ACCURATE MEASUREMENT OF SYSTEM CALL SERVICE TIMES FOR TRACE-DRIVEN SIMULATION OF MEMORY HIERARCHY DESIGNS

Richard L. Oliver and Patricia J. Teller†
The University of Texas at El Paso
roliver and pteller@cs.utep.edu

Ward P. McGregor
IBM Corporation
wardp@us.ibm.com

Abstract *This paper presents a methodology for accurately measuring system call service times and process blocking times in a UNIX-like computing environment. Problems that may cause inaccurate measurements stem from the method used to charge processes for execution time, the difference in the time consumed during a system call and a time quantum or dispatch interval, and the instrumentation overhead included in the measurement program. The collected measurements for each system call are used to form a distribution from which process blocking times can be drawn or for which the mean can be used as the process blocking time.*

1. INTRODUCTION

Commonly, the evaluation of memory hierarchies of uniprocessors and multiprocessors is based on simulation results. The most advanced simulators simulate operating system activity, in addition to the activity associated with the user processes that comprise a multiprogram workload. Accuracy of the resultant simulations depends on the accurate interleaving of the memory addresses referenced by the workload user and operating system processes.

Execution-driven simulators like SimOS [Rosenblum, *et al.*, 1995] and Talisman [Bedichek, 1995] simulate user and operating system activity, and accurately interleave the memory addresses referenced by user and operating system processes. A complete multiprogram memory address trace (CMMAT) generated by such an execution-driven simulator is a *scheduled* trace that reflects the characteristics of the simulated architecture and operating system. As explained in [Oliver and Teller, 1998], such a trace cannot be used to accurately simulate a system with a processor cycle time, memory hierarchy design, system call service times, or scheduling policy that is/are different from that of the system that generated the trace.

A *dynamically-scheduled* CMMAT is one that is created dynamically from the augmented uniprogram memory address traces (UMATs) of the workload user processes and the memory address traces of operating system processes that execute system service routines on behalf of the user processes [Oliver and Teller, 1998]. An augmented UMAT includes information that identifies the points in the reference stream at which system calls are generated. Assuming that processes voluntarily context switch when a system call is generated and involuntarily context switch at the termination of a time quantum or when preempted by a process with higher priority, the accurate interleaving of user process trace segments and operating system process traces is dependent on the accuracy of the estimated *process blocking times*, *i.e.*, the time that processes block while waiting for system services. In contrast to the *scheduled* CMMATs described above, *dynamically-scheduled* multiprogram memory address traces can be used to accurately simulate a system with a processor cycle time, memory hierarchy design, system call service times, or scheduling policy that is/are different from that of the system that generated the traces [Oliver and Teller, 1998].

This paper describes our methodology for estimating system call service times. The resultant system call service time estimates are used in simulations of memory hierarchy designs for multiprogrammed computer systems. The paper is organized as follows. Section 2 briefly describes related research. Afterwards, in Sections 3 through 6, we discuss the problems that may be encountered when trying to accurately estimate system service call times, and present our algorithm for measuring system call service times. This is followed, in Sections 7 through 10, by a description of our experimental design and data analysis, and results of our experiments. We conclude in Section 11 with a brief discussion of our future related research. Further details concerning our methodology, the measurement programs, and the complete results of our experiments on the Sun Microsystems SPARCstation ELC can be found in [McGregor, 1997].

2. RELATED RESEARCH

Bugge, *et al.* [1990] generate dynamically-scheduled multiprogram memory address traces (MMATs) which do not include memory addresses referenced by operating system processes. A dynamically-scheduled MMAT is created dynamically from multiple augmented uniprogram memory address traces. The system calls that are identified in the augmented UMATs are members of one of two classes:

†This research is supported by the Defense Advanced Research Program Agency (DARPA), U.S. Department of Defense - Navy Contract #N66001-97-C-8533.

I/O system calls and other system calls.

