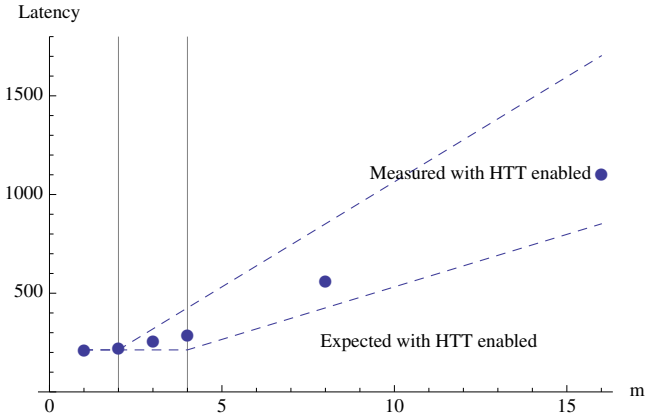
The answer is that S_0 has not remained constant as the number of threads were increased, but has stretched to a new value $S_1 > S_0$. For $m > 4$, the run time becomes

$$R = mS_1. \tag{13.4}$$

Although R is still linear rising in Figure 13.5(b), it is rising with an increased slope relative to the expected dashed line. Of the 30% increase in the measured value of R, PDQ reveals that 20% is due to a sudden increase in the thread *service* time. A reasonable conclusion is that this increase $(S_1 - S_0)$ is associated with the extra time needed for internal state management, within the hyperthreaded microprocessor, when the number of thread requests exceeds the number of VPUs.



(a) Predicted and measured execution times for Figure 13.4. Without HT enabled the data match the PDQ predictions



(b) Longer than expected elapsed times with HT are associated with a 20% increase in the service time due to polling

Fig. 13.5. Latency analysis corresponding to the throughputs in Figure 13.4

In Figure 13.2, the dual-core Compaq ML530 has two independent sets of AS registers which Intel denotes by *1a0* and *1a1* belonging to core 1, and *2a0* and *2a1* on core 2. In Figure 13.5(b), the elapsed times begin at the foot of the hockey stick where threads are each assigned to the available VPUs (empty

thread buffers) in the order *1a0, 2a0*, i.e., one thread per core. The third thread has to be assigned to a core that is now already busy, e.g., *1a1*. Notice that the data point for $m = 3$ in Figure 13.5(b) appears to lift off from the lower hockey stick, reflecting the extra time needed for internal management of the micro-VMM registers and caches. The fourth thread is then assigned to, say, buffer *2a1* on an already busy core 2. This completes the transition away from the lower hockey stick. The increase in service times is reflected in performance data as prolonged execution times.

The foregoing analysis was based on a controlled CPU-intensive workload. IO-intensive workloads are likely to show a different elapsed time profile but they can also be analyzed in the same way.

13.4 Meso-VMM Scale: Hypervisors

Virtualization of software services offers the ability to do server consolidation, co-located hosting, distributed Web services, isolation, secure computing platforms and application mobility, without the need to be concerned about how they get accomplished. But anyone who has tried to do performance tuning on a VMM like, XenServer or ESX Server knows, the devil is in the details.

> `perfdynamics.blogspot.com Friday, May 25, 2007`
>
> *All virtualization is about illusions and although it is perfectly reasonable to perpetrate such illusions upon a blissfully unaware user, it is should be considered forbidden to propagate those same illusions upon a performance analyst.*

Virtualization, in the sense it is being used here, means that instead of just having applications run on a single O/S instance (Figure 13.6), an arbitrary number of different O/S instances or guests can run concurrently on the same platform under the overarching supervision of a VMM or hypervisor (see Figures 13.7 and 13.8). The VMM provides the interface between each O/S and the actual hardware resources. Beyond the O/S instance seen by each user, the details of the hypervisor are generally hidden. Like all things *virtual*, VMMs are really about *illusions* and those illusions can be a source of real problems if too much important performance data remains hidden from the system administrator due to a lack of proper instrumentation. So, what would good VMM hypervisor instrumentation look like? To better answer that question, we should understand something about the basic principles of operation of a VMM, viz., the fair-share scheduler.

The partitioning of resources in a physical machine to support the concurrent execution of multiple O/S guests poses some important challenges:

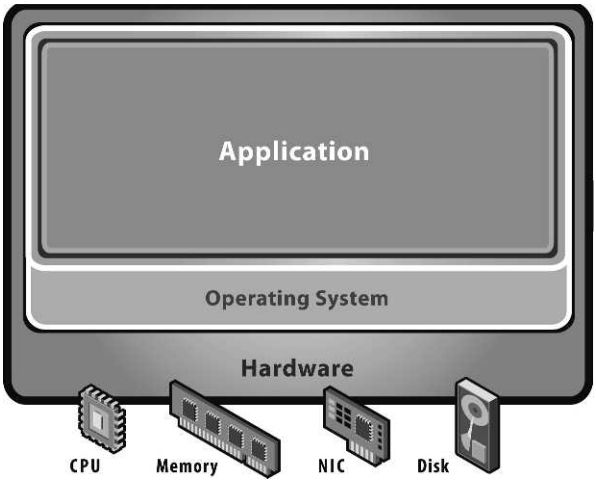


Fig. 13.6. Schematic representation of a single application running on a native O/S instance supported by physical resources such as CPU, memory, network, and disk storage [Source: VMware]

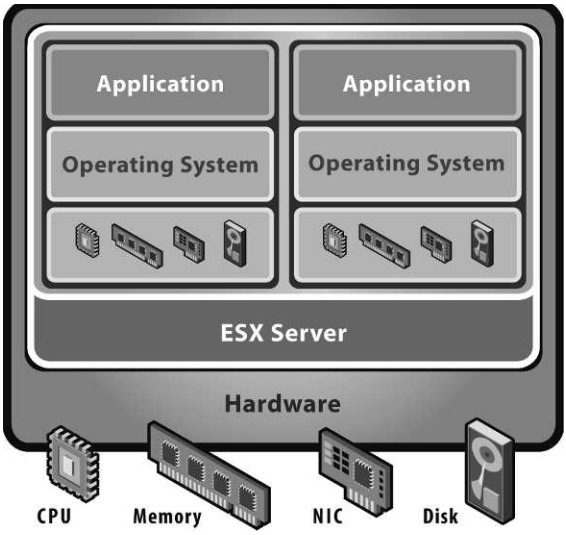


Fig. 13.7. Schematic representation of the organization of VMware ESX Server hypervisor interposed between virtual resources (*center*) and physical resources (*bottom*) [Source: VMware]

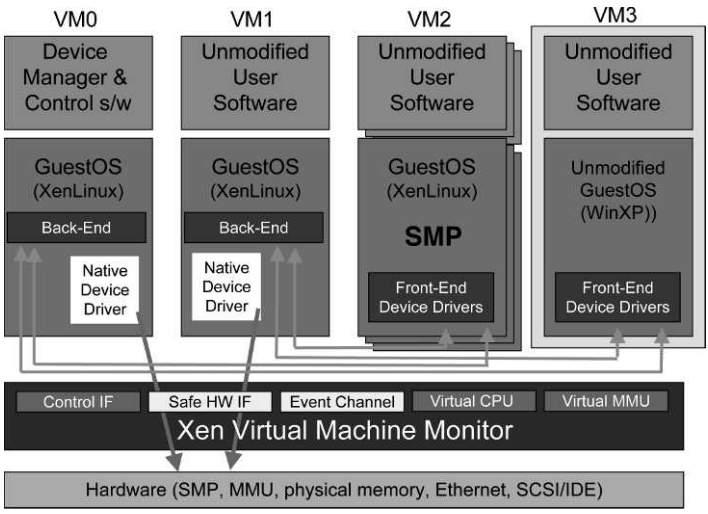


Fig. 13.8. Organization of XenServer hypervisor

1. Each OS instance must be truly isolated from the others. It is unacceptable for the execution of one OS to adversely affect the performance of another, unless explicitly intended via prioritization or QoS policies.
2. It is necessary to support a variety of different guests to accommodate the heterogeneity of popular applications.
3. The performance overhead introduced by each guest should be small.

