CycleMeter: Detecting Fraudulent Peers in Internet Cycle Sharing

Zheng Zhang, Y. Charlie Hu, and Samuel P. Midkiff
Purdue University
West Lafayette, IN 47907
{zhang97, ychu, smidkiff}@purdue.edu

Abstract

Internet cycle sharing systems that utilize idle computing resources dramatically increase the available resources for high performance computing. Fraudulent resource providers, however, can subvert these systems. While previous research has investigated protection against resource providers that return bad results, we consider a different fraudulent behavior – *cycle short-changing* – in which the resource provider faithfully executes the submitted job, but using a smaller percentage of the CPU resources than he/she promises. To detect this short-changing, we propose *CycleMeter*, a tool that allows a remotely executing application to accurately monitor the percentage of CPU resources it is utilizing throughout its execution period. CycleMeter employs a microbenchmark to measure the instantaneous CPU utilization of the application, and employs a simple and practical mechanism for embedding the microbenchmark into the application. Our experimental results on three operating systems and uniprocessor and multiprocessor machines show that CycleMeter is portable, incurs a low overhead, and is highly effective in detecting a spectrum of cycle short-changing behavior.

Keywords: benchmarking, monitoring, operating systems, peer-to-Peer

1 Introduction

Cycle sharing over the Internet is becoming increasingly popular as a way to allow otherwise idle cycles to be used. Large numbers of Internet connected computers can join a peer-to-peer network and donate idle CPU cycles to fulfill the computing needs of peer nodes. Nodes can be credited, or be charged, according to their contributions to, or consumption of, resources [Butt et al. 2004; Lo et al. 2004]. With individual hosts becoming more powerful and more idle cycles becoming available, peer-to-peer cycle sharing is becoming an ideal mechanism to utilize idle CPU resources.

Significant technical challenges must be overcome to unleash the potential of the massive computational resources that are going unused:

1. How are resources discovered, and how are providers of these resources compensated (and cheaters punished)?
2. How is the *host* machine, i.e. the machine executing the job, protected from hostile binaries?
3. How does the *submitter* machine, i.e. the machine submitting a job, know its job is being faithfully executed as the job is executed?
4. How does the *submitter* machine know the *host* machine is providing the application with the fraction of the CPU cycles that were promised, so that the submitter can properly compensate the host machine for the cycles?

The first three of these items are the subject of research by the authors of this paper and others [Anderson 2004; Barham et al. 2003; Butt et al. 2004; Butt et al. 2003; Jiang and Xu 2004; Yang et al. 2005b; Yang et al. 2005a]. These systems provide key functionalities of Internet sharing such as resource discovery, host safety, accounting, and remote monitoring.

In this paper, we focus on the fourth problem: how can the submitter detect fraudulent behavior where the resource provider short-changes the submitter by not providing the percentage of the machine's CPUs that were promised to the submitter. For example, the resource provider could promise to provide the submitted job with exclusive use of CPU, but actually schedule the job along with another job, with each only getting half of the CPU, and then demand compensation from each submitter for the exclusive use of the CPU resources. Similarly, the resource provider could promise to run a two-threaded job on a two-processor machine, but actually schedule the two threads on the same processor. If the payment is based on the total number of execution cycles indicated by the elapsed wall clock time, the resource provider can use the OS provided time-sharing scheduling to effectively lower the percentage of cycles that are con-

SC2006 November 2006, Tampa, Florida, USA
0-7695-2700-0/06 $20.00 ©2006 IEEE

tributed to the job, prolong the elapsed time, and thereby receive a higher payment than he/she should.

Estimating the extent to which a CPU is shared (i.e. CPU sharing) appears challenging because an operating system provides the application with the illusion that it exclusively owns the CPU: a normal application process does not know when it is context-switched out by the operating system. A straightforward solution appears to be to check whether the elapsed time of the job is reasonable for the computing platform. But accurately predicting a reasonable elapsed time is difficult for two reasons. First, for many programs, the correlation between the coarse properties of input such as its size do not accurately predict the running time of the job. Second, in a cycle sharing system, the platform may change from run to run, further making it difficult to predict the application running time. Monitoring the incremental progress of parts of the program also suffers from the same problem, as the reasonable elapsed time to execute a certain part of the job is difficult to predict. Another solution is to use the CPU usage reporting facilities provided by the operating system to monitor the actual CPU time for a job. For example, Unix-like operating systems provide the `times()` system call that returns the time the CPU has actually spent in the calling process. However, fraudulent machine owners can easily tamper with the operating system to make `times()` always return a time slightly smaller than the time elapsed since process creation, creating the illusion that the caller process exclusively owns the CPU.

In this paper, we propose a tool that allows an application to estimate the occurrence and extent of CPU sharing when it is executed on a remote host machine. The key idea of our technique is that *a process can infer the occurrence of context-switches by fine-grained time monitoring*. Our proposed tool, CycleMeter, leverages this key idea to estimate the application's CPU utilization (i.e. the percentage of the CPU resources the application is utilizing). CycleMeter has two components: a microbenchmark for measuring instantaneous CPU utilization and a simple mechanism that embeds the microbenchmark into the remotely executing applications.

The microbenchmark repeatedly records the real time spent executing a short sequence of instructions. From this, the CycleMeter tool infers how often context-switches that lead to the execution of another process occur by using a statistical method.

To monitor the extent of CPU sharing suffered by a remotely executing application, the microbenchmark is embedded into the application and executed in the context of the application. For each invocation, the microbenchmark effectively samples the CPU utilization, and the overall CPU utilization throughout the application's execution is calculated by integrating over these samples. Since the cost of each invocation is much lower than the time between consecutive invocations, the overhead of our monitoring scheme is kept low.

CycleMeter is designed to support general applications; it can correctly estimate the degree of CPU sharing in a job mix consisting of both CPU-intensive and I/O-intensive jobs, as well as on multiprocessor machines. CycleMeter is also highly portable; it make no assumptions about the operating system scheduling algorithms.

Our experimental results on three operating systems using a mix of applications show that the microbenchmark is accurate in measuring the instantaneous CPU utilization under various workloads including I/O-intensive applications. We also show that CycleMeter runs with low overhead (3.33%), is effective in detecting a spectrum of fraudulent behaviors, and gives a good estimation of the overall CPU utilization over long execution period.

We summarize our contributions as follows:

- We propose the first technique that addresses cycle short-changing, a fraudulent behavior not considered by previous Internet cycle sharing research. Our technique measures CPU utilization without relying on any knowledge of the hardware or the operating system.
- We present an accurate and portable microbenchmark for measuring instantaneous CPU utilization and a scheme to embed the microbenchmark into applications to effectively detect cycle short-changing with low runtime overhead over long execution period.
- We present experimental results on Linux and FreeBSD which have representative CPU scheduling algorithms as well as on uniprocessor and multiprocessor machines, showing the effectiveness of our proposed CycleMeter tool.

The rest of the paper is organized as follows. Section 2 describes the assumptions about the fraudulent environment that CycleMeter targets. Section 3 gives an overview of CycleMeter. Section 4 presents the design of the CycleMeter microbenchmark. Section 5 describes the strategies to embed CycleMeter into applications. Section 6 presents the experimental results. Section 7 discusses several possible attacks on CycleMeter and how they can be countered. Section 8 discusses the related work. Finally, Section 9 concludes the paper.

2 The Cheating Model

Untrusted systems can be divided into two categories: malicious and rationally fraudulent (e.g. the environment targeted by the Samsara [Cox and Noble 2003] system). Malicious systems are willing to expend significant resources to damage their victims. For example, a malicious system might be willing to execute a program until the final results are to be written to disk, and then terminate it. Our system is assumed to operate in a less hostile, but potentially fraudulent environment.

