Predicting Disk Scheduling Performance with Virtual Machines

Robert Geist, Zachary H. Jones, James Westall

Clemson University

Abstract. A method for predicting the performance of disk scheduling algorithms on real machines using only their performance on virtual machines is suggested. The method uses a dynamically loaded kernel intercept probe (*iprobe*) to adjust low-level virtual device timing to match that of a simple model derived from the real device. An example is provided in which the performance of a newly proposed disk scheduling algorithm is compared with that of standard Linux algorithms. The advantage of the proposed method is that reasonable performance predictions may be made without dedicated measurement platforms and with only relatively limited knowledge of the performance characteristics of the targeted devices.

1 Introduction

In the last five years, the use of virtual computing systems has grown rapidly, from nearly non-existent to commonplace. Nevertheless, system virtualization has been of interest to the computing community since at least the mid-1960s, when IBM developed the CP/CMS (Control Program/Conversational Monitor System or Cambridge Monitor System) for the IBM 360/67 [1]. In this design, a low-level software system called a *hypervisor* or *virtual machine monitor* sits between the hardware and multiple guest operating systems, each of which runs unmodified. The hypervisor handles scheduling and memory management. *Privileged* instructions, those that trap if executed in user mode, are simulated by the hypervisor's trap handlers when executed by a guest OS.

The specific architecture of the host machine essentially determines the difficulty of constructing such a hypervisor. Popek and Goldberg [2] characterize as *sensitive* those machine instructions that may modify or read resource configuration data. They show that an architecture is most readily virtualized if the sensitive instructions are a subset of the privileged instructions.

The principal roadblock to widespread deployment of virtual systems has been the basic Intel x86 architecture, the de facto standard, in which a relatively large collection of instructions are sensitive but not privileged. Thus a guest OS running at privilege level 3 may execute one of them without generating a trap that would allow the hypervisor to virtualize the effect of the instruction. A detailed analysis of the challenges to virtualization presented by these instructions is given by Robin and Irvine [3].

The designers of VMWare provided the first solution to this problem by using a binary translation of guest OS code [4]. Xen [5] provided an open-source virtualization of the x86 using *para-virtualization*, in which the hypervisor provides a virtual machine interface that is similar to the hardware interface but avoids the instructions whose virtualization would be problematic. Each guest OS must then be modified to run on the virtual machine interface.

The true catalysts to the widespread development and deployment of virtual machines appeared in 2005-2006 as extensions to the basic x86 architecture, the Intel VT-x and AMD-V. The extensions include a "guest" operating mode, which carries all the privilege levels of the normal operating mode, except that system software can request that certain instructions be trapped. The hardware state switch to/from guest mode includes control registers, segment registers, and instruction pointer. Exit from guest mode includes the cause of the exit [6]. These extensions have allowed the development of a full virtualization Xen, in which the guest operating systems can run unmodified, and the Kernel-based Virtual Machine (KVM), which uses a standard Linux kernel as hypervisor.

Nevertheless, VMWare, Xen, and KVM are principally aimed at facilitating user-level applications, and thus they export only a fairly generic view of the guest operating system(s), in which the devices to be managed are taken from a small, fixed collection of emulated components. This would seem to preclude the use of virtual machines in testing system-level performance, such as in a comparative study of scheduling algorithms. The devices of interest are unlikely to be among those emulated, and, even if they are, the emulated devices may have arbitrary implementation, thus exhibiting performance characteristics that bear little or no resemblance to those of the real devices. For example, an entire virtual disk may be cached in the main memory of a large NAS device supporting the virtual machine, thus providing a disk with constant ($O(1)$) service times.

In [7], we introduced the *iprobe*, an extremely light-weight extension to the Linux kernel that can be dynamically loaded and yet allows the interception and replacement of arbitrary kernel functions. The *iprobe* was seen to allow a straightforward implementation of PCI device emulators that could be dynamically loaded and yet were suitable for full, system-level driver design, development, and testing.

The goal of this effort is to extend the use of the *iprobe* to allow performance prediction for real devices using only virtual machines. We suggested this possibility in [8]. In particular, we will predict the performance of a collection of disk scheduling algorithms, one of which is new, for a targeted file-server workload on a specific SCSI device. We then compare the results with measurements of the same algorithms on the real hardware. We will see that the advantage of our approach is that reasonable performance predictions may be made without dedicated measurement platforms and with relatively limited knowledge of the performance characteristics of the targeted devices.