Historically, Linux and indeed all UNIX schedulers, have been based on a time-share scheduler (TSS) in order to meet user response time requirements, as opposed to batch processing requirements. Another type of scheduler is known as a *fair-share scheduler* or FSS. An FSS requires the explicit awarding of resource shares to users by the system administrator. The important point here is that it is not commonly appreciated that an FSS also forms the underpinnings of all modern VMMs.

The way an FSS is implemented in Linux is via *cgroups* [Menage 2008] or task control groups (TCGs), which are an extension of the *completely fair scheduler* (CFS) that was merged into Linux 2.6.24. Its primary purpose is the capacity and resource management of processors, memory, and I/O devices via Linux containers. Cgroups are a low-level mechanism in Linux for both containers and virtual machines. If a program is part of a particular control group, it will be given a share of the machine resources. The particular share of those resource is specified in much the same way as one accounts for equity in corporate profits, i.e., through the literal awarding of shares. To better understand how an FSS differs from a TSS, we begin by quickly reviewing the principles of operation for the TSS.

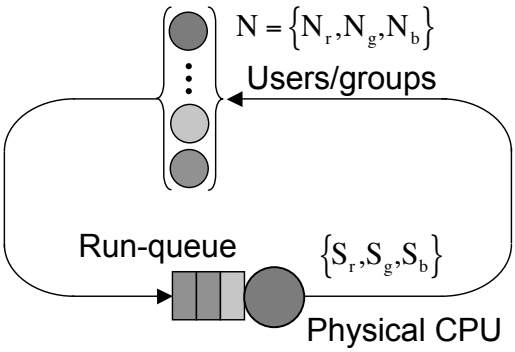


Fig. 13.9. Time-share scheduler model in PDQ. Each user group N_r, N_g, N_b requires service time S_r, S_g, S_b at the commonly shared physical CPU

The purpose of a TSS is simply to provide each user with the *illusion* that he is the only person using the physical platform. In Linux, each user process is in one of three possible states: *running*, *runnable* or *sleeping*. If a process is running, it will be in the lower part of Figure 13.9, executing on the physical CPU. If the process is runnable but not executing, then it will reside in the waiting line or run-queue [Gunther 2007a], shown immediately to the left of the physical CPU. If a process has not completed execution when the *time-quantum* expires (e.g., 10 ms or 50 ms in VMware ESX Server) it is returned to the tail of the run-queue. Otherwise, the process is sleeping because it is not ready to execute, perhaps waiting on an I/O to complete, as shown in the upper part of the diagram.

By contrast, the FSS shown schematically in Figure 13.10, provides each of the users in $\mathrm{Group_r}$, $\mathrm{Group_g}$, and $\mathrm{Group_b}$ with the illusion that he possesses an entire platform of his own—*a virtual machine*—whose performance is scaled according to his resource entitlement: $\mathcal{E}_r, \mathcal{E}_g, \mathcal{E}_b$, which determines the effective speed of his respective virtual processor $\mathrm{VPU_r}, \mathrm{VPU_g}, \mathrm{VPU_b}$. The physical service time S_{guest} for each guest process becomes a *virtual* service time given by

$$S^V_{\mathrm{guest}} = \frac{S_{\mathrm{guest}}}{\mathcal{E}_{\mathrm{guest}}}, \tag{13.5}$$

which is either faster or slower than S_{guest} according to how it is scaled by the awarded share entitlement. The processing rate is simply the inverse of the service time.

Both XenServer and VMware ESX Server use a generalized form of FSS called *proportional share scheduling*. Each runnable guest receives a share of the processor in proportion to its weight. For example, a single-processor VMware ESX Server guest OS is allocated 1000 shares by default [Gunther 2007b]. Share allocation can have a significant impact on overall performance. An FSS introduces a scheduling superstructure on top of a conventional TSS

to connect processes with users and their resource entitlements as represented in the following, highly simplified, pseudocode.

VMM Share Scheduling: Polls every 4000 ms or with frequency $f = 250$ mHz to compare physical processor usage per user entitlement. See Figure 13.10.

```
for(u = 0; u < USERS; u++) {
    usage[u] *= decayUsage;
    usage[u] += cost[i];
    cost[u]   = 0;
}
```

VMM Priority Adjustment: Polls every 1000 ms and decays the internal FSS process priority values. See Figure 13.10.

```
priDecay in [0..1];
for(k = 0; k < PROCS; k++) {
    sharepri[k] *= priDecay;
}
priDecay = a * p_nice[k] + b;
```

Time Share Scheduling: Examines CPU ticks to adjust process priorities like the conventional Linux or UNIX TS scheduler in Figure 13.9.

```
for(u = 0; u < USERS; u++) {
    sharepri[u] += usage[u] * p_active[u];
}
```

Process-level polling is identical to standard TSS, while VMM fair-share polling controls capacity consumption. It should be clear from this pseudocode that FSS has additional overheads and latencies compared to the simpler TSS. Once again, the polling operation in Figure 13.10 is analogous to that of our sprinting grocery store cashier. In reality, however, the polling latencies have been significantly reduced in more recent meso-VMM implementations. See Section 13.4.2. The more serious latencies, which are now beginning to be addressed by VMM vendors, concern I/O devices.

The net effect of FSS under maximal load can be best understood by considering a pair of users with entitlements $\mathcal{E}_1$ and $\mathcal{E}_2$; the FSS algorithm endeavors to make the resource utilization (ρ) due to each user correspond to the ratio of their entitlements:

$$\frac{\rho_1}{\rho_2} \to \frac{\mathcal{E}_1}{\mathcal{E}_2}. \tag{13.6}$$

In other words, the long-term goal of FSS is to try and match the sampled ratio of utilizations (expressed as a percentage) to the ratio of their entitlements (expressed as a percentage).

The shares provide resource minimums or least upper bounds rather than upper bounds or maximums. If you give one resource group 10% of the bandwidth of a particular disk and another 90%, then if the more privileged group isn't using its full 90%, the other group can have whatever is left over.

As with other Linux constructs, TCGs are organized in terms of a virtual file system, and therefore they can be managed by reading and writing

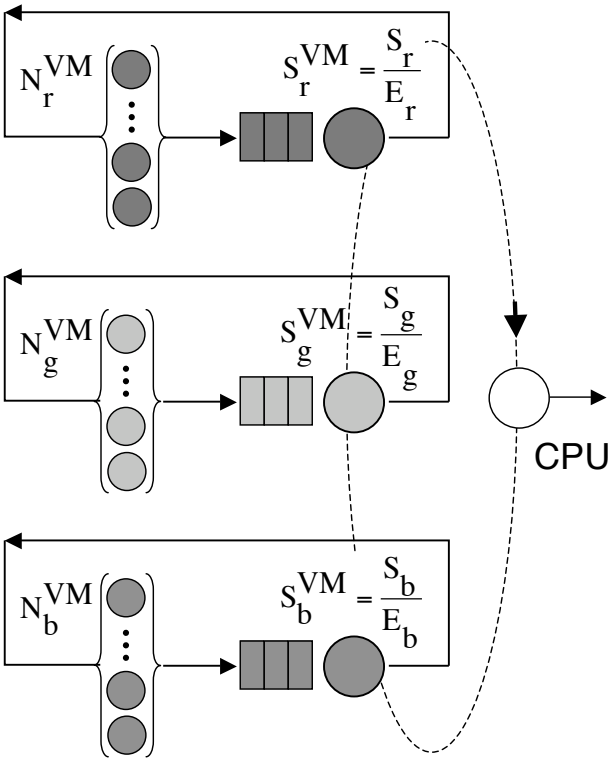


Fig. 13.10. Fair-share scheduler model of three user groups N_r, N_g, N_b, each with its own virtual processors operating with effective service time proportions $S_r^{VM}, S_g^{VM}, S_b^{VM}$ of the shared physical CPU. CPU sharing is accomplished by a polling-like mechanism described in Section 13.3, which acts like the athletic grocery store cashier

files. As you create additional cgroups and assign them shares of the CPU, the dispatcher will recalculate the percentage of the total CPU each cgroup will get, saving you from having to calculate percentages that will add up to 100%. More details of setting up TCGs can be found in the online documentation [Menage 2008].