In particular, we assume that the host machine may not provide all the promised computing cycles to the submitted

job. This is done by time-share scheduling the submitted job with other jobs. Examples of fraudulent behavior include (1) scheduling a low-priority job with the submitted job, (2) scheduling an I/O intensive job with low CPU demand along with the submitted job, and (3) sporadically scheduling short jobs with the submitted job, (4) pretending to be a multiprocessor machine but actually running the multi-threaded submitted job on the uniprocessor machine. To hide its fraudulent behavior in time-sharing the CPU among multiple jobs, we assume the fraudulent resource provider will alter the operating system CPU usage utility system calls to prevent a submitted job from getting correct CPU usage statistics. However, we assume that the host provides an accurate wall clock time facility, as spoofing the wall clock can be easily detected by comparing it with the submitter's wall clock.

Our CycleMeter tool allows an application to uncover CPU sharing during its execution and thereby detects the host machine's cheating behavior. Furthermore, CycleMeter can accurately determine the total CPU usage during the execution of the application, which can be used for cheat-proof accounting and compensation. Our tool makes it very difficult to cheat because the placement of executions of the microbenchmark makes it *appear* to be monitoring almost the entire program execution, but CycleMeter's sampling mechanism *actually* monitors only a small part of the execution. Because CycleMeter appears to cover most of the program, automatic tools that are designed to tamper with CycleMeter must be equally thorough in removing the CycleMeter related code from the application. Moreover, techniques such as those in [Collberg et al. 2004] can be used to further thwart these automatic tools. Program alteration requiring human intervention is costly enough to preclude its use.

3 CycleMeter Overview

Figure 1 gives an overview of CycleMeter. On Unix-like operating systems, CycleMeter is implemented as a library that is linked with the submitter's application and monitors the remote execution of the application. To enable such monitoring, the submitter inserts a CycleMeter initialization call into the application at compile time. The initialization call installs CycleMeter as a `SIGALRM` handler that is periodically triggered by the `SIGALRM` signal from the operating system, and executes the subroutine `microbench_probe()`. During the execution of the remote binary, CycleMeter executes the subroutine `microbench_measure()` to determine the CPU utilization. The details of these subroutines are presented in the next section. At the completion of the submitted job binary on the remote host, CycleMeter presents the CPU utilization samples of the run to the submitter. The submitter can then use these samples to detect the occurrence of any fraudulent behavior, and to determine the total CPU usage by the application. We note that instead of using a single CycleMeter initialization call, an enhanced compiler can use inlined code that is interspersed with, and scattered throughout the `main()`

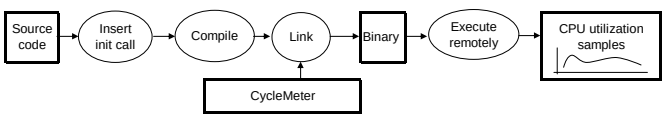


Figure 1: Overview of CycleMeter

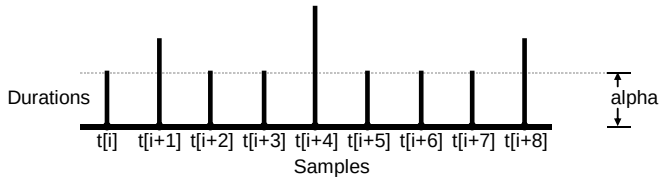


Figure 2: Most of the code fragment invocations without context-switch yield a fixed elapsed time of α

function of regular code. This, along with code obfuscation techniques or the techniques of [Collberg et al. 2004], will make detecting and disabling the initialization code difficult without expensive manual intervention.

4 Microbenchmark

To measure the instantaneous CPU utilization by an application, we designed a microbenchmark that is periodically executed by the application. We present the design of this microbenchmark in this section.

4.1 Design Concept

The microbenchmark measures the CPU utilization over an execution duration T which is typically 1 or 2 seconds, larger than the execution quantum of the time-sharing scheduler. The microbenchmark repeatedly invokes the same fragment of code and monitors the real (wall clock) elapsed time for each invocation. If the code fragment has been executed without being interrupted by any context-switch, the monitored real elapsed time is a fixed value α. However, if a context-switch occurs during the invocation, the monitored real elapsed time increases significantly, as illustrated by $t[i+1], t[i+4]$ and $t[i+8]$ in Figure 2. Given the repeatedly executed code fragment is short, and context-switch rarely occurs during a single execution of such a short fragment, α will manifest itself as the *mode* in the collection of monitored real elapsed times, i.e. the item that appears most frequently. Therefore, the CPU time that would be spent in the microbenchmark with no CPU sharing is effectively α multiplied by the number of times the code fragment is executed. The ratio of this time to the total elapsed (wall clock) time over the microbenchmark execution yields the CPU utilization.

4.2 Implementation Details

In the following, we discuss the implementation details of the microbenchmark.

4.2.1 Two Phases of the Microbenchmark

Our microbenchmark consists of two phases, a *probing phase* and a *measuring phase*. Figure 3 shows the pseudocode of the corresponding major subroutines. In both procedures, the microbenchmark repeatedly measures the elapsed time of each invocation of the code fragment `do_some_computation()` by polling the wall clock at the start and end of the invocation. The purpose of the probing phase is to learn the value of α. The corresponding procedure `microbench_probe()` collects N measurements and stores them for post-processing for inferring α. The purpose of the measuring phase is to sample an execution duration, usually longer than the probing phase, and use the learned α value to estimate CPU utilization. The corresponding procedure `microbench_measure()` differs from `microbench_probe()` in that it performs measurement for a specified time duration T, and instead of storing the measurement data into a buffer, it accumulates them as the sum S. Finally the microbenchmark determines the CPU utilization as

$$CPU\_Utilization = \frac{\alpha \times M}{S}$$

where M is the number of measurements and S is the sum of all measurements obtained in the measuring phase.

Decoupling the probing and the measuring phases conserves memory. This is because a relatively small loop size in the probing phase suffices to infer α, but to get an unbiased CPU utilization estimate a relatively large loop size is needed in the measuring phase. If we decouple the probing and measuring phases, the buffer to store measurement data is only needed in the measuring phase and hence is kept small. Another advantage of decoupling is that for multiple microbenchmark invocations on the same platform, only a single run of the probing phase is required.

4.2.2 Choice of Parameters

The instruction sequence denoted by subroutine `do_some_computation()` is a loop with K iterations. The parameter K together with N (outer loop size in the probing phase) and T (the time interval for the measuring phase) affect the accuracy and overhead of the microbenchmark.

1) *Choice of K*

Ideally, the parameter K should be chosen to make the probability that the code fragment execution is interrupted low, so that the α-being-mode assumption holds. We conservatively estimate the upper bound of K as follows. On most operating systems the minimum time slice for which a process can run without being context-switched is 10 milliseconds. Conservatively, we assume the execution of the microbenchmark process is interrupted every 10 milliseconds, and hence the probability that no interruption occurs during time interval α is $1-\alpha/(10\text{msec})$. The sufficient condition that α manifests itself as the mode is that the above probability is larger than $1/2$. This inequality yields $\alpha < 5$ msec, i.e. 5 millisecond is the upper bound. On the other hand, K should not be too small to affect the measurement coverage. This is because as K decreases, the outer loop size in `microbench_measure()` increases in order to meet the requirement of the measurement duration T, the total looping overhead increases, and hence the part of the measurement duration T that is covered by the measurement decreases.

```
microbench_probe() {
  //learn the process virtual time
  for (i = 0; i < N; i ++) {
    start = gettime();
    do_some_computation(K);
    stop = gettime();
    t[i] = stop - start;  //record
  }
}

microbench_measure() {
  //probe for CPU sharing
  S = 0; M = 0;
  while (S < T)
  {
    start = gettime();
    do_some_computation(K);
    stop = gettime();
    S += start - stop;  //accumulate
    M ++;
  }
}
```

Figure 3: Pseudocode of the two microbenchmark subroutines. `gettime()` returns the wall clock time.