During simulation the dynamically-scheduled MMAT is created; each of the UMATs is conceptualized as a process, which is considered for execution on the simulated CPU by a scheduler that uses a round-robin, priority-based, preemptive policy. An UMAT contributes addresses to the MMAT only when its associated process is being executed by the simulated CPU, during which time it contributes the addresses that are referenced. The number of references made during a dispatch interval is dependent on the memory delays assessed by the simulated system. The dispatch interval of a process begins as a result of the process being chosen by the scheduler and it ends when a system call in the UMAT associated with the process is encountered or when the scheduling policy dictates the end of the interval (preemption by a higher priority process or time quantum expiration). A process is considered by the scheduler when it is not blocked on a system call. In Bugge's system, when a system call that involves I/O operations is encountered, the associated process is blocked for a random time between 20 and 40 milliseconds. Accordingly, for this simulated time period, its UMAT does not contribute addresses to the MMAT being created. When a system call that does not involve I/O operations is encountered, the process is blocked for a random time between 0.8 and 2.0 milliseconds.

Oliver and Teller [1998] generate dynamically-scheduled MMATs and describe a methodology for generating dynamically-scheduled CMMATs, which include memory addresses referenced by operating system processes. Like Bugge, *et al.* [1990], they use augmented UMATs that identify points at which system calls are generated to denote trace interleaving points. Unlike Bugge, *et al.*, MMATs are created dynamically, drawing the service time of a system call from a distribution for that system call, the distribution having a mean μ and standard deviation σ. The means of distributions for system call service times differ from those of Bugge, *et al.* by an order of magnitude due to the differences in the operating system and hardware used. In addition, many of the standard deviations of the distributions are large. In the remainder of this paper, this methodology is described and resultant means and standard deviations of a large number of UNIX-like system calls are presented.

3. PROBLEMS INHERENT IN ESTIMATING SYSTEM CALL SERVICE TIMES

There are two significant problems that make it difficult to accurately estimate system call service times in a UNIX-like system. The first is associated with the way in which processes are charged for consumed execution time, and the relationship between the time duration of a dispatch interval (time quantum) and system call service. The second problem is associated with the execution time of the instructions that are associated with the measurement activity, *i.e.*, the *instrumentation overhead*, which should not be included in the time measurement of system call service.

4. CHARGED EXECUTION TIME

Tannenbaum states that in most UNIX systems, the strategy that is used to charge time to processes is very prone to error [Tannenbaum, 1992]. In such systems, if a process voluntarily context switches before its allocated time quantum expires, the remainder of the time quantum is allocated to another process. At a clock tick, which signifies the end of a time quantum, the operating system examines the state of the process that is interrupted. If the process is in user/system mode, a full time quantum is charged to the user/system time of that process. Note that a full time quantum is charged to the interrupted process whether or not it consumes the full time quantum. One example of such an operating system is Sun Microsystems OS 4.1.3.

As a result of this problem, a single measurement of a system call may not be reliable. As our results indicate, in Sun Microsystems OS 4.1.3, many system calls have average service times of a few microseconds. In contrast, a time quantum in Sun Microsystems OS 4.1.3 is 10 milliseconds. In these cases, system calls are charged either no time or 10 milliseconds. As a consequence of this problem, and as demonstrated in the next section, the measurement of a system call's service time must be accomplished via a code segment that includes a loop that executes the system call a large number of times.

5. SYSTEM CALL SERVICE TIME MEASUREMENT ALGORITHM

UNIX-like system calls can be partitioned into two distinct categories: (1) those that consume renewable system resources and (2) those that consume finite non-renewable system resources. An example of a system call in category 1 is *getpid*, which only returns the process ID of the calling process. An example of a system call in category 2 is *open*, which consumes file descriptors, of which there are a finite number.

The algorithm that is used to measure the service time of system calls in category 1 is given in Figure 1. The algorithm for the system calls in category 2 is given in Figure 2. Each algorithm executes the system call the same total number of times (*Big Number*). The difference between the algorithms is that the latter one divides these number of executions into *n* sets, each set containing *m* (*Limit*) executions, where *m* is equal to the number of available targeted

system resources.

For these algorithms to produce accurate system call service times, *Big Number* must be chosen carefully and *Loop Overhead* must be predicted accurately or amortized over a large enough number of system calls.