The natural inclination is to make use of otherwise idle processing resources but that notion rests on two assumptions:

1. You are not being charged for the consumption of processing resources. If your manager only has a budget to pay for a maximum of 10% processing on a shared server, then it would be fiscally undesirable to exceed that limit.
2. You are unconcerned about service targets. It is a law of nature that users complain about perceived changes in response time. If there are

operational periods where response time is significantly better than at other times, those periods will define the future service target.

Such dynamically changing capacity can have detrimental consequences for both performance perceived by the users and the overall capacity allocation strategy.

Next, let's see what instrumentation and tools are available for performance management based on this FSS infrastructure.

13.4.1 Performance Monitoring Tools

As mentioned earlier, sufficient instrumentation is missing to enable administrators to diagnose performance problems in virtualized environments. We performance analysts need fewer bells and more whistles! It seems that some meso-VM vendors may have been listening. Indeed, the pendulum may have indeed swung the other way, so that the problem is now more one of filtering the veritable fire-hose of performance data.

The primary diagnostic tool for VMware ESX Server is `esxtop` in Figure 13.11. It comes pre-installed with ESX service console. A remote version called `resxtop` ships with the Remote Command Line interface (RCLI) package.

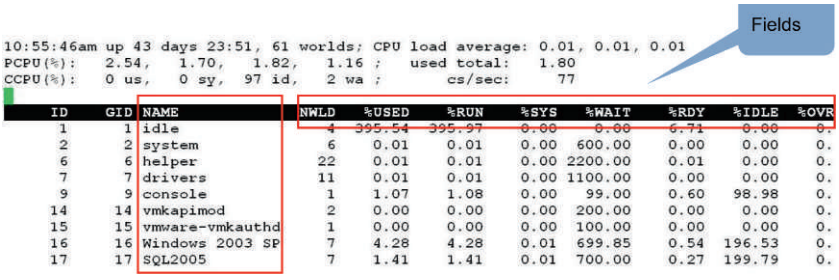


Fig. 13.11. esxtop is the performance diagnostic tool for ESX Server [Source: VMware]

Since `esxtop` only monitors instantaneous performance metrics, it is not meant for long-term performance monitoring, data mining, reporting, and alerting. For that, VMware ESX Server provides VI Client to review historical data, do performance analysis by looking for data patterns. Some performance metrics are static and do not change during runtime, e.g., `MEMSZ` (memory size), while metrics like CPU busy are computed dynamically. Other metrics, such as memory and network statistics, are calculated from the difference between two successive snapshots.

As discussed in Section 13.4, ESX Server uses a proportional-share scheduler to help with resource management using limits, shares, and reservations.

Performance tools like esxtop and VI Client can be used to assess the performance impact of a given choice of these resource management constraints.

13.4.2 Controlled Measurements

To determine any confounding effects between micro-VMM hyperthreading (Section 13.3) and meso-VMM performance (Section 13.4), Figure 13.12 shows measured VMware ESX Server throughput as a function of the number of active VMs or guests with SMT enabled. These controlled measurements compare Red Hat Enterprise Linux 3 running the SPEC CPU2000 gzip workload [SPEC 2000].

The test platform is an HP four-way ProLiant DL580 with HTT enabled. 4-way × 2 threads = 8 VPUs to VMWare and should therefore exhibit a knee at eight VMs. Recalling the discussion in Section 13.3, however, it is more realistic to expect the actual virtual capacity to be closer to

$$4\text{-way} \times 2 \times \frac{3}{4} = 6 \text{ VMs}.$$

Figure 13.12 shows that if the saturation throughput is projected back toward the dashed line, the actual knee occurs somewhere between 5 and 6 VMs and this accounts for why the theoretical throughput ceiling is never achieved.

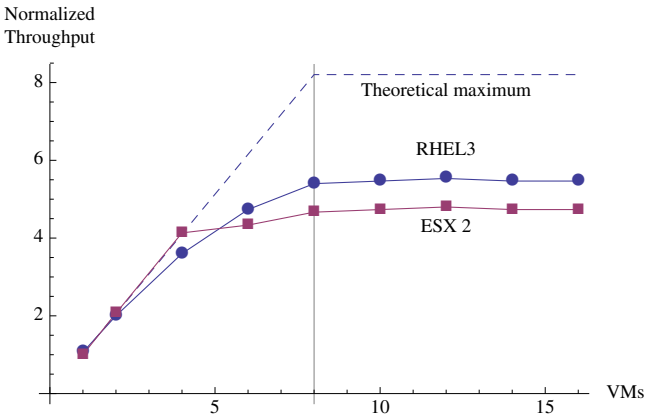


Fig. 13.12. Relative throughput scalability of ESX Server 2 and Red Hat Enterprise Linux 3 running the SPEC CPU2000 gzip workload with Intel hyperthreading enabled. Cf. Figure 13.4(b) [Source: VMware]

You might now be wondering whether, in light of our understanding of thread-limited throughput, the PDQ performance model should now be modified to include this new information. In other words, should the dashed line labeled "Theoretical maximum" in Figure 13.12 be lowered to better match

the data points? This kind of maneuvering in the PDQ model is not recommended. One reason is that performance modeling is about insight, not curve-fitting. Insight demands simplicity, not complexity. Therefore, the PDQ performance model should be kept simple, not necessarily realistic.

There is a second reason to leave the PDQ model as it is. A less recognized purpose of performance modeling is to force an *explanation*. Performance modeling is not just about prediction, it is also about explanation.

> As a general rule, it is more important to expose performance that is missing, rather than merely mimic existing performance. Things change (especially in computer systems), and the PDQ model, as it is, alerts you to what can be, not just what is. The same comment can also be made about the controlled performance measurements in Figure 13.4.

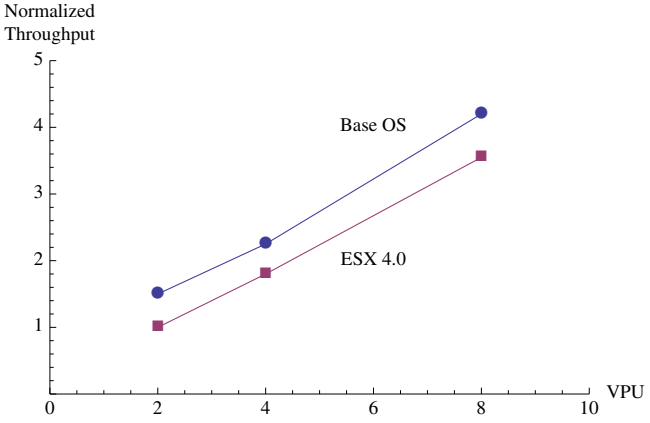


Fig. 13.13. Relative scalability of ESX 4.0 (upper curve) and native OS (lower curve) executing a TPC-C-like order entry Oracle 11g database workload [Source: VMware]

Although these data are from 2006, they reinforce the point about doing controlled measurements to assess VMM scalability. Although it represents a much greater testing effort, the results are more useful than simple-minded maximum performance numbers, such as those available for XenServer [Tolly 2008]. More recent VMware measurements have been documented for a database processing workload based on the TPC-C benchmark [TPC 2010]. Figure 13.13 compares ESX 4.0 with native OS scalability and Figure 13.14 compares ESX 4.0 and ESX 3.5 performance. Linear scalability up through eight VMs is now evident.