The remainder of the paper is organized as follows. In the next section we provide background on both kernel probes and disk scheduling. In section 3, we propose a new disk scheduling algorithm that will serve as the focus of our tests.

In section 4, we show how the kernel probes may be used to implement a *virtual performance throttle*, the mechanism that allows performance prediction from virtual platforms. In section 5 we describe our virtual machine, the targeted (real) hardware, and a test workload. Section 6 contains results of algorithm performance prediction from the virtual machine and measurements of the same algorithms on the real hardware. Conclusions follow in section 7.

2 Background

2.1 Kernel Probes

The Linux kernel probe or *kprobe* utility was designed to facilitate kernel debugging [10]. All *kprobes* have the same basic operation. A *kprobe* structure is initialized, usually by a kernel module, i.e., a dynamically loaded kernel extension, to identify a target instruction and specify both pre-handler and post-handler functions. When the *kprobe* is registered, it saves the targeted instruction and replaces it with a breakpoint. When the breakpoint is hit, the pre-handler is executed, then the saved instruction is executed in single step mode, then the post-handler is executed. A return resumes execution after the breakpoint.

A variation on the *kprobe*, also supplied with Linux, is the *jprobe*, or jump probe, which is intended for probing function calls, rather than arbitrary kernel instructions. It is a *kprobe* with a two-stage pre-handler and an empty post-handler. On registration, it copies the first instruction of the registered function and replaces that with the breakpoint. When this breakpoint is hit, the first-stage pre-handler, which is a fixed code sequence, is invoked. It copies both registers and stack, in addition to loading the saved instruction pointer with the address of the supplied, second-stage pre-handler. The second-stage pre-handler then sees the same register values and stack as the original function.

In [7], we introduced the intercept probe or *iprobe*, which is a modified *jprobe*. Our *iprobe* second-stage pre-handler decides whether or not to replace the original function. If it decides to do so, it makes a backup copy of the saved (function entry) instruction and then overwrites the saved instruction with a no-op. As is standard with a *jprobe*, the second-stage pre-handler then executes a *jprobe* return, which traps again to restore the original register values and stack. The saved instruction (which now could be a no-op) is then executed in single step mode. Next the post-handler runs. On a conventional *jprobe*, this is empty, but on the *iprobe*, the post-handler checks to see if replacement was called for by the second-stage pre-handler. If this is the case, the single-stepped instruction was a no-op. The registers and stack necessarily match those of the original function call. We simply load the instruction pointer with the address of the replacement function, restore the saved instruction from the backup copy (overwrite the no-op) and return. With this method, we can intercept and dynamically replace any kernel function of our choice. Note that it is possible to have two calls to the same probed function, one that is to be intercepted and one that is not. A discussion of the handling of potential attendant race conditions in SMP systems may be found in [9].

2.2 Disk Scheduling

Scheduling algorithms that re-order pending requests for disks have been studied for at least four decades. Such scheduling algorithms represent a particularly attractive area for investigation in that the algorithms are not constrained to be work-conserving. Further, it is easy to dismiss naive (but commonly held) beliefs about such scheduling, in particular, that a greedy or shortest-access-time-first algorithm will deliver performance that is optimal with respect to any common performance measure, such as mean service time or mean response time. Consider a hypothetical system in which requests are identified by their starting blocks and service time between blocks is equal to distance. Suppose the read/write head is on block 100 and requests in queue are for blocks 20, 82, 120, and 200. The greedy schedule and the differing, optimal schedule are shown in Table 1.

algorithm	mean service	mean response
greedy 82, 120, 200, 20	79.0	131.5
optimal 120, 82, 20, 200	75.0	124.5

Table 1. Sub-optimal performance of the greedy algorithm.

Disk scheduling algorithms are well-known to be analytically intractable with respect to estimating response time moments. Early successes in this area, due to Coffman and Hofri [10] and Coffman and Gilbert [11] were restricted to highly idealized, *polling* servers, in which the read/write head sweeps back and forth across all the cylinders, without regard to the extent of the requests that are actually queued.

Almost all knowledge of the performance of real schedulers is derived from simulation and measurement studies. Geist and Daniel described UNIX system measurements of the performance of a collection of "mixture" algorithms that blended scanning and greedy behavior [12]. Worthington, Ganger, and Patt [13] showed, in simulation, that scheduling with full knowledge of disk subsystem timing delays, including rotational delays and cache operations, could offer major performance improvements. They also concluded that knowledge of the cache operation was far more important than an accurate mapping of logical block to physical sector, a point which we will address.