Big Number must be chosen to permit a very large number of system calls to be executed, a number that requires a large number of time quanta. In addition, the number of time quanta must be large enough so that if the measurement code does not consume the last quanta, the error of not accounting for the portion of the last quanta consumed is not significant. *Big Number* can be chosen correctly by increasing it until the variance of the sample means within each trial is minimal. One execution of the measurement code is a trial and each trial computes the specified number of sample means.

```
for i=1 to Number of SampleMeans {
    get time t1
    for j = 1 to Big Number {
        system call
    }
    get time t2
    t = t2 - t1 - Loop Overhead
    Avg = t / Big Number
    print (Avg)
}
```

Figure 1: Service Time Measurement Algorithm for Category 1 System Calls

```
for i=1 to NumberOfSampleMeans {
    for j=1 to (BigNumber/Limit) {
        get time t1
        for k = 1 to Limit {
            system call
        }
        get time t2
        t = t2 - t1
        Acc = Acc + t
        Return system resources
    }
    Acc = Acc - Loop Overhead
    Avg = Acc / Big Number
    print (Avg)
}
```

Figure 2: Service Time Measurement Algorithm for Category 2 System Calls

6. INSTRUMENTATION OVERHEAD

Loop Overhead in the algorithms of Figures 1 and 2 represents the execution time consumed by the instructions executed between the two calls to the system call, *getrusage*, which is used to "get time". We refer to this execution time as the *instrumentation overhead*. To accurately measure system call service time, the instrumentation overhead must be predicted accurately. This can be done by measuring, using the same measuring technique described above, the execution time of a null loop and the execution time of the *getrusage* system call. The null loop is constructed by removing the system call from the measurement code. To measure the null loop, the measurement code is compiled using a non-optimizing compiler. Validation of this method is achieved by inspection of the assembled measurement code and calculation of its execution time on the target RISC processor, using the processor's MHz or MIPS rating and assuming that it executes one instruction per cycle.

7. EXPERIMENTAL DESIGN AND DATA ANALYSIS

The experimental design described in this section and the results of our experiments, which are presented in Section 9, permit us to state that our methodology accurately measures system call service times. This experiment was carried out on three different computer systems: a stand-alone Sun Microsystems SPARCstation ELC running Sun Microsystems OS 4.1.3, which was not connected to the network, and two systems that were connected to the network: (1) a first-generation RS/6000 with a clock speed of 62.5 MHz, a 8-Kbyte L1 instruction cache, and a 64-Kbyte L1 data cache, running AIX 3.2.5, and (2) a SPARCstation 5, Model 110, with a clock speed of 110 MHz, running Sun Microsystems OS 4.1.4.

For any given system call, the measurement code is run at three different times of the day so that the observations reflect the execution times of the system call at various levels of system activity. Each execution of the measurement code is called a trial and, in our case, each of three trials collects 10 measurements or observations. Each observation is the average execution time of the subject system call; in most cases, the subject system call is executed 1 million times, *i.e.*, *Big Number* is equal to 1 million.

The observations are analyzed using the one-way ANOVA to determine if there is a significant difference between the observations collected within each trial and the observations collected among the total number of trials. This analysis determines if the state of the system at the time at which the observations are taken influences the measurements of system call service time. Since our methodology is designed to accurately measure system call service time regardless of system activity, we want to test the experimental hypothesis that the system call service time measurement is affected by system activity. To reject the experimental hypothesis, and accept the null hypothesis, the three trials must have a common variance.

To test the null hypothesis we use the one-way ANOVA. We compare the sample variances among trials, *Among-SM,* with the sample variances within trials, *Within-SM*. *Among-SM* denotes the variance of the overall mean of each trial. It estimates the dispersion in the sampling distribution of the averages of each trial. Thus, it must be multiplied by $n = 10$ to estimate the dispersion of the original distribution [Dowdy and Wearden, 1991]. The degree of freedom for the one-way ANOVA is: (a-1), a * (n-1), where a is the number of trials and n is the number of sample means per trial. For our experiments, where $a = 3$ and $n = 10$, the degree of freedom is 2,27 (numerator, denominator). *Within-SM* is the average of the three trial variances (overall means of the trials) weighted by the degrees of freedom for all the sample means, which in our case is 27.

The test of the hypothesis concerning equality of means is an F test for the equality of two variances, where F = *Among-SM / Within-SM*. Thus, if the null hypothesis holds then the *Within-SM* for each trial is not significantly different from the *Among-SM*. This relationship makes it possible for us to use the one-way ANOVA to determine the similarity of the measurements collected and ensures the validity of our experiment.