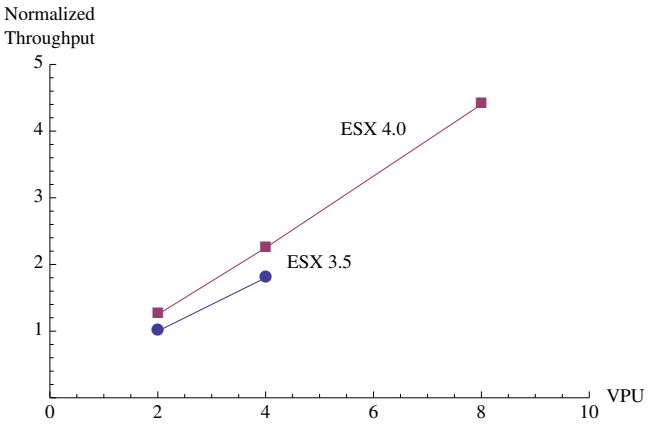


Fig. 13.14. Relative scalability of ESX 4.0 (upper curve) and ESX 3.5 (lower curve) executing the same workload as in Figure 13.13. An eight-way VPU configuration is not supported in ESX 3.5 [Source: VMware]

For these controlled measurements, the driver-side hardware comprised a single-socket, quad-core 2.50 GHz Intel E5420 ("Harpertown") processor with 4 GB of RAM. The eight-way ESX Server-side SUT (system under test) consisted of two 2.93 GHz quad-core Intel Xeon X5570 Nehalem processors, 36 GB of memory, with SMT and turbo mode disabled in the BIOS. The respective software configurations involved VMware ESX 4.0 Build # 136362, and VMware ESX 3.5 Update 3. The guest and native OS were RHEL 5.1 64-bit Linux, and the database management system was a trial version of Oracle 11g R1.

To ensure that performance comparisons between ESX Server and the native OS were as similar as possible, all tests were conducted with the number of physical CPUs used by ESX Server equal to the number of VPUs configured in the virtual machine.

13.5 Macro-VMM Scale: Clouds and P2P

In this section we consider virtualization associated with large-scale macro-VMs such as clouds and peer-to-peer (P2P) hypernet networks. The latter include Gnutella (Fig. 13.15), Napster, Freenet, Limewire, Kazaa, `SETI@Home`, BitTorrent, Skype, instant messaging, WiFi, PDAs and even cellphones. They have progressed from simple one-off file transfers to a scalable means for distribution of applications such as games, movies, and even operating systems.

Although P2P networks and clouds share the common focus of harnessing resources across multiple administrative domains, they can be distinguished as follows. Clouds support a variety of applications with a focus on providing infrastructure with quality-of-service to moderate-sized, homogeneous, and

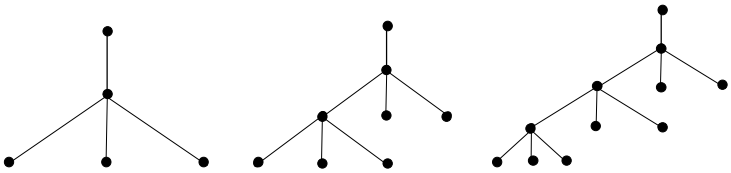


Fig. 13.15. Cayley trees with degree 4 vertices similar to those used in P2P networks like Gnutella and Napster

partially trusted communities [Foster 2005]. P2P supports intermittent participation in vertically integrated applications for much larger communities of untrusted, anonymous individuals. P2P systems provide protocols for sharing and exchanging data among nodes. The network architecture tends to be more decentralized, and dynamics requiring resource discovery.

GRID computing has focused on scientific and engineering applications, where it attempts to provide diverse resources that interoperate [Gilbert et al. 2005]. The concept behind the GRID is analogous to that of the electrical power grid. When you throw the switch, you expect the light to come on. GRID computing is most often discussed within the context of scientific and engineering applications because they are generally very CPU-intensive. ASCI BlueMountain, part of the ASCI-Grid with 6144 processors, employs FSS job scheduling [Kleban and Clearwater 2003]. The interested reader should see [Strong 2005] for an overview of the potential application of GRIDs in the commercial enterprise.

These technologies are not mutually exclusive. P2P technologies could be used to implement GRID systems that avoid or alleviate performance bottlenecks [Talia and Trunfio 2004]. Although these technologies are still rapidly evolving, applications are becoming more robust (it's not just about music files anymore), so capacity planners should prepare themselves for the occasion when these macro-VMs connect into your data center.

13.5.1 Macro-VM Polling

Polling protocols are employed by macro-VMs in at least two ways: for maintaining connectivity between peers and for security on the network. Each type of polling protocol has important ramifications for network performance and capacity. Although generally more nebulous and system-specific than micro-VM or meso-VM polling mechanisms, the particular case of wireless networks (see IEEE 802.11 standard) provides an illustrative example of their potential performance impact.

When carrying both voice and data, VoIP packets require contentionless periods in the transmission protocol, whereas data packets can tolerate contention (simple retry). Wireless access points poll, regardless of whether data is available for transmission or not. When the number of stations in the ser-

vice set grows, the polling overhead is known to become large. Without some kind of service differentiation, performance degrades. One enhancement that has been considered to increase network capacity is a polling list where idle nodes are dynamically deleted or active ones are added. This helps to increase the number of contentionless periods, thereby improving WLAN capacity by about 20%.

Polling to maintain P2P network security is employed in the sense of collecting opinions or votes. Providing security for distributed content sharing in P2P networks is an important challenge due to vulnerabilities in many protocols in sharing the "reputations" of peers. Certain polling protocols are subject to attacks which can alter the results of any voting procedure. Securing macro-VM networks has capacity planning implications.

The goal of macro-VMs is to enable scalable virtual organizations to provide a set of well-defined services. Key to performance is the network topology and its associated bandwidth. To assess the scalability of network bandwidth, this section draws on performance bounding techniques described in Chap. 7.

13.5.2 Scalability Analysis Using PDQ

The main results are summarized in Table 13.2, which shows each of the topologies ranked by their relative bandwidth. The 20-dimensional hypercube outranks all other contenders on the basis of query throughput. For a horizon containing two million peers, each servant must maintain 20 open connections, on average. This is well within the capacity limits of most TCP/IP implementations. The ten-dimensional hypertorus is comparable to the 20-hypercube in bandwidth up to a horizon of one million peers but falls off by almost 10% at two million peers.

Table 13.2. P2P hypernet topologies ranked by maximal relative bandwidth (BW), showing connections per peer (C/N), average number of network hops (H), and the number of supported peers (N) in millions

Hypernet Topology	C/N	H	$N \times 10^6$	BW
20-Cube	20	10	2.1	100
10-Torus	20	11	2.1	93
20-Cayley	20	6	2.8	16
8-Cayley (Napster)	8	8	1.1	13
4-Cayley (Gnutella)	4	13	1.1	8

The 20-valent Cayley tree is included since the number of connections per peer is the same as that for the 20-cube and the ten-torus. An horizon of six hops was used for comparison because the peer population is only 144,801 nodes at five hops. Similarly for eight-Cayley, a nine hop horizon would contain

7.7 million peers. These large increments are a direct consequence of the high vertex degree per node. The four-Cayley (early Gnutella network in Fig. 13.15) and eight-Cayley (Napster network) show relatively poor scalability at one million peers [Ritter 2002]. Even doubling the number of connections per peer produces slightly better than 50% improvement in throughput.