More recently, Pratt and Heger [14] provided a comparison of the four schedulers distributed with Linux 2.6 kernels. Until 2.6, Linux used a uni-directional or circular scan (CSCAN), in which requests are served in ascending order of logical block number until none remains, whereupon the read/write head sweeps back down to the lowest-numbered pending request. With recognition that the best scheduler is likely workload-dependent, Linux authors changed the 2.6 kernel to allow single-file, modular, drop-in schedulers that could be dynamically switched. Four schedulers were provided. The default is the completely fair queueing (*cfq*) algorithm, which has origins in network scheduling. Each process has its own logical queue, and requests at the front of each queue are batched, sorted and served.

The *deadline* scheduler was designed to limit response time variance. Each request sits in two queues, one sorted by CSCAN order, one FIFO, and each has a deadline. The CSCAN order is used, unless a deadline would be violated, and then FIFO is used. As with many algorithms, reads are separated from and given priority over writes because the requesting read process has usually suspended to await I/O completion, and so there are actually four (logical) queues. The *anticipatory* scheduler is no longer supported, and the *noop* scheduler is essentially FIFO, which delivers poor performance on almost all workloads, and thus these two will not be discussed.

A fundamental departure from greedy algorithms, scanning algorithms, and $O(N)$ mixtures thereof was offered by Geist and Ross [15]. They observed that, over the preceding decades, CPU speeds had increased by several orders of magnitude while disk speeds remained essentially unchanged. They suggested that $O(N^2)$ algorithms might be competitive and offered a statically optimal solution that was based on Bellman's *dynamic programming* [16], in which a table of size $O(N^2)$ containing optimal completion sequences was constructed. Although their algorithm was shown to deliver excellent performance in tests on a real system, there were two easily-identifiable problems. It ignored the dynamics of the arrival process, and it ignored the effects of any on-board disk cache.

More recently, Geist, Steele, and Westall [17] partially addressed the issue of arrival dynamics. Although at first glance counter-intuitive, it is often beneficial for disk schedulers with a non-empty queue of pending requests to do nothing at all [18]. The process that issued the most recently serviced request is often likely to issue another request for a nearby sector, and that request could be served with almost no additional effort. They added a so-called *busdriver* delay, to mimic the actions of a bus driver who would wait at a stop for additional riders, to the table-based, dynamic programming algorithm of Geist and Ross, and showed that it delivered excellent performance, superior to any of the four schedulers distributed with Linux 2.6, for a fairly generic, web file-server workload designed by Barford and Crovella [19].

3 A New Scheduler

We now propose an extension of Geist-Steele-Westall algorithm to capture the effects of the on-board cache. On-board caches are common, although their effect on the performance of standard workloads is often minimal. All UNIX-derivative operating systems allocate a significant portion of main memory to I/O caching. In Linux this is called the page buffer cache. The file systems also issue readahead requests when they detect sequential reads of a file's logical blocks. Since the page buffer cache is usually an order of magnitude larger than any on-board disk cache, most of the benefits of caching are captured there. Nevertheless, users can force individual processes to avoid the page buffer cache, and so there are cases where the on-board disk cache could have significant effect.

We model the effects of the on-board cache with just three parameters, the number of segments, the number of sectors per segment, and the pre-fetch size, also in sectors. We assume the cache is fully associative with FIFO replacement, and, should a request exceed the segment size, we assume wrap-around. Although manufacturers are often secretive about the operations of on-board disk caches, the minimal information we require can usually be obtained from the SCSI mode page commands. The sdparm utility [20] is a convenient tool for accessing such.

With this information we maintain a shadow cache within the scheduler. Upon entry to the scheduler, even if a previously computed optimal sequence is still valid and no $O(N^2)$ table-building would be required, we check the entire arrival queue of pending requests against the shadow cache for predicted cache hits. If any request is found to be a predicted hit, it is scheduled immediately.

The shadow cache comprises pairs of integers denoting the start and end sectors of the span assumed to be contained within each segment. When a request is dispatched, the pair (start sector, end sector + pre-fetech) is written into the shadow cache at the current segment index, and the current segment index is advanced. To check for a predicted read hit, the requested span is compared against the spans contained in each segment.