With the above two requirements, the microbenchmark adaptively selects K from a set of candidates $\{100, 1000, 10000, 100000, ..\}$. It starts from the smallest one (100) and searches for the largest K with which the invocation of the learning phase yields a corresponding α less than 5 msec (i.e. meets the first requirement). Note that it is not possible to select a pathological K value, which breaks the α-being-mode assumption and compromises the α estimation. We can prove it by contradiction as follows. If such a K is selected, the inferred α should be less than 5 msec by the adaptive selection itself. However, according to our conservative K upper bound estimation, in order to break the α-being-mode assumption, α should at least be greater than 5 msec. With such a K, the inferred α should be an overestimation because context-switches are counted in. So it should be greater than 5 msec as well, which contradicts our initial assumption.

2) *Choice of N*

The parameter N determines the number of code fragment invocations in the probing phase. The choice of N is not critical for two reasons. First, whether the mode assumption holds or not is independent of the particular choice of N. The α is always the mode unless N is too small, say 2. Second, the choice of N does determine the cost of the probing phase, but since the probing phase is executed only once for multiple microbenchmark executions, the choice of N does not affect the overall overhead much. In our implementation of the microbenchmark, we choose to set the parameter N 100.

3) *Choice of T*

The parameter T determines the duration of the measuring phase and thereby the measurement overhead. The larger the value of T is, the more accurate the measurement is, but also the more overhead the microbenchmark incurs. We leave it as a tunable parameter to be adjusted according to the requirement of the CPU utilization monitoring.

4.2.3 Choice of Wall Clock

High resolution wall clock improves the accuracy of the microbenchmark measurement. Some architectures have a cycle counter such as the `rdtsc` instruction on Pentiums. Our microbenchmark uses the hardware cycle counter if it is available, otherwise, it resorts to the generic `gettimeofday()` system call. As discussed in Section 2, we assume `gettimeofday()` is not spoofed.

4.2.4 Mode Calculation

There is a subtlety in calculating the mode. The well-known method to calculate mode is to use a histogram. But the problem with using a histogram is that it may distort the true distribution with inappropriate selection of the width and the boundaries of the bins causing the mode to not be accurately reflected. However, it is difficult to choose the appropriate bin width and boundaries before the distribution is known. Kernel density estimation [Parzen 1962], a method well-known in statistics, can overcome this problem. To use it, we first define a kernel function $K(t)$ with the property

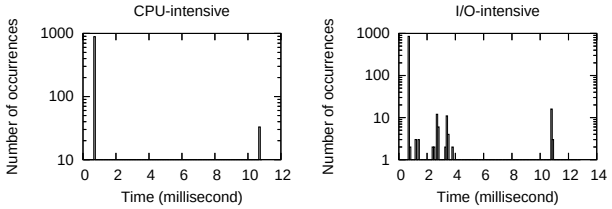
$$\int_{-\infty}^{+\infty} K(t)dt = 1$$

Then the density at point x of a sample set is defined as

$$\frac{1}{n}\sum_{i=1}^{n}\frac{1}{h}K(\frac{x-x_i}{h})$$

where h is the kernel width, n is the number of samples within h of x, and x_i is the ith such sample. This density is smooth compared to that of histogram, in which the density is defined as the number of samples that fall into the corresponding bin. The kernel function we use is

$$K(t) = \begin{cases} 1-|t| & \text{if } |t|<1 \\ 0 & \text{otherwise} \end{cases}$$



(a) (b)

Figure 4: Histogram of the measured elapsed time samples. (a) The microbenchmark shares CPU with a CPU-intensive application, (b) The microbenchmark shares CPU with an I/O-intensive application. (The y axis has log scale)

This function has the desirable property that it gives greater weights to closer samples and is fast to compute. We choose kernel width h large enough to cover all samples. To calculate the mode, we first calculate the density at each sample and then choose the sample with the maximum density as the mode.

4.3 I/O-intensive Applications

The microbenchmark can correctly estimate the CPU utilization when sharing the CPU with I/O-intensive applications. At first glance, CPU utilization in the presence of I/O-intensive applications seems difficult to measure. The intuition is that if the I/O operations happen in a burst, the I/O-intensive application tends to interrupt the execution of the microbenchmark more frequently and thus the α may not manifest itself as the mode. However, this is not true because the measured elapsed time samples resulting from the interruption are scattered (i.e. are not strongly clustered) and thus are not likely to overwhelm α as the mode. Figure 4(a) and (b) show the histogram of the measured elapsed time samples when the microbenchmark shares the CPU with a CPU-intensive application and an I/O-intensive application respectively. In the CPU-intensive case, once the microbenchmark is context-switched out, the resulting elapsed time is always around 11 millisecond, where a cluster is formed. In the I/O-intensive case, although there are more non-α samples, they are scattered and thus do not reduce the strength of the mode. In both cases, the α is approximately 0.7 millisecond and is manifested as the mode, and the microbenchmark is able to work correctly.

4.4 Multiprocessor and Multi-threaded Applications

As can be seen from its design, the microbenchmark functions correctly irrespective of whether the applications that share the CPU are single-threaded or multi-threaded. If the monitored application itself is multi-threaded, the microbenchmark will be invoked by each thread. The microbenchmark measure the CPU utilization by each thread,

and the sum of the measured CPU utilizations over all threads is reported as the CPU utilization by the application process. On a multiprocessor machine, the CPU utilization reported by the microbenchmark should be divided by the number of CPUs. By calculating the CPU utilization in the above way, we can catch the fraudulent resource provider who schedules a multi-threaded application on a smaller number of CPUs than he/she promised. For example, if both threads of a two-threaded application are scheduled on the same CPU on a two-processor machine, the CPU utilization by each thread will be 50%, and hence the CPU utilization is reported as $(50\% + 50\%)/2 = 50\%$, indicating the application receives only half of the computing cycles. On the other hand, if scheduled on two-processor machine, both threads will measure the CPU utilization as 100%, and hence the CPU utilization by the application is $(100\% + 100\%)/2 = 100\%$.

4.5 Hardware and OS Artifacts

When the microbenchmark is executed, other activities besides being context-switched out to a competing process, e.g. cache misses, page faults, and the execution of kernel daemons, may stall the application's execution. These hardware and OS artifacts do not affect the accuracy of the microbenchmark for the following reasons. First, cache misses will not affect the determination of α. This is because compared to the value of α, the delay caused by a cache miss is negligibly small. Second, page faults do not compromise the accuracy of the microbenchmark, because the memory footprint of the microbenchmark is small and hence page faults are rare when the microbenchmark is executing. Finally, the fraction of CPU cycles spent in the execution of kernel daemons such as `bdflush` are not accounted for in the CPU utilization measured by CycleMeter. This is reasonable because first, the fraction is small, and second, multiple processes share the CPU and jointly contribute to the OS level activities, making it difficult to charge these activities to the monitored process.

5 Embedding Microbenchmarks

To monitor the CPU utilization of a remotely executing application over its long execution period, the microbenchmark is embedded into the application process and periodically invoked by the application to take snapshots of the instantaneous CPU utilization. The collection of snapshots serves two purposes. First, it provides information to detect fraudulent behavior. The submitter can infer cheating behavior if some of the CPU utilization snapshots deviate from the expected value. Second, the snapshots can be used to calculate the accumulative CPU usage of the monitored application; this can be used, for example, for cheat-proof resource accounting and compensation.