Because bandwidth in these topologies grows in proportion to added nodes or peers (Fig. 13.16), no throughput ceiling of the type appearing in Fig. 13.4 is observed. *BitTorrent* is a P2P file-sharing protocol which effectively implements higher-order topologies dynamically in software. Every client downloading a file from the network usually donates part of its own bandwidth, making it much faster than earlier P2P technologies like Gnutella or Kazaa.

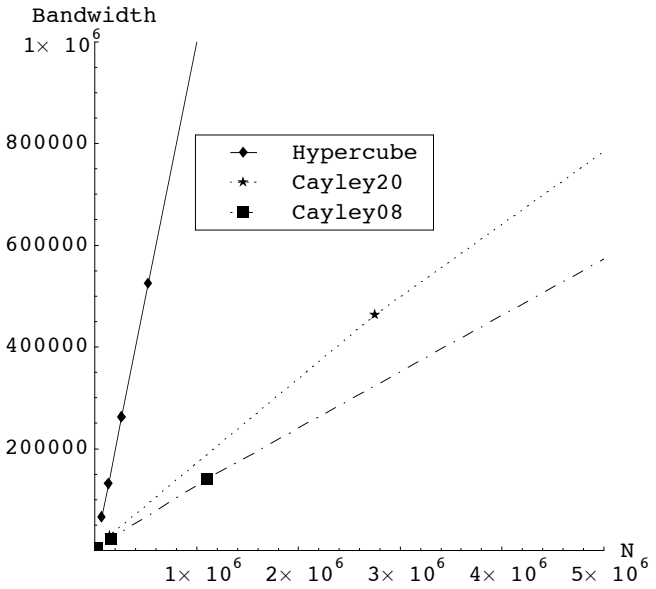


Fig. 13.16. Predicted bandwidth as a function of peers (N) for different hypernet topologies in Table 13.2

Though BitTorrent is a good protocol for broadband, it is less effective for dial-up, where dropped connections are common. On the other hand, many HTTP servers drop connections over several hours, while many torrents exist long enough to complete a multiday download often required for large files. An uploading client is flagged as snubbed if the downloading client has not received any data from it in over 60 seconds.

Some BitTorrent clients also report the *share ratio*, a number relating the amount of data uploaded to the amount downloaded. A share ratio of 1.0 means that a user has uploaded as much data as he has downloaded. Some

networks, for example, prevent access to new torrents for the first 24 to 48 hours (i.e., up to 1.7×10^5 seconds in Table 13.1) that the torrent is active to people with overall ratios of less than 1.0 and a certain amount of data uploaded.

13.6 Cloud Computing Models

A typical message-passing multicomputer architecture is depicted schematically in Fig. 13.17. In the context of the so-called *LogP* model, which we take up in more detail in Sect. 13.6.5, message-passing performance can be parameterized in terms of the interconnect latency (L), the message processing overhead (o), the message generation rate (g) and the number of physical processors (P). Hence the name.

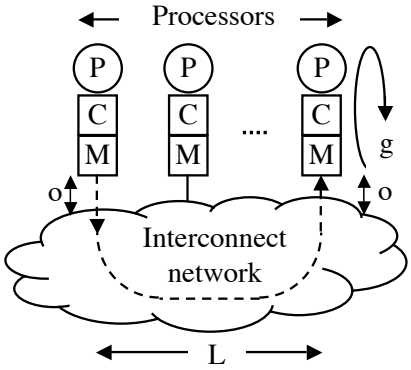


Fig. 13.17. Generic multicomputer with interconnection network comprising P processors with their respective caches C and local memories M. The parametric model is discussed in Sect. 13.6.5.

The particular case of a queueing model with just one routing stage ($k = 1$ in our notation) in the interconnect network is known as the *Machine Repairman Model* or MRM. The queueing concepts behind this model are discussed in Sects. 4.8.1, 4.8.3 and 4.11.12. In Sect. 13.6.5 we shall generalize the MRM to include interconnects with $k > 1$ stages but we will retain the acronym MRM for simplicity. Multi-hop stages inevitably require the use of queueing network solvers like PDQ.

The correspondence between the two diagrams can be easily understood as follows. The processing nodes labeled PCM in Fig. 13.17 become the infinite servers at the top of Fig. 13.18; the memory modules CM are not drawn explicitly. Similarly, the cloud labeled *interconnect network* in Fig. 13.17 is represented by the queueing center in the lower part of Fig. 13.18. An important attribute is that requests and responses circulate from the P processors

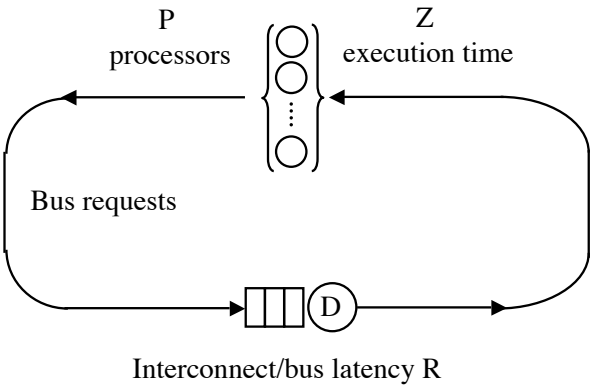


Fig. 13.18. Representation of the multicomputer in Fig. 13.17 as a queue-theoretic Machine Repairman Model (MRM). Machines correspond to processors (P) each with mean execution or "up" time Z. Machines that are "down" are queued at a repairman who takes a mean service time D. The queue length determines the corresponding interconnect network latency of the message-passing architecture. Cache and memory delays are also included in the queueing time R. See Table 13.3.

Table 13.3. MRM parameters associated with Fig. 13.18

P	Number of active processors
Z	Mean execution time at a processor
X	Mean system throughput
D_k	Mean interconnect latency for k routing stages
D_{max}	Bottleneck latency
$\mathcal{D}$	Minimum interconnect latency
R	Interconnect latency including waiting time

to the queue and then feed back to the processors. No requests enter or leave the MRM system. Moreover, the identity of which processor is sending and which is receiving messages is not enumerated since the system is evaluated in steady state. It is possible to extend the MRM to distinguish between requests and response by introducing multiple classes of traffic, but we shall not require that level of detail for the subsequent discussion.

The steady-state variables that define the performance of the MRM queueing model are defined in Table 13.3. From these parameters it follows that

$$D_{max} = Max(D_1, D_2, \ldots, D_k) \tag{13.7}$$

and the minimum network latency is

$$\mathcal{D} = \sum_k D_k. \tag{13.8}$$

The mean system throughput in Fig. 13.18 is defined by

$$X(P) = \frac{P}{R(P) + Z}. \tag{13.9}$$

Both X and R are implicit functions of P, which is reflective of the feedback flow in Fig. 13.18, so (13.9) must be calculated using an analytic queueing solver like PDQ. Here, however, we are mostly interested in performance bounds rather than the full performance characteristics. One such bound is the maximum achievable throughput

$$X_{max}(P) = \frac{1}{D_{max}}. \tag{13.10}$$

which is controlled by the bottleneck latency.

Another throughput bound is due to *synchronous* queueing where all P processors suspend execution and issue a request message simultaneously. Then, the latency R to traverse the interconnect is the product of the mean time $\mathcal{D}$ that it takes to route each message and the total number of messages P in transit i.e., $R(P) = P\mathcal{D}$. Substituting into (13.9) produces

$$Xsyn(P) = \frac{P}{P\mathcal{D} + Z}. \tag{13.11}$$

Although this bound is known in queueing theory [Lazowska et al. 1984], its connection with Amdahl's law, which we derive next, seems not to have been recognized previously.