Finally, we use a form of soft deadline to control response time variance. Reads are given priority over writes, and only reads use the table-building algorithm. Writes are served in CSCAN order. If the oldest pending read request exceeds a maximum READDELAY parameter, we forgo table-building and serve reads in CSCAN order as well, until the oldest reader no longer exceeds this age. The deadline is soft because it only guarantees service within the next sweep.

4 A Virtual Performance Throttle

We now describe how the real performance of these algorithms can be predicted using a virtual machine. We start by specifying a service time model for the targeted physical drive. A highly accurate model would require a detailed mapping of logical to physical blocks, which, for modern disks, is often serpentine in nature [21]. Within a single track, consecutive logical sectors usually map directly to consecutive physical sectors, but the mapping of logical tracks to physical tracks is considerably more complex because cylinder seek time is now less than head switching time. As a result, logical tracks are laid out in bands that allow multiple cylinder seeks per head switch when the disk is read in a logically sequential way. Nevertheless, as we will illustrate in the next section, at the macroscopic level, where seek distances are measured in millions of sectors, a linear model will suffice. Thus we assume we have a service time model of the form $X_r = R_r/2 + S_r(d_r/D_r)$, where R_r is rotation, S_r is maximum seek time, and D_r is maximum seek distance.

The idea of the virtual performance throttle (VPT) is to insert an *iprobe* into the SCSI path of the virtual system to force virtual service times that are proportional to real ones, with an identifiable constant of proportionality. If the virtual seek distance is d_v with maximum D_v, then we want the virtual system to deliver a service time of kX_r, where $d_r/D_r = d_v/D_v$, for some system-wide constant, k. Instead, it will deliver a service time of X_v. Clearly, we can achieve our goal by delaying the completed virtual request by $kX_r - X_v$, except that k is unknown to us. Further, due to factors outside our control, such as server load and network congestion on the hardware supporting the virtual machine, an appropriate k may change during the course of our tests. Thus we need a self-scaling system.

We specify an initial value of k and allow the VPT to constrain the flow of disk requests being served based on this value and the linear service time model. The VPT uses two kernel probes in the generic SCSI driver: a *jprobe* in the down path to record when requests leave for the virtual disk and an *iprobe* in the up path to intercept and delay completed requests upon return from the virtual disk. The *jprobe* calculates the target completion time of the request and passes it to the *iprobe*, which then determines how long the request should be delayed after its completion but before returning it to the requesting process. The *iprobe's* queue of delayed completions is checked periodically with a timer. Once the target completion time has passed, the request is injected back into the SCSI generic path.

As noted, the service time on the virtual disk is subject to change, and so it is possible that a request will arrive to the *iprobe* after its target completion time. This means that the scale factor, k, is too small. The *iprobe* increases it and passes the new value to the *jprobe*. Similarly, if the VPT develops a large queue of delayed requests, we know that the target performance is an underestimate, and the *iprobe* can decrease k to shorten the overall length of the simulation. The *iprobe* reports the current value of k on each change. In practice, we use preliminary tests with a dynamic k to find stable values and then fix k at a stable value for each real test.

5 Platform and Workload

The real test platform used in our study was a Linux (2.6.30) system with two, Intel Xeon 2.80GHz processors, 1 GB main memory, a Western Digital IDE system drive and two external Seagate Cheetah 15K.4 SCSI drives, each with its own Adaptec 39320A Ultra320 SCSI controller. Tests were restricted a single Cheetah drive. The disk is a model ST373454 with 4 recording surfaces and a formatted capacity of 73.4 GBytes. It rotates at 15,000 rpm yielding a rotation time of 4 ms. The disk has 50,864 tracks per recording surface and was formatted at 512 bytes per sector.

To approximate the linear service time at the macroscopic level, we disabled the on-board cache, opened */dev/sda* using the O_DIRECT mode, which forced a bypass of the page buffer cache, and read 100,000 randomly selected pages. The time required to read each page and the distance in sectors from the previously read page were captured. We sorted this data in order of increasing distance and plotted distance versus time. The result was a reasonably linear band of noise approximately 4 ms in width. The data was then smoothed using a filter that replaced each point with the average of the (up to) 1001 points centered at the point in question. The filtered data and the least squares approximation to it are plotted in Figure 1. The linear model is $X_r = 4.25 + 5.25(d_r/D_r)$.