5.1 Design Options

To monitor CPU utilization, the microbenchmark should be embedded into the application and executed in the context of the application. An alternative is to suspend the application process and create a separate process to execute the microbenchmark. However, some schedulers such as that of FreeBSD favor young processes over old processes, which make this problematic. Since the microbenchmark process and the application process are of different ages and are therefore treated differently, the CPU utilization seen by the microbenchmark will not reflect the utilization seen by the application process. Forking a process from the application process to execute the microbenchmark can avoid the aging problem. However, on some operating systems such as Linux, upon forking, the child and the parent processes split the quantum of the original process, and hence forking may not be as reliable as directly executing the microbenchmark in the context of the application.

There are two possible strategies to decide when to invoke the microbenchmark in an application process. One strategy is interval-based, which uses the timer mechanism to periodically interrupt the execution of the application and invoke the microbenchmark. The other is code-based, which strategically inserts the microbenchmark into the code of the application, for example, at subroutine boundaries. The microbenchmark is executed upon entering the subroutines. The advantage of the latter strategy is that the fraudulent resource provider can not infer or predict the occurrences of microbenchmark invocations, while for the former strategy, the fraudulent resource provider can potentially detect when the microbenchmark is to be invoked because the arrival of a timer interrupt is the signature of a microbenchmark invocation. In our implementation, we choose the former strategy, as it is simple, and the timing and the overhead of the microbenchmark are more controllable. The time interval between two consecutive invocations is fixed throughout the monitoring.

5.2 Implementation of Embedding Code

We have built the CycleMeter tool to facilitate embedding the microbenchmark. As discussed in Section 3, CycleMeter is implemented as a library on Unix-like operating systems. The microbenchmark can be embedded into the application with minimum modification to the source code. CycleMeter appears to the monitored application as the `SIGALRM` signal handler. Every time the `SIGALRM` signal arrives, the signal handler invokes the microbenchmark to measure the instantaneous CPU utilization and then schedules another `SIGALRM` signal for the next microbenchmark invocation. . In multi-threaded applications, each thread schedules its own `SIGALRM` signals and handles these signals independently of other threads.

Application	Input	Type
Simplescalar	SPEC 2000 181.mcf	CPU-intensive
Octave	Matrix multiplication	CPU-intensive
Gzip	A 600 MB file	CPU- and I/O-intensive
Gcc	Octave source code	CPU- and I/O-intensive
Pbzip2	A 600 MB file	CPU-intensive

Table 1: Applications used in the experiments

6 Experimental Results

In this section, we present the experimental results under various workloads on three operating systems: Linux 2.4 and FreeBSD 4.6 on uniprocessor and multiprocessor machines, and Windows XP Professional on a uniprocessor machine. We first discuss the experiments and results run on the Linux and FreeBSD machines, and in Section 6.5 discuss the experiments and results run on the Windows machine.

6.1 Applications, Workload and Machines

Applications We selected five applications for our evaluation that we composed as the workload to run on the remote host. They are shown in Table 1. *Simplescalar* [SimpleScalar LLC] is a cycle-level processor simulator widely used in computer architecture studies, and is computation-intensive. *Octave* [Eaton] is a high-level language intended for numerical computations. In the experiments, *Octave* is used for computing matrix multiplication and is also CPU-intensive. *Gzip* is a widely-used general purpose data compressor, and in the experiments, it is used to compress a 600 MB file. To compress a file, *Gzip* repeatedly reads file blocks from the disk, compresses them in the memory and writes them back to the disk. Hence, it is both CPU- and I/O- intensive. *Gcc* compiles the *Octave* source code, and is also both CPU- and I/O-intensive. Although both *Gzip* and *Gcc* are CPU- and I/O-intensive, they differ in their I/O access patterns. The I/O and computation are steadily interleaved in fine-grained fashion in *Gzip*, but alternately occur with bursts in *Gcc*. *Pbzip2* [Gilchrist] is a parallel version of data compression software *bzip2*. It is a multi-threaded application implemented with Pthreads library. In our experiments, *Pbzip2* is used to compress a 600 MB file with multiple concurrent threads. *Pbzip2* works under the highest compression ratio option, and computation dominates the whole execution.

Workload We embedded the microbenchmark into *Simplescalar* and *Pbzip2*, and used them as submitted jobs. Different combinations of applications were selected to run together with the submitted job under various priority settings. Table 2 shows the various workloads used in the experiments. The workloads include both a variety of application computation-I/O types and a spectrum of fraudulent cycle-short-changing behavior. In *SS_1*, *SS_2* and *SS_3*, the submitted job and the competing job, both being *Simplescalar*, ran with different nice values. The competing job ran with Unix command `nice -n <nice_value>`. A higher nice value means lower priority and therefore less CPU demand. These workloads are intended for evaluating the microbenchmark when sharing the CPU with CPU-intensive jobs under various priority settings. *SS_2* and *SS_3* also exemplify low-priority job cheating, in which another job is scheduled with the submitted job but runs with a low priority. *Oct_1* directly runs *Octave*, and *Oct_2* iteratively runs *Octave* for 40 seconds and suspends it for a random period ranging from 40 to 300 seconds. *Oct_2* is an example of a sophisticated sporadic scheduling cheating, in which short jobs are scheduled sporadically. *Gzip* and *Gcc* are a direct execution of the corresponding applications. They are intended to evaluate the microbenchmark when the competing jobs have I/O operations, and are examples of I/O intensive job cheating in which an I/O-intensive job with less CPU demand is scheduled with the submitted job. *Pbz* runs *Pbzip2* with two threads on uniprocessor machines and runs with four threads on a four-processor SMP machine. We assume in the former case that the uniprocessor host lies to the submitter that the it has two processors, and thereby exemplifies fake multiprocessor cheating. In the latter case, the multiprocessor host faithfully dedicates its four processors to the multi-threaded application. Finally, three mixture workloads *Mix_1*, *Mix_2* and *Mix_3* are composed from the five applications.

Workload	Applications	Nice
SS_1	*SS* + *SS*	0
SS_2	*SS* + *SS*	10
SS_3	*SS* + *SS*	15
Oct_1	*SS* + *Octave*	0
Oct_2	*SS* + *Octave* (sporadically)	0
Gzip	*SS* + *Gzip*	0
Gcc	*SS* + *Gcc*	0
Pbz	*Pbzip2*	
Mix_1	*SS* + *SS* + *Octave*	0
Mix_2	*SS* + *SS* + *Gzip*	0
Mix_3	*SS* + *SS* + *Gzip* + *Gcc*	0

Table 2: Workloads used in the experiments. (The first item in the "Applications" column is the submitted job embedded with CycleMeter and the other items are the competing jobs that share the CPU with the submitted job. "Nice" column lists the nice of the competing jobs. *SS*=*Simplescalar*)

Machines Two of the three experimental machines have identical hardware, an Intel(R) Pentium 4 2.0 GHz processor with 2 GB RAM size, with one running Linux 2.4 and the other running FreeBSD 4.6. The third machine is a 4-processor SMP with 5.4 GB shared RAM running Linux 2.4. Each CPU is an Intel(R) Xeon MT 1.5 GHz multiprocessor. On all experimental machines, the Pthreads library is implemented using kernel threads.