13.6.1 Fixed-Size Bounds

An empirical measure of parallel performance is the *speedup* ratio

$$S(P) = \frac{T_1}{T_P} \tag{13.12}$$

where T_P is the elapsed time on P processors. The elapsed time T is equivalent to the execution time T_1 on a uniprocessor. The remaining time T_P can then be reduced by partitioning the application across P processors running in parallel. Symbolically,

$$T_1 = T \text{ and } T_P = \sigma T + (1 - \sigma)\frac{T}{P}. \tag{13.13}$$

Substituting (13.13) into (13.12) and simplifying produces Amdahl's law [Amdahl 1967, Ware 1972, Preparata 1995, Argentini 2002]

$$S_A(P) = \frac{P}{1 + \sigma\ (P - 1)}, \tag{13.14}$$

for a fixed-size workload with the percentage of time spent in uniprocessor mode expressed in terms of a single parameter (σ), known as the *serial fraction*, having range $0 \leq \sigma \leq 1$. In the infinite processor limit, the speedup (13.14) becomes

$$\lim_{P \to \infty} S_A(P) = \frac{1}{\sigma}. \tag{13.15}$$

This limit can be interpreted as the best achievable speedup when σ^{-1} processors are running 100% busy. For example, if the serial fraction $\sigma = 0.10$ then the speedup limit corresponds to ten saturated processors.

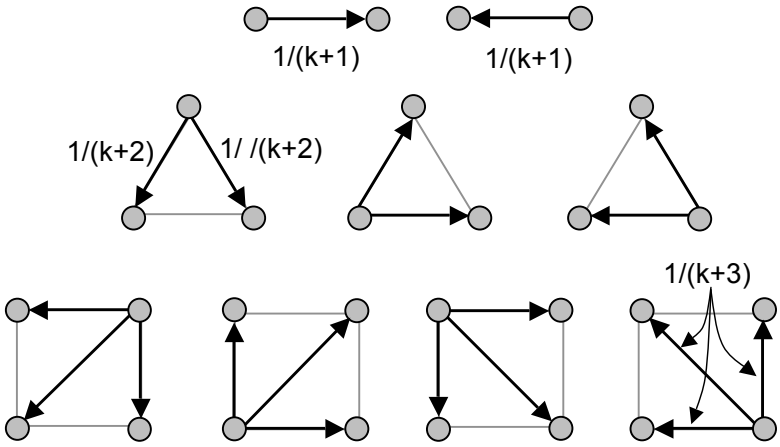


Fig. 13.19. Pictorial representation of Amdahl's law for $P = 2, 3$, and 4 processors (vertices) where each arrow represents the direction of the current communication cycle with rational overhead $\sigma = 1/k$

A more physically revealing form of Amdahl's law can be written in terms of the message-exchange diagrams shown in Fig. 13.19. Then, (13.14) can be re-expressed as the difference between ideal linear speedup and an associated inter-processor messaging overhead:

$$S_A(P, k) = P - G(P, k)P(P - 1). \tag{13.16}$$

Here, $k = 1/\sigma$, and the cost of inter-processor messaging

$$G(P, k) = \frac{1}{k + (P - 1)} \tag{13.17}$$

corresponds to the Pth term of a *harmonic series* (see Table 13.4) and is depicted by the arrows in Fig. 13.19. If $P = 4$ and the serial fraction $\sigma = 1/7$, then both (13.14) and (13.16) predict a speedup of $S_A(4) = 2.80$ where the second term in (13.16) corresponds to the four-node diagrams in the last row of Fig. 13.19.

Equation (13.16) can also be interpreted as a kind of *broadcast* interaction where the requesting processor causes every other processor to halt execution and listen to the message [Gunther 1996]. The processors then respond

Table 13.4. Amdahl cost function $G(P)$ expressed in terms of the parameter k and the serial fraction σ

P	1	2	3	4	5
$G_k(P)$	$\frac{1}{k}$	$\frac{1}{k+1}$	$\frac{1}{k+2}$	$\frac{1}{k+3}$	$\frac{1}{k+4}$
$G_\sigma(P)$	σ	$\frac{\sigma}{1+\sigma}$	$\frac{\sigma}{1+2\sigma}$	$\frac{\sigma}{1+3\sigma}$	$\frac{\sigma}{1+4\sigma}$

cyclicly in the same way. Although the broadcast cost in Table 13.4 for a single processor $G(1)$ is nonzero under this interpretation, the total overhead in (13.16) is zero due to the $(P-1)$ factor in the second term. The broadcast interpretation does not represent an *efficient* interaction, only a logically *correct* interaction associated with Amdahl's law. Next, we show that (13.14) and (13.16) have a related physical interpretation under the MRM model.

To see this connection, let

$$\sigma = \frac{\mathcal{D}}{\mathcal{D}+Z} \tag{13.18}$$

such that the range of σ values is determined by MRM variables $\mathcal{D}$ and Z:

$$\sigma \to \begin{cases} 0 \text{ as } \mathcal{D} \to 0 \text{ with } Z = \text{const.} \\ \\ 1 \text{ as } Z \to 0 \text{ with } \mathcal{D} = \text{ const.} \end{cases} \tag{13.19}$$

When $\sigma = 0$ the interconnect latency $\mathcal{D}$ is zero because there is a maximal execution time Z with no messages exchanged between processors. Consequently, there cannot be any queueing contention in Fig. 13.18. Conversely, $\sigma = 1$ corresponds to zero execution time and maximal queueing latency on the communication network.

Substituting (13.18) into (13.14) produces

$$S_A(P) = \frac{P(\mathcal{D}+Z)}{P\mathcal{D}+Z}. \tag{13.20}$$

which can be interpreted immediately in terms of the synchronous throughput (13.11). Amdahl speedup (13.20) is the ratio of the synchronous throughput with P processors to the synchronous throughput on a single $(P = 1)$ processor. Amdahl's law therefore corresponds to worst-case queueing in our MRM model. It is the speedup bound for synchronous messaging, which can also be regarded as the queueing analog of the broadcast protocol defined by (13.16). Once again, we are not trying to identify the most efficient messaging protocol but rather the correct dynamics expressed by Amdahl's law.

13.6.2 Harmonic Bounds

Although Amdahl's law constitutes the worst-case queueing bound in MRM parlance, even lower bounds on speedup do exist [Flynn 1995, Gelenbe 1989]. For the sake of completeness, we briefly consider how they can be interpreted within the context of the MRM model.

If the workload is equally likely to make use of any subset of processors (equipartitioning), the speedup becomes

$$S_H(P) = \frac{P}{1 + \frac{1}{2} + \frac{1}{3} + \ldots \frac{1}{P}}. \tag{13.21}$$

Since the harmonic series in the denominator can be approximated by

$$\sum_{n=1}^{P} \frac{1}{n} \simeq \ln(P), \tag{13.22}$$

an upper bound on the equipartitioned speedup is

$$S_H(P) \simeq \frac{P}{\ln(P)}. \tag{13.23}$$

From the standpoint of MRM, S_H has to be viewed as a *worse than worst case* speedup bound ($S_H \ll S_A$) as evidenced so graphically in Fig. 13.20. Benchmark measurements that conform to (13.23) are likely to be a signal that serious performance tuning is required, although exceptions can arise e.g., a divide-and-conquer algorithm might produce this kind of speedup characteristic where the logarithm in (13.23) is expressed in base 2.

If instead of the harmonic sum (13.22) of *all* possible processing subsets, we consider the *harmonic mean* μ_{H_P} of two extreme subsets of processors

$$\frac{1}{\mu_{H_2}} = \frac{1}{2}\left(\frac{1}{P_1} + \frac{1}{P_2}\right), \tag{13.24}$$

where $P_1 = \kappa$ is a CPU-saturated subset of the entire parallel set $P_2 = P$, then

$$\mu_{H2} = \frac{2\,\kappa P}{\kappa + P}. \tag{13.25}$$

Replacing the constant κ with the inverse of the serial fraction σ, we find that (13.25) is related to (13.14) by

$$S_A(P) \simeq \frac{1}{2}\,\mu_{H2}. \tag{13.26}$$

In other words, the asynchronous Amdahl bound is extremely well approximated (to within 1% error in Fig. 13.20) by half the harmonic mean of the maximally parallel and maximally saturated processor subsets.