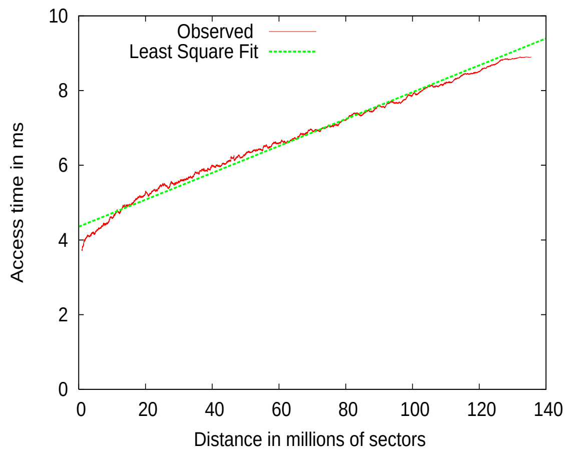

Fig. 1. Stochastic sector to sector costs

The Cheetah manual [22] indicates that 7,077KB is available for caching, which yields 221 512-byte sectors per segment. We assumed that a single span of requested sectors, plus the pre-fetch, would be cached in each segment. Single request spans larger than 157 sectors (221-64) were rare in our workloads, but for those cases we assumed a wrap-around with over-write within the segment.

The KVM-based virtual machine was hosted on an IBM 8853AC1 dual-Xeon blade. It was configured with a 73GB virtual SCSI disk, for which the available emulator was an LSI Logic / Symbios Logic 53c895. The virtual disk image was stored on a NetApp FAS960c and accessed via NFS.

We compared the performance of our cache-aware, table-scheduling (CATS) algorithm with *cfq* and *deadline* on both real and virtual platforms. The CATS *busdriver* delay was set to $7ms$ and the maximum READDELAY to $100ms$.

We used two workloads in this study. Both were similar, at the process level, to that used by Geist, Steele, and Westall [17], which was based on the approach used by Barford and Crovella [19] in building their Scalable URL Reference Generator (SURGE) tool. Each of 50 processes executed an ON/OFF infinite request loop, shown in pseudo-code in Figure 2.

The Pareto(α,k) distribution is a heavy-tailed distribution,

$$F_X(x) = \begin{cases} 1-(k/x)^{\alpha} & x \geq k \\ 0 & elsewhere \end{cases} \tag{1}$$

The discrete Zipf distribution is given by

$$p(i) = k/(i+1), \qquad i = 0, 1, ..., N \tag{2}$$

```
forever{
    generate a file count, n, from Pareto($\alpha_1$,$k_1$);
    repeat(n times){
        select filename using Zipf(N);
        while(file not read){
            read page from file;
            generate t from Pareto($\alpha_2$,$k_2$);
            sleep t milliseconds;
            }
        }
    }
```

Fig. 2. ON/OFF execution by each of 50 concurrent processes

where k is a normalizing factor, specifically, the reciprocal of the $N+1^{st}$ harmonic number. A continuous approximation,

$$F_X(x) = \frac{log(x+1)}{log(N+2)} \qquad 0 \leq x \leq N+1 \tag{3}$$

suffices for our study.

File count parameters were taken directly from the Barford and Crovella study, (α_1, k_1) = (2.43,1.00). The shape parameter of the sleep interval, α_2 = 1.50, was also taken from this study, but we used a different scale parameter, k_2 = 2.0, because our sleep interval was milliseconds per block rather than seconds per file.

Both the virtual SCSI drive and the Cheetah drive were loaded with 1 million files in a two-level directory hierarchy where file sizes were randomly selected from a mixture distribution also suggested by Barford and Crovella. This mixture distribution is lognormal(9.357,1.318) below 133 KB and Pareto(1.1,133K) above, where the lognormal(μ,σ) distribution function is given by:

$$F_Y(y) = \int_0^y e^{-\frac{(log_e t-\mu)^2}{2\sigma^2}} / (t\sigma\sqrt{2\pi})dt \quad y > 0 \tag{4}$$

To induce reasonable fragmentation, we erased a half million files, selected at random, and then added back a half million files with different, randomly selected sizes. Due to the relatively small capacity of these drives, we chose to truncate at 100 MB any files that would have exceeded that size.

For each algorithm, for each test run, we captured the arrival times, service initiation times, and service completion times of 50,000 requests. We captured these time stamps by directly instrumenting the kernel outside of the schedulers, and, during each test run, we stored the time stamps to a static kernel array. The time stamp data was extracted from the kernel array after the test run by using a custom system call.