6.2 Operating System Schedulers

Before presenting the experimental results, we briefly review the schedulers of the two operating systems used. Both Linux and FreeBSD schedule processes based on dynamic priorities which vary over time. However, the two are slightly different in the way they assign time slices. The Linux scheduler discriminates against a low priority process by giving it a time slice less than the default. The FreeBSD's multiple queue feedback scheduler discriminates against a low priority process by increasing the time slice of the other processes. Therefore, the time slice on FreeBSD is generally longer than on Linux. Moreover, on Linux the time slice of a process is determined by the static priority and thus does not change with the dynamic priority, while on FreeBSD the time slice changes as the dynamic priority ages and is rejuvenated. The implication of longer and dynamic time slices on FreeBSD is that the process's instantaneous CPU utilization at the time scale of 1 or 2 seconds changes over time, while it is more stable on Linux.

6.3 Evaluation of the Microbenchmark

We first evaluated the accuracy of the microbenchmark measurement, i.e. how accurately the embedded microbenchmark measures the instantaneous CPU utilization for the (wall clock) time interval during which it is executed, with respect to the actual CPU utilization for the same time interval.

We ran the workloads listed in Table 2 on all the three experimental machines. The microbenchmark was embedded in the submitted applications *Simplescalar* and *Pbzip2*, and was invoked 20 times with fixed time interval of 10 seconds. The settings of the microbenchmark are as follows: the instruction sequence loop size (K) is adaptively selected and ends up being 100000 on all the three machines, and the number of measurements for the learning phase (N) is 100. The wall clock is chosen from both the Pentium cycle counter and generic `gettimeofday()`, and the duration of the execution snapshot to be measured (T) is 0.5, 1 or 2 seconds. We also obtained the actual CPU utilization of the execution time interval by polling system call `times()`. System call `times()` returns the real time elapsed, the user time (the CPU time spent by the process in the user mode), and the kernel time (the time spent by the process in kernel mode) since the process creation. The sum of the latter two yields the time when process utilizes the CPU. We polled `times()` both at the start and the end of the microbenchmark execution, took the difference to get the incremental change, and then calculated actual CPU utilization from it.

Figure 5 shows the relative error of the microbenchmark instantaneous CPU utilization measurement across various platforms with various microbenchmark parameter settings. The relative error is obtained by comparing each CPU utilization measurement against the corresponding actual value, and calculating the average. Overall, the average measurement error is small, with most of the errors less than 3% and the maximum error less than 6%. We make the following comparison observations.

CPU-intensive vs. I/O-intensive: Workloads *Gcc*, *Gzip*, *Mix_2* and *Mix_3* have a large number of I/O operations. No significant difference between them and the other CPU-intensive workload can be observed on FreeBSD (Figure 5(a) and (d)). However, measurement errors for these workloads, although small, are larger on Linux (Figure 5(b) and (e)) than on FreeBSD. Although we do not precisely understand the cause, we suspect that this is because on Linux `times()` is less accurate than on FreeBSD. On Linux, the scheduler updates the current process CPU statistics only on clock ticks instead of on every occurrence of a context switch. For example, on our experimental machine running Linux 2.4, each clock tick is 10 millisecond. If between two consecutive clock ticks, process A is first scheduled, and then preempted by process B due to I/O, then on the second clock tick, the operating system updates the CPU statistics only for the current process B and thus counts the entire 10 millisecond as the execution on process B.

Linux vs. FreeBSD: No significant difference between Linux and FreeBSD can be observed except for the I/O-intensive workload discussed in the previous paragraph.

Long vs. short measurement duration (T): Larger T generally results in more accuracy because more measurement samples make the measurement more statistically robust, as shown in all sub-figures in Figure 5.

No sharing vs. with sharing: All the workloads on the uniprocessor machines (Figure 5(a) (b) (d) and (e)) have CPU sharing while none of them has this on the multiprocessor machine (Figure 5(c) and (f)). Not surprisingly, the measurement is more accurate without CPU sharing than with CPU sharing.

Cycle counter vs. `gettimeofday()`: No significant difference between cycle counter (Figure 5(a) (b) and (c)) and `gettimeofday()` (Figure 5(b) (e) and (f)) can be observed. This is because the microbenchmark does not require nanosecond-level resolution.

Single-threaded vs. multi-threaded and uniprocessor vs. multiprocessor: The two-threaded application *Pbzip2* in the workload *Pbz* on uniprocessor machines should have CPU utilization of 50% (because the uniprocessor machines claims to have two processors), and the CPU utilization is accurately measured by the microbenchmark. On the multiprocessor machine, the microbenchmark also accurately measures the CPU utilization in this workload as 100%. In the workloads consisting of single-threaded applications on the multiprocessor machine, the submitted job should have a CPU utilization of 100%, and the utilization is also accurately measured by the microbenchmark.

In summary, the microbenchmark is accurate in measuring the instantaneous CPU utilization in various application mixes, for various execution durations, both under Linux and FreeBSD, both with CPU sharing and without CPU sharing,

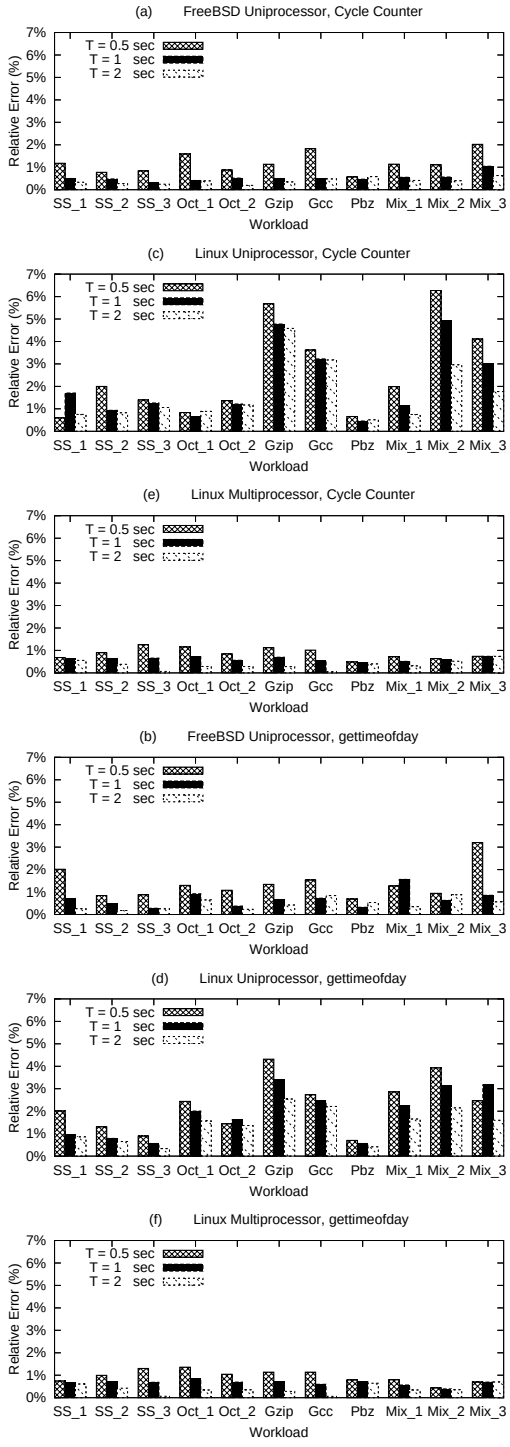


Figure 5: The relative errors of the microbenchmark instantaneous CPU utilization measurements on the three experimental machines with various microbenchmark parameter settings. The x-axis is the workload defined in Table 2.

and both on uniprocessor and multiprocessor machines. Additionally, the accuracy of the microbenchmark does not require very high resolution of the wall clock time.