This connection between Amdahl's law and harmonic bounds (13.23) and (13.25) is discussed in [Flynn 1995, Chap. 8] and [Gelenbe 1989, Chap. 3] but otherwise seems not to be widely known. The MRM model makes it clear via the appropriate choice of parameter mappings (Fig. 13.22).

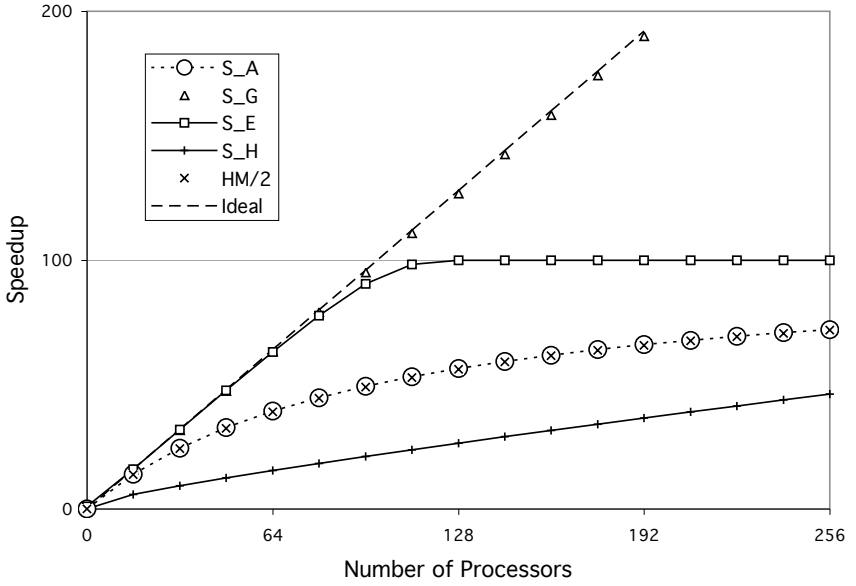


Fig. 13.20. Speedup bounds for $\sigma = 0.01$. S_A is the synchronous speedup; the harmonic mean (13.26) is superimposed on the same curve, S_E is the asynchronous speedup, S_G is the near-linear scaled speedup and the lowest curve S_H is the equipartitioned speedup (13.23)

13.6.3 Scaled-Size Bounds

It has been recognized for some time that parallel speedup predicted for real problems can differ from that predicted by Amdahl's law because the percentage of time (σ) spent in sequential sections of the program can depend on the size of the problem. Scaling up the problem size can improve speedup as follows.

If the serial fraction is assumed to vary linearly with P, the speedup (13.12) becomes

$$S(P) = \frac{T_1}{T_P} = \frac{\sigma + (1-\sigma)P}{\sigma + (1-\sigma)}, \tag{13.27}$$

which simplifies to

$$S_G(P) = P + \sigma(1-P). \tag{13.28}$$

By analogy with Sect. 13.6.1, (13.28) is sometimes referred to as the *Gustafson-Baris law* [Gustafson 1988], which has been demonstrated for certain parallel applications [Gustafson et al. 1988].

The relationship to the MRM model is easily determined by substituting (13.18) into (13.28) to produce

$$S_G(P) = \frac{\mathcal{D} + PZ}{\mathcal{D} + Z}. \tag{13.29}$$

If we invoke the change of MRM variable $Z \mapsto Z'/P$, (13.29) becomes

$$S_G(P) = \frac{P(\mathcal{D} + Z')}{P\mathcal{D} + Z'}, \tag{13.30}$$

which is identical to (13.20). Once again, we have arrived at a well-known parametric performance model by identifying the correct set of variables in the unifying MRM model (see Fig. 13.22).

The MRM interpretation of Gustafson's law (13.28) is that rescaling offsets the impact of the synchronous queueing by inflating the mean execution time Z in direct proportion to the number of processors P. In other words, rescaling the problem size minimizes message traffic on the network.

13.6.4 Erlang Model

The near-linear speedups implied by (13.28) are generally extremely difficult to achieve in practice. Based on the MRM analysis of Sect. 13.6.1, a more attainable compromise would be asynchronous messaging. The asynchronous speedup can be determined from the queueing formula

$$S_E(P) = \frac{1}{\sigma}\left\{1 - E_B\left(\frac{1-\sigma}{\sigma},\, P\right)\right\}, \tag{13.31}$$

where $E_B(A, P)$ is the Erlang B function (4.72) with $A = Z/\mathcal{D}$ the *service ratio* for Fig. 13.18. We have used the definition of σ in (13.18) to rewrite the service ratio as $A = (1 - \sigma)/\sigma$ in (13.31).

The factor $1 - E_B$ in (13.31) is the probability that the interconnect network is busy. It is usually more convenient to compute $E_B(A, P)$ using the iterative algorithm

```
erlangB = A / (1 + A);
for (k  = 2; k <= P; k++) {
    erlangB *= A / (A * erlangB + k);
}
```

which accumulates the result in the variable `erlangB` (cf. Listing 4.6 in Chap. 4).

The idea that asynchronous speedup effects can be predicted using (13.31) seems to be entirely novel and is a direct consequence of realizing that $S_A(P)$ in Sect. 13.6.1 describes the synchronous MRM dynamics of Fig. 13.18. A comparison of S_E and S_A is provided in Fig. 13.20 for a serial fraction of $\sigma = 0.01$. It is noteworthy that, although S_E offers greater speedup due to asynchronous messaging, the infinite processor asymptote $S_\infty = 100$ is reached with a smaller configuration of physical processors ($P < 128$) than with S_A. The reason for this can easily be explained in terms of MRM queueing effects (Fig. 13.22).

From Sect. 13.6.1, S_A is associated with the lower bound on the normalized throughput due to maximal synchronous queueing in the network. Under light

messaging loads ($P < 64$ in Fig. 13.20), the speedup is essentially linear rising because the asynchronous messaging due to each additional processor creates relatively little queueing contention in the network. However, at some point ($P \geq 128$ in Fig. 13.20) the network saturates (i.e., reaches 100% busy routing messages) and becomes the bottleneck.

In Sect. 13.6 the bottleneck was identified with D_{max} (the longest of the routing times for a k-stage interconnect) because it controls the maximum available throughput by virtue of (13.10). Any messages coming from faster routers upstream simply make the bottleneck queue longer, while faster downstream routers remain underutilized, waiting for completions at the bottleneck. In Fig. 13.20, D_{max} is associated with the normalized throughput, becoming bounded at $S_\infty = 100$.

13.6.5 LogP Model

LogP [Culler et al. 1996] is a latency model rather than a speedup model. Unlike the single parameter speedup models we have discussed so far, LogP has four parameters which can be measured directly [Kielmann et al. 2000]. Because of the correspondence between Figs. 13.17 and 13.18, however, the mapping between the original LogP parameters and the MRM variables of Sect. 13.6 is the most direct of all and can be summarized as follows:

$$LogP \equiv \begin{cases} L \text{ Latency} & \mapsto \mathcal{D} \text{ or } R \\ o \text{ Overhead} & \mapsto Z_{min}/2 \\ g \text{ Generate} & \mapsto Z/P \\ P \text{ Processors} & \mapsto P \end{cases} \tag{13.32}$$

Taken collectively, these parameters (which also annotate Fig. 13.17) give this latency model its name.