The two workloads were identical at the process level but decidedly different at the drive level. For the first, each file was opened with mode flag *O_DIRECT*, which, as noted earlier, forced the associated I/O to by-pass the main memory page buffer cache. The second workload differed from the first only in that the *O_DIRECT* flag was not used. The page buffer cache, a standard feature of UNIX-derivative operating systems, is dynamic and can grow to become quite large. For tests described here, 65MB was often observed.

Finally, although the Cheetah drive supports tagged command queueing (TCQ), we disabled it for all tests. We found that, for all schedulers, allowing re-scheduling by the drive hardware decreased performance. We would have thought this to be an anomaly, but we have observed the same result for other SCSI drives on other Linux systems.

6 Results

The results for the first workload, using *O_DIRECT*, are shown in Table 2. We

	real			virtual (k=8)		
algorithm	*cats*	*deadline*	*cfq*	*cats*	*deadline*	*cfq*
mean service (ms)	1.96	2.71	1.39	2.58	3.24	2.36
variance service	8.51	9.76	5.85	9.03	8.23	7.78
mean response (ms)	37.35	59.87	124.70	53.79	78.27	117.13
variance response	6961.50	561.15	839270.49	16641.07	633.28	28651.71
throughput (sectors/ms)	8.19	6.08	2.19	6.15	5.06	3.38

Table 2. Performance on *O_DIRECT* workload.

see that the virtual system uniformly predicted higher mean service, higher mean response, and lower throughput than was found from measurements of the real system. Nevertheless, on all three measures, the predicted performance rank of the three algorithms was correct: CATS performs better than *deadline*, which performs better than the Linux default, *cfq*. Thus algorithm selection could be made solely on the basis of the virtual system predictions.

We also gauged the effectiveness of the shadow cache in predicting real cache hits by placing record markers within the captured time stamp trace of the CATS scheduler on all records that were predicted by the shadow cache to be hits. We examined the service time distribution of a large collection of single-sector requests, independent of any trace, and found a prominent initial spike at 250 microseconds. We then processed the CATS trace and marked any record with service time below 250 microseconds as an actual hit. We found that the shadow cache correctly predicted 97% of the 31,949 actual hits observed.

When I/O is staged through the main memory page buffer cache (the second workload), the results are decidedly different, as shown in Table 3. Again the virtual system uniformly overestimated mean service time and mean response time and underestimated throughput, but again it correctly predicted the performance rank of all three algorithms on all three measures.

	real			virtual (k=8)		
algorithm	*cats*	*deadline*	*cfq*	*cats*	*deadline*	*cfq*
mean service (ms)	6.53	7.41	7.80	7.15	7.60	8.57
variance service	11.13	8.80	17.62	6.22	6.05	10.30
mean response (ms)	114.91	121.87	179.17	189.45	198.33	258.75
variance response	8080.16	3296.87	35349.39	19292.52	6839.66	65796.33
throughput (sectors/ms)	12.00	12.04	9.08	11.44	11.68	8.82

Table 3. Performance on non-*O_DIRECT* workload.

7 Conclusions

We have suggested a method for predicting the performance of disk scheduling algorithms on real machines using only their measured performance on virtual machines. The method uses a dynamically loaded kernel intercept probe (*iprobe*) to adjust low-level virtual device timing to match that of a simple model derived from the real disk device. We used this method to predict the performance of three disk scheduling algorithms, one of which is new. Although the virtual system was seen to uniformly underestimate performance, it correctly predicted the relative performance of the three algorithms as measured on a real system under two workloads.

It it fair to charge that we are simply using a virtual operating system as an elaborate simulator. Nevertheless, this simulator provides almost all the subtle nuances of a real operating system and yet requires almost no programming effort on our part. A low-level model of device service time performance and the drop-in *iprobe* are all that is required.

We are currently working on methods to achieve more accurate absolute predictions. As yet we have not accounted for measurement overhead inherent in our method. We believe that this is one of the causes of the uniformly underestimated performance. Another potential source of error is the model service time ascribed to a cache hit, which we fix at 250 microseconds, even though this only an observed maximum. Unfortunately, reducing this value, X_r, will require an increase in scale factor, k, so that $kX_r - X_V$ remains non-negative, and increasing k increases run-time. We may be faced with a trade-off between accuracy and (simulation) run-time. Nevertheless, we believe that accounting for these factors will result in more accurate predictions.