6.4 Evaluation of CPU Utilization Monitoring

Next we evaluated the CPU utilization monitoring using the embedded microbenchmark over long execution periods. Our evaluation answered two questions: how effectively can our monitoring scheme detect cheating, and how accurate is it in estimating the total consumed cycles?

We evaluated the monitoring scheme under two configurations: (1) sampling with an interval of 30 seconds where the duration of each sampling is $T = 1$ second; and (2) sampling with intervals of 60 seconds where the duration of each sampling is $T = 2$ seconds. The expected runtime overhead of both configurations is as low as $1/30 = 3.33\%$, which we confirmed by measuring one of the workloads. Although both configurations have the same runtime overhead, they represent different strategies. Relatively speaking, configuration 1 has a high sampling rate but short sampling duration, and configuration 2 has a low sampling rate but long sampling duration. Configuration 1 tends to be more accurate in capturing the changes of CPU utilization, and hence better targets at workloads with frequent changes in CPU demand. For workloads with stable CPU demand, samples measured at long durations are more accurate, and hence these workloads tend to be better targeted by configuration 2. Other settings of the embedded microbenchmark are the same as in the previous experiment: the instruction sequence loop size is self-tuned; the number of measurements for the learning phase is $N = 100$; and the cycle counter is used as the wall clock.

Effectiveness of cheating detection Figures 6(a), (b), (c) and (d) show the measured CPU utilization curves in workloads *SS_2*, *Oct_2*, *Gcc* and *Pbz* respectively on the Linux uniprocessor machine under the configuration $T = 1$ second. These workloads exemplify low-priority job cheating, sporadic scheduling cheating, I/O-intensive job cheating, and fake multiprocessor cheating respectively. Due to space limitations, only representative scenarios are shown. The curve is fit to the measured CPU utilization samples by linear interpolation. For better visual presentation the actual instantaneous CPU utilization was continuously (every 2 seconds) reported by the `top` utility running along with the workloads, and these data are now plotted together with the microbenchmark measurements. For the case of *SS_2* and *Pbz*, the microbenchmark curve almost overlaps with the `top` curve. For *Oct_2* and *Gcc*, two curves are close but do not overlap. This is because sampling can not completely capture the changes of CPU utilization. In all four scenarios, the occurrence of gaps between the measured CPU utilization from the expected (i.e. promised by the resource

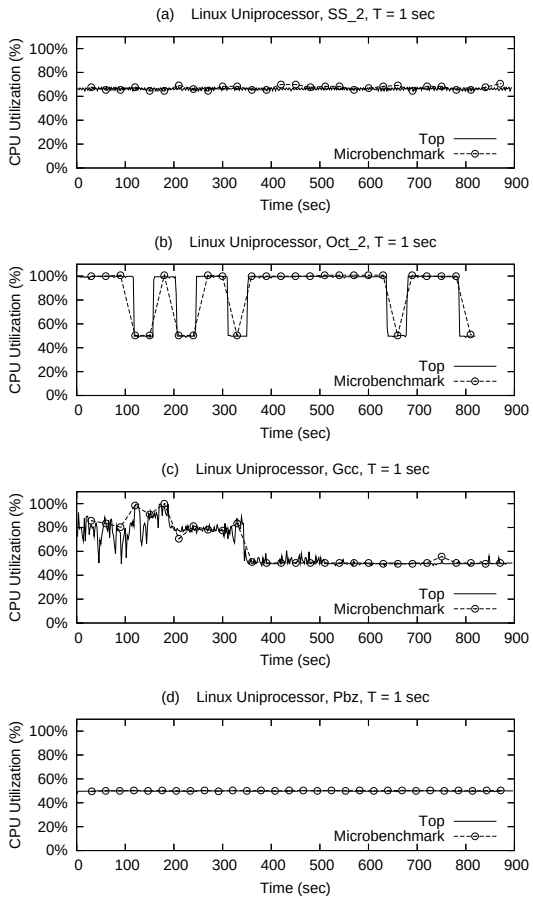


Figure 6: The microbenchmark CPU utilization measurement of four representative workloads on the Linux uniprocessor machine, $T = 1$ second.

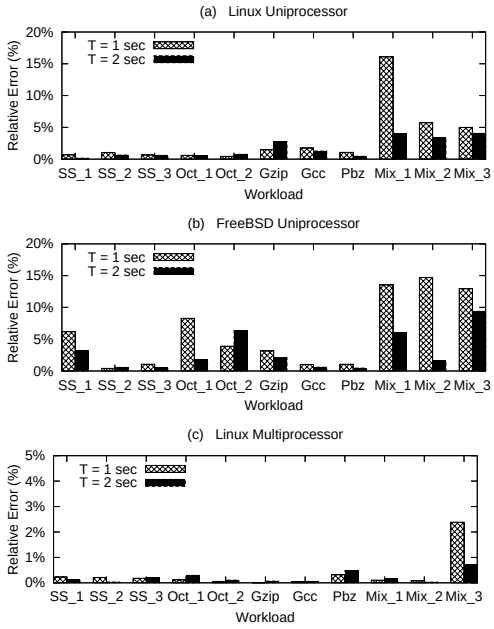


Figure 7: The relative errors of overall CPU utilization estimation on the three experimental machines. The x-axis is the workload defined in Table 2.

provider) CPU utilization of 100% can be easily observed and the occurrence of cheating can be inferred. The same observation can be made for other workloads on both Linux and FreeBSD, and also under the configuration $T = 2$. This shows that CycleMeter is effective in detecting a spectrum of cheating behavior.

Accuracy of overall CPU utilization estimation The overall CPU utilization estimate from the instantaneous utilization measurements can be used for cheat-proof accounting and compensation. Figure 7 summarizes the relative error of the overall CPU utilization estimate compared to the actual utilization given by the `times()` system call on the three experimental machines. The overall CPU utilization is estimated as the integration of the microbenchmark measurement curve over the entire the execution period divided by the length of the period.

Our first observation is that the measurement error is generally small on Linux (Figure 7(a)). The exception is the workload *Mix_1*. The detailed measurement is shown in Figure 8. The error is large because the instantaneous CPU utilization measurement over any 1 second can represent the overall utilization in this workload. In this workload, each of the 3 jobs is executed for a quantum of 100 millisecond in a round-robin fashion. The measurement always starts when the microbenchmark gets the CPU. Within a duration of 1 second, 4 quanta out of a total of 10 quanta are assigned to the microbenchmark and thereby observed as the instantaneous CPU utilization. This explains why most of the points in Figure 8(a) are around 40%. This effect is mitigated by using larger value of T, as is shown by the case $T = 2$ seconds in Figure 8(b).

Our second observation is that compared to the case of Linux, most workloads exhibit a larger error on FreeBSD. This is because on Linux the CPU utilization is quite stable over time, while on FreeBSD the CPU utilization fluctuates even if the applications do not have changing CPU demand. As discussed in Section 6.2, the FreeBSD's multiple queue feedback scheduler assigns changing time slices to processes. This irregularity of scheduling makes the CPU utilization different from time to time. However, the error on FreeBSD is reduced in the configuration where $T = 2$ seconds, because a longer sampling duration makes the measurement less affected by the changing time slice.

In summary, the embedded microbenchmark is effective in detecting the fraudulent behavior, and gives a good estimation of the overall CPU utilization over the long execution period.