Although LogP intrinsically involves asynchronous messaging, the original formulation of the model [Culler et al. 1996] had no communication contention i.e., no queueing ($L \mapsto \mathcal{D}$) in MRM parlance. Taking as an example the measured nCUBE2 parameters reported in [Culler et al. 1996], with message size $M = 160$ bits, channel width $W = 1$ bit, mean routing hops $H = 5$, mean network latency $L = 360$ cycles and $g \equiv 2o = 6400$ cycles, the minimum round-trip time is

$$RTT_{min} = L + g = 6,700 \text{ cycles}. \tag{13.33}$$

This is equivalent to the MRM response time

$$R = \mathcal{D} + \frac{Z}{P} \tag{13.34}$$

with zero contention i.e., $P = 1$, where it is assumed that there can be no more than one outstanding message per processor in transit. Moreover, since we know $\mathcal{D}$ and Z from (13.34), we can apply (13.18) to estimate that the serial

contention is $\sigma = 0.0274$ for the nCUBE2. Surprisingly, even though LogP is traditionally used as a latency model, we can assess its speedup characteristics using the MRM mappings in Fig. 13.22.

MRM can provide deeper insight into LogP contention effects but, as noted in Sect. 13.6, with multiple queueing stages representing a multi-hop network we need to use an analytic queueing solver like PDQ, since each stage could have different service times D_k (cf. Table 13.3). With such tools we can predict the full latency characteristics for the nCUBE2 in Fig. 13.21 with $k = 5$ hops.

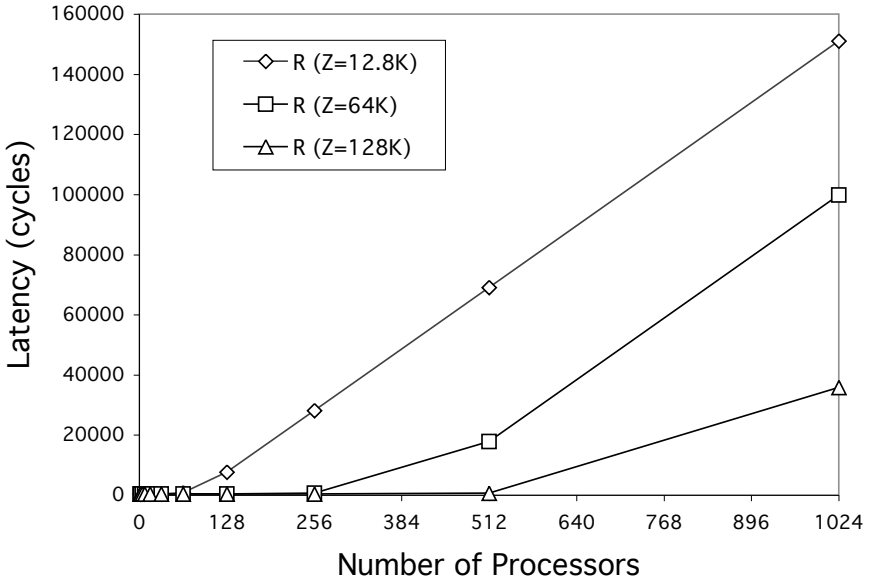


Fig. 13.21. Latency characteristics computed using the MRM generalization of the LogP model of nCUBE2

The lowest latency curve (bottom) corresponds to a mean execution time of $Z = 128,000$ cycles, while the highest latency curve (top) corresponds to $Z = 12,800$ cycles; the longer execution time producing fewer messages. Even for the latter case, however, the RTT grows from 360 cycles at $P = 1$ to almost 36,000 cycles under contention from $P = 1024$ processors. The ability to overlap messages could significantly reduce this contention.

In MRM, the *optimal processor configuration* is associated with the knee in the classic "hockey stick" characteristic; like those shown in Fig. 13.21. That knee can be predicted from

$$P_{opt} = \frac{\mathcal{D} + Z}{D_{max}}. \tag{13.35}$$

the ratio of the minimum RTT to the bottleneck latency (see Sect. 13.6). Mapping the MRM variables back to the LogP model, we find (13.35) corresponds to

$$\frac{L + 2o}{\lceil M/W \rceil}, \tag{13.36}$$

which is the ratio of the minimum RTT in (13.33) to the bisection bandwidth [Culler et al. 1996]. For the nCUBE2 configurations in Fig. 13.21, $P_{opt}(Z = 12.8K) = 81$, $P_{opt}(Z = 64K) = 401$, and $P_{opt}(Z = 128K) = 801$ processors, respectively; well below the $P = 1024$ available.

Extensions to LogP such as LogGP, accounting for longer messages [Alexandrov et al. 1997], the inclusion of memory models [Cameron and Sun 2003], and applications to MPI [Byna et al. 2003, Kielmann et al. 2000], can also be included as extensions to MRM but we do not pursue them here.

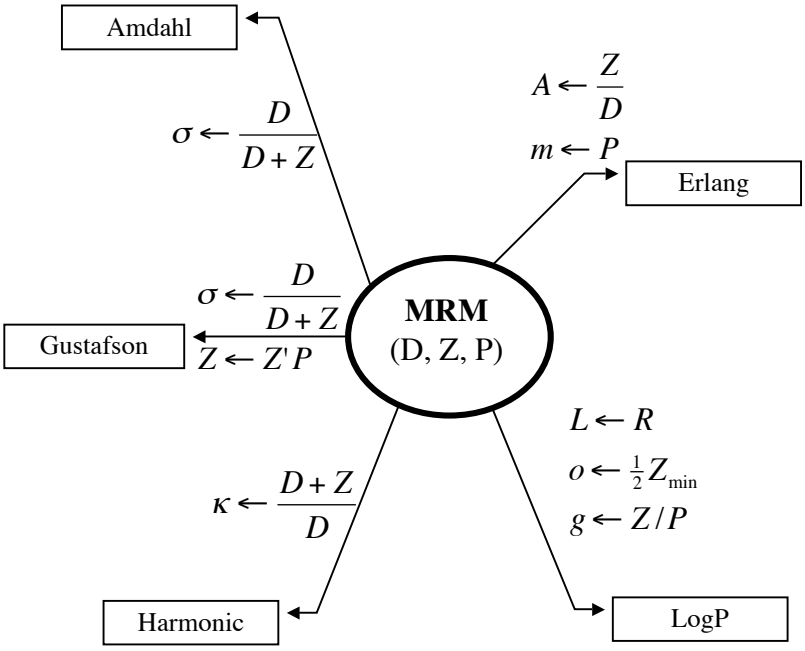


Fig. 13.22. Mappings between MRM variables and parametric model parameters

13.7 Summary

In this chapter, we have seen that virtualization is really a *spectrum* of technologies, many of which may coexist in the same datacenter, e.g., hyper-

threading, hypervisors, and private clouds. The usual tendency is to focus on only one of these forms of virtualization. This rather narrow view can be counterproductive when it comes to enterprise performance management. It is better think *holistically* when it comes to virtualization. It is also important to understand the basic principles of operation for a fair-share scheduler (FSS) since it forms the infrastructure for all hypervisors or meso-VMM virtualization. This important point has not been well documented previously in the context of virtualization and that's why a good part of Section 13.4 was devoted to it.

The best way to assess provisioning and consolidation of hardware and applications, especially for micro-VMMs and meso-VMMs, is through *controlled measurements*; not just through performance data collected from monitors like `esxtop` on production systems. Recent VMware ESX Server measurements clearly demonstrate how much the overheads due to virtualization are continually being reduced; especially for CPU-intensive workloads.

A cautionary note regarding consolidation onto virtualized servers: The focus often tends to be on maximizing processor utilization, and this is certainly a reasonable first-order measure of server capacity consumption; however, one should not lose sight of response-time targets. For example, an application might need to run at no higher than 10% processor utilization in order to meet the SLO (service-level objective) for that application. Consolidating that application with other applications onto a virtualized server may well achieve better capacity utilization, running at higher server utilization, but that could also make it impossible to reach the response time SLOs.

Finally, as cloud computing services mature, you can expect that the same performance management principles we have discussed here will also be applicable. Putting them into practice now will put you ahead of the virtualization game.