References

1. Creasy, R.: The origin of the VM/370 time-sharing system. IBM Journal of Research & Development **25** (1981) 483–490
2. Popek, G.J., Goldberg, R.P.: Formal requirements for virtualizable third generation architectures. Commun. ACM **17** (1974) 412–421
3. Robin, J.S., Irvine, C.E.: Analysis of the Intel®Pentium's$^{\mathrm{TM}}$ability to support a secure virtual machine monitor. Proc. of the 9th conf. on USENIX Security Symposium (SSYM'00), Berkeley, CA, USA, USENIX Assoc. (2000) 10–10

4. VMWare, Inc.: Understanding full virtualization, paravirtualization, and hardware assist. http://www.vmware.com/files/pdf/VMware_paravirtualization.pdf (2007)
5. Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauery, R., Pratt, I., Warfield, A.: Xen and the art of virtualization. In: Proc. 19th ACM Symp. on Operating System Principles, Bolton Landing, New York (2003) 164–177
6. Kivity, A., Kamay, Y., Laor, D., Lublin, U., Liguori, A.: *kvm*: the Linux virtual machine monitor. In: Proceedings of the Linux Symposium, Ottawa, Ontario, Canada (2007) 225–230
7. Geist, R., Jones, Z., Westall, J.: Virtualizing high-performance graphics cards for driver design and development. In: Proc. 19^{th} Annual Int. Conf. of the IBM Centers for Advanced Studies (CASCON 2009), Toronto, Ontario, Canada (2009)
8. Geist, R., Jones, Z., Westall, J.: Virtualization of an advanced course in operating systems. In: Proc. 3rd Int. Conf. on the Virtual Computing Initiative (ICVCI3), Raleigh, North Carolina (2009)
9. Jones, Z.H.: A Framework for Virtual Device Driver Development and Virtual Device-Based Performance Modeling. PhD thesis, Clemson University (2010)
10. Coffman, E., Hofri, M.: On the expected performance of scanning disks. SIAM J. on Computing **11** (1982) 60–70
11. Jr., E.G.C., Gilbert, E.N.: Polling and greedy servers on a line. Queueing Syst. **2** (1987) 115–145
12. Geist, R., Daniel, S.: A continuum of disk scheduling algorithms. ACM TOCS **5** (1987) 77–92
13. Worthington, B.L., Ganger, G.R., Patt, Y.N.: Scheduling algorithms for modern disk drives. In: Proceedings of the 1994 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems, Nashville, TN, USA (1994) 241–251
14. Pratt, S., Heger, D.A.: Workload dependent performance evaluation of the Linux 2.6 i/o schedulers. In: Proc. of the Linux Symposium. Volume 2., Ottawa, Ontario (2004) 425–448
15. Geist, R., Ross, R.: Disk scheduling revisited: Can $o(n^2)$ algorithms compete? In: Proc. of the 35^{th} Annual ACM SE Conf., Murfreesboro, Tennessee (1997) 51 – 56
16. Bellman, R.E.: Dynamic Programming. Dover Publications, Incorporated (2003)
17. Geist, R., Steele, J., Westall, J.: Enhancing webserver performance through the use of a drop-in, statically optimal, disk scheduler. In: Proc. of the 31^{st} Annual Int. Conf. of the Computer Measurement Group (CMG 2005), Orlando, Florida (2005) 697 – 706
18. Iyer, S., Druschel, P.: Anticipatory scheduling: a disk scheduling framework to overcome deceptive idleness in synchronous i/o. In: SOSP '01: Proceedings of the eighteenth ACM symposium on Operating systems principles, New York, NY, USA, ACM (2001) 117–130
19. Barford, P., Crovella, M.: Generating representative web workloads for network and server performance evaluation. In: Proc. ACM SIGMETRICS Measurement and Modeling of Computer Systems. (1998) 151–160
20. Douglas Gilbert: sdparm utility 1.03. http://sg.danny.cz/sg/sdparm.html (2008)
21. Qian, J., Meyers, C.R., Wang, A.I.A.: A Linux implementation validation of track-aligned extents and track-aligned raids. In: USENIX Annual Technical Conference, Boston, MA (2008) 261–266
22. Seagate Technology LLC: Product Manual Cheetah 15K.4 SCSI. Pub. no. 100220456, rev. d edn., Scotts Valley, CA (2005)