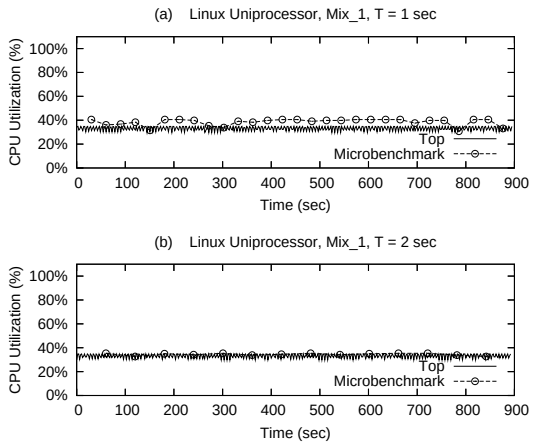


Figure 8: The microbenchmark CPU utilization measurement of workload *Mix_1* on the Linux uniprocessor machine. (a) $T = 1$ second, (b) $T = 2$ seconds

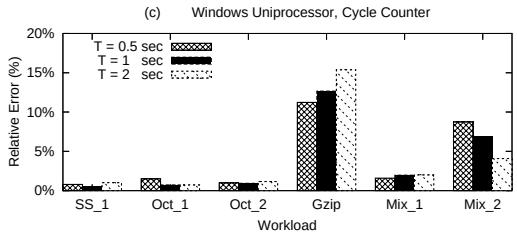


Figure 9: The relative errors of the microbenchmark instantaneous CPU utilization measurement on Windows with various microbenchmark parameter settings. The x-axis is the workload defined in Table 2.

6.5 CycleMeter on Windows

In previous sections, we have shown the experimental results on two Unix-like operating systems, Linux and FreeBSD. We believe Unix-like operating systems are widely-used in scientific computing. However, as a proof of concept, we also evaluated CycleMeter on Windows. The machine that we used has an Intel(R) Celeron 2.66 GHz processor with 256MB RAM, running Microsoft Windows XP Professional. Due to the availability of benchmark application on Windows platform, 6 of the total 11 workloads are used. We used the same settings as in Section 6.3. Figure 9 shows the relative error of the microbenchmark instantaneous CPU utilization measurement with various microbenchmark parameter T settings. The results are comparable to those on Linux and FreeBSD.

7 SIGALRM-based Attacks

CycleMeter relies on the `SIGALRM` signal provided by the operating system to periodically trigger the microbenchmark. At runtime, the arrival of the signal becomes the signature of a microbenchmark invocation, apparently providing a chance for the fraudulent operating system to infer or predict the occurrences of microbenchmark invocation, and making possible attacks to defeat CycleMeter. In this section, we describe several possible attacks against CycleMeter that exploit this signal mechanism and discuss how to counter them.

The first attack is that the fraudulent resource provider completely disables or selectively drops the signals delivered to the submitted application. This will result in disabling or reducing the frequency of the monitoring process. This attack can be countered as follows. Once the execution is complete, the submitter determines the total time for the execution by his/her wall-clock, and then determines the expected number of microbenchmark invocations by dividing the elapsed time by the time interval between invocations. This number is then checked against the actual number of times the microbenchmark was invoked as reported by CycleMeter. If the resource provider has significantly reduced the frequency of the microbenchmark invocations, a significant difference can be observed between the expected and the actual number of invocations. Hence, this attack can be countered.

The second attack is that the fraudulent resource provider behaves honestly only when the microbenchmark is being executed, and is fraudulent everywhere else, i.e. the provider gives the submitted application the exclusive use of the CPU only in the periods when the microbenchmark is being executed. The fraudulent resource provider can determine this period of time as the next couple of seconds after a signal arrival. The third attack is that the fraudulent resource provider follows the code path from the `SIGALRM` signal handler and corrupts or modifies the CycleMeter's data structures used to store timing information. These two attacks can be countered by using the code-based embedding which was discussed in Section 5.1. That is, instead of being triggered by the operating system's signals, the microbenchmark is embedded inline with the submitted binary by a compiler and is executed as the submitted job is making progress. Compiler-inserted counter-based sampling [Arnold and Ryder 2001] can be used to further reduce the overhead. This technique has been used in runtime instrumentation and shown to incur low overhead. We will explore the code-based embedding in our future work.

It is important to note that these attacks require complicated modifications to the operating system kernel or writing a complicated piece of fraudulent software that hosts the execution of the submitted jobs. The implementation is time-consuming and potentially results in a less stable computing environment, which probably precludes the use of these attacks.

8 Related Work

Most work on cheating detection in Internet cycle sharing has focused on detecting resource providers that incorrectly

execute the submitted jobs. We review this work and related work on CPU sharing estimation.

Detecting Incorrect Execution Because of the untrusted nature of the P2P environment, ensuring fairness in charging for the service and verifying the execution results are essential problems in P2P cycle sharing. On one hand, the client may cheat by not paying the computing server for its work; on the other hand, the server can cheat the client by charging for non-existent or false computation. While the former can be discouraged by maintaining a reputation system in the P2P network [Andrade et al. 2004], the latter is much harder to detect and defeat. Existing schemes are based on code encryption (in special cases) [Sander and Tschudin 1998b], trusted hardware [Yee 1994], a combination of hardware and encryption [Loureiro et al. 2002], dummy objects [Meadows 1997], or a quiz-scheme [Lo et al. 2004]. Recent work has addressed the problem of monitoring the progress of submitted jobs and verifying the correctness of remote execution. Butt et al. [Butt et al. 2004] propose a progress monitoring scheme for Java programs based on finite state machines. Yang et al. [Yang et al. 2005b] propose a monitoring scheme for Java using a location-beacon-based finite state machine and partial replay of the computation to support monitoring for progress and correctness. Fei et al. [Fei et al. 2006] propose a monitoring scheme that exploits the design of a software distributed shared memory system to replicate the computation performed at one of the host nodes to ensure the faithful execution of submitted program.

Detecting CPU Sharing To the best of our knowledge, CycleMeter is the first tool that detects CPU sharing, a stealthy way of cheating in Internet cycle sharing environments. CycleMeter shares with the Hourglass [Regehr 2002] tool the use of fine-grained time monitoring to infer the occurrences of context switches. However, it differs from Hourglass in several significant ways. First, Hourglass is a tool for analyzing, measuring and debugging real time applications. CycleMeter, on the other hand, is embedded into the scientific applications and used to monitor CPU utilization of remote execution in Internet cycle sharing systems. Second, Hourglass uses a predetermined threshold to detect the context- switches, and the threshold must be properly selected with knowledge of the operating system and the hardware. In contrast, CycleMeter relies on a statistical method to infer the occurrence of context-switches and does not rely on any threshold. Third, CycleMeter is shown to be applicable to multi-threaded programs, sharing with I/O-intensive applications, and on multiprocessor machines.

9 Conclusions

This paper has described the design and implementation of CycleMeter, a tool that allows the utilization of one or more CPUs by an application to be monitored, even in a fraudulent environment where the system call `times()` may have been tampered with. Moreover, the tool works on both uni- and multi-processor machines. The tool prevents a provider of CPU cycles from promising a job submitter more cycles than the provider is capable of delivering. The resulting information can be used to determine payment to the provider of the CPU, or to validate that providers of CPU cycles have kept their side of an agreement.

Acknowledgment

We thank Ali Butt for useful comments on this paper. This work was supported in part by NSF CAREER award grant ACI-0238379 and NSF grants CCR-0313026 and CCR-0313033.

References

AMZA, C., COX, A., DWARKADAS, S., KELEHER, P., LU, H., RAJAMONY, R., YU, W., AND ZWAENEPOEL, W. 1996. TreadMarks: Shared memory computing on networks of workstations. *IEEE Computer 29*, 2 (Feb.), 18–28.

ANDERSON, D. P. 2004. BOINC: A system for public-resource computing and storage. In *Proc. 5th IEEE/ACM International Workshop on Grid Computing*.

ANDRADE, N., BRASILEIRO, F., CIME, W., AND MOWBRAY, M. 2004. Discouraging free riding in a peer-to-peer CPU-sharing grid. In *Proceedings of the 13th IEEE International Symposium on High performance Distributed Computing (HPDC)*.

ARNOLD, M., AND RYDER, B. G. 2001. A framework for reducing the cost of instrumented code. In *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation (PLDI)*, ACM.

BAILEY, D., BARTON, J., LASINSKI, T., AND SIMON, H. 1991. The NAS parallel benchmarks. Tech. Rep. TR RNR-91-002, NASA Ames, Aug.

BARHAM, P., DRAGOVIC, B., FRASER, K., HAND, S., HARRIS, T., HO, A., NEUGEBAUER, R., PRATT, I., AND WARFIELD, A. 2003. Xen and the art of virtualization. In *Proc. 19th ACM Symposium on Operating Systems Principles (SOSP)*.

BIANCHINI, R., KONTOTHANASSIS, L. I., PINTO, R., DE MARIA, M., ABUD, M., AND AMORIM, C. L. 1996. Hiding communication latency and coherence overhead in software dsms. In *Proceedings of the seventh international conference on Architectural support for programming languages and operating systems*, ACM Press, 198–209.

BIERMAN, E., AND CLOETE, E. 2002. Classifi cation of malicious host threats in mobile agent computing. In *Proceedings of the 2002 annual research conference of the South African institute*

of computer scientists and information technologists on Enablement through technology, South African Institute for Computer Scientists and Information Technologists, 141–148.

BUTT, A. R., ZHANG, R., AND HU, Y. C. 2003. A self-organizing fbck of Condors. In *Proceedings of Supercomputing*.

BUTT, A. R., FANG, X., HU, Y. C., AND MIDKIFF, S. 2004. Java, peer-to-peer, and accountability: Building blocks for distributed cycle sharing. In *Proceedings 3rd USENIX Virtual Machine Research and Technology Symposium*.

CARTER, J., BENNETT, J., AND ZWAENEPOEL, W. 1991. Implementation and performance of Munin. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles*, 152–164.

COLLBERG, C., CARTER, E., DEBRAY, S., HUNTWORK, A., LINN, C., AND STEPP, M. 2004. Dynamic path-based software watermarking. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation (PLDI)*, ACM.

COX, L. P., AND NOBLE, B. D. 2003. Samsara: Honor among thieves in peer-to-peer storage. In *Proc. 19th ACM Symposium on Operating Systems Principles (SOSP)*.

EATON, J. W. Gnu octave, http://www.octave.org/.

FEI, L., FANG, X., HU, Y. C., AND MIDKIFF, S. P. 2006. Monitoring remotely executing shared memory programs in software DSMs. In *Proceedings of the 20th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*.

GHARACHORLOO, K., LENOSKI, D., LAUDON, J., GIBBONS, P., GUPTA, A., AND HENNESSY, J. 1990. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, 15–26.

GILCHRIST, J. Parallel bzip2 (pbzip2) data compression software, http://compression.ca/pbzip2/.

HU, Y. C., COX, A., AND ZWAENEPOEL, W. 2000. Improving fi ne-grained rrregular shared-memory benchmarks by data reordering. In *Proceedings of IEEE/ACM SC'2000*.

JIANG, X., AND XU, D. 2004. Collapsar: A VM-based architecture for network attack detention center. In *Proceedings of the 13th USENIX Security Symposium (Security '04)*.

JOHNSON, D. B., AND ZWAENEPOEL, W. 1987. Sender-based message logging. In *The 7th annual international symposium on fault-tolerant computing (FTCS)*, IEEE Computer Society.

KELEHER, P., COX, A. L., AND ZWAENEPOEL, W. 1992. Lazy release consistency for software distributed shared memory. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, 13–21.

LAMPORT, L. 1979. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers C-28*, 9 (Sept.), 690–691.

LEE, S.-I., JOHNSON, T. A., AND EIGENMANN, R. 2003. Cetus – an extensible compiler infrastructure for source-to-source transformation. In *Proceedings of the 16th International Workshop on Languages and Compilers for Parallel Computing (LCPC)*.

LI, K., AND HUDAK, P. 1989. Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems 7*, 4 (Nov.), 321–359.

LO, V., ZHOU, D., ZAPPALA, D., LIU, Y., AND ZHAO, S. 2004. Cluster computing on the fly: P2P scheduling of idle cycles in the internet. In *Proceedings of The 3rd International Workshop on Peer-to-Peer Systems (IPTPS)*.

LOUREIRO, S., BUSSARD, L., AND ROUDIER, Y. 2002. Extending tamper-proof hardware security to untrusted execution environments. In *Proceedings of the Fifth Smart Card Research and Advanced Application Conference (CARDIS'02), USENIX - IFIP*.

MEADOWS, C. 1997. Detecting attacks on mobile agents. In *Foundations for Secure Mobile Code Workshop*, 64–65.

PARZEN, E. 1962. On estimation of a probability density function and mode. *Ann. Math. Stat. 33*, 1065–1076.

REGEHR, J. 2002. Inferring scheduling behavior with hourglass. In *In Proceedings of the USENIX Annual Technical Conference (FREENIX Track)*.

SANDER, T., AND TSCHUDIN, C. F. 1998. Protecting mobile agents against malicious hosts. In *Mobile Agents and Security*, Springer-Verlag, 44–60.

SANDER, T., AND TSCHUDIN, C. F. 1998. Towards mobile cryptography. In *Proceedings of the IEEE Symposium on Security and Privacy*.

SAROIU, S., GUMMADI, P. K., AND GRIBBLE, S. D. 2002. A measurement study of peer-to-peer fi le sharing systems. In *Proceedings of Multimedia Computing and Networking (MMCN)*.

SIMPLESCALAR LLC. Simplescalar, http://www.simplescalar.com/.

SINGH, J., WEBER, W.-D., AND GUPTA, A. 1991. SPLASH: Stanford parallel applications for shared-memory. Tech. Rep. CSL-TR-91-469, Stanford University, Apr.

WASSERMAN, H., AND BLUM, M. 1997. Software reliability via run-time result-checking. *Journal of the ACM (JACM) 44*, 6, 826–849.

YANG, S., BUTT, A. R., HU, Y. C., AND MIDKIFF, S. P. 2005. Lightweight monitoring of the progress of remotely executing computations. In *Proceedings of the 18th International Workshop on Languages and Compilers for Parallel Computing (LCPC)*.

YANG, S., BUTT, A. R., HU, Y. C., AND MIDKIFF, S. 2005. Trust but verify: Monitoring remotely executing programs for progress and correctness. In *Proceedings of ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*.

YEE, B. 1994. *Using secure coprocessors*. PhD thesis, Carnegie Mellon University.

YEE, B. S. 1999. A sanctuary for mobile agents. In *Secure Internet Programming*, 261–273.

ZHAO, S., ZHOU, D., LIU, Y., LO, V., AND ZAPPALA, D. 2004. Result verifi cation in open peer-to-peer cycle sharing systems. In *submitted to Sigcomm'04 Poster Session*.

ZHOU, D., AND LO, V. 2004. Cluster computing on the fly: Resource discovery in a cycle sharing peer-to-peer system. In *GP2PC Workshop, CCGrid'04*.

ZORAJA, I., RACKL, G., AND LUDWIG, T. 1999. Towards monitoring in parallel and distributed environments. In *International Conference on Software in Telecommunications and Computer Networks SoftCOM '99*, 133–141.