Within-SM represents the variance due to random sampling and is caused by sampling error. In order to construct a meaningful confidence interval for our tests, we use the standard error of the overall mean

$$\varepsilon = \sqrt{\frac{Within - SM}{N}}$$

where N is the total number of observations. The confidence interval is determined by the overall average of the observations +- $t_{a/2,N-a}$ (the critical value of the Student's t distribution needed for inference [Dowdy and Wearden, 1991]) and a is the rejection level of our experiment, the probability of committing a Type 1 error in our experiment. A Type 1 error occurs whenever we reject the null hypothesis when it really is true. In our experiment, a = 0.01, which means that we are 99% certain that the true measurement of the system call execution times fall within our confidence interval.

8. VALIDATION OF EXPERIMENTAL MEASUREMENTS

To validate the experimental measurements, a simple mathematical model is used. The mathematical model is:

t = (L^1 * *Loop Overhead*) +
(2* L^1 * S*ystemCallExecutionTime*) +
(L^1 * L^2 * *Loop Overhead*)

where *Loop Overhead* and *SystemCallExecutionTime* are defined by the experimental method described above in Section 5. The system call execution time can be defined by the mean of the collected measurements or it can be drawn from the distribution that is comprised of the collected observations. Given L^1 and L^2, t is computed.

The program of Figure 3, which also computes t, is executed to validate the correctness of the collected measurements that are used in the mathematical model. Since the inner loop is a null loop, as explained above in Section 6, L^2 must be chosen carefully. In our experiments, L^2 is 1,000,000.

```
getrusage t1
for i= 1 to L1 {
   system call
   for j = 1 to L2
   { }
   system call
}
getrusage t4
t = t4 - t1
print (t)
```

Figure 3: Validation Program

All of our collected measurements were validated in this way.

9. EXPERIMENTAL RESULTS

The experiment described above was conducted on three different computer systems. The results of these experiments are given below. Table 4 gives the timing results (in seconds) for all three systems. It indicates for each system call and for each system the mean in milliseconds and the standard deviation in milliseconds. In Table 4, System 1 is the Sun Microsystems SPARCstation ELC, System 2 is the RS/6000, and System 3 is the Sun Microsystems SPARCstation 5. System call names that are followed by an asterisk indicate that these system calls are in category 2, *i.e.*, system calls that consume finite renewable resources.

Note that the standard deviations are very small and the measurements were validated using the mathematical model described above. Thus, our methodology produced good and accurate estimates of system call service times and, accordingly, process blocking times.

Measurement programs were not developed for all of the system calls that do not consume finite non-renewable system resources. The system calls that were not measured are: *fork*, *exec* (and all the associated exec system calls), *plock*, *shmget*, and the basic UNIX-like network operating system calls such as: *bind*, *listen*, *send*, and *recv*. These system calls were not generated by the workloads that we used to study memory hierarchy designs. The algorithms presented in Figures 1 and 2 can be extended to develop measurement programs for these system calls.

TABLE 4. System Call Service Times

	System 1		System 2		System 3	
System Call	**Mean**	**Std. Dev.**	**Mean**	**Std. Dev.**	**Mean**	**Std. Dev.**
chdir	0.4261	0.0005789				
chmod	0.7448	0.0013348				
chown	0.1986	0.0000226				
creat	0.0712	0.0000770				
fstat	0.4220	0.0000000	0.036343333	0.00005234		
getdents	0.0229	0.0000130				
gethostname	0.0218	0.0000256				
getpagesize	0.0201	0.0000256	0.000225	0.00000309	0.016391	0.000007128
getpid	0.0207	0.0000192	0.004386667	0.00001742	0.01870333	0.000009237
getpriority	0.0369	0.0000236	0.034483333	0.00018957	0.03004333	0.000028386
getrusage	0.0327	0.0000240	0.013443333	0.00003758	0.02298	0.000020882
getsockname	0.0238	0.0000228	0.006374	0.00000759	0.018882	0.000010868
gettablesize	0.0202	0.0000230	0.000226	0.00000346	0.01631067	0.000006603
gettimeofday	0.0206	0.0000315				
getuid	0.0209	0.0000232	0.004406333	0.00000747	0.01838833	00.00715
link	1.3590	0.0046431				
lstat	0.2205	0.0005559				
stat	0.2100	0.0002482				
symlink	0.6763	0.0003833				
setitime	0.1177	0.0000000				
setuid	0.0659	0.0000224	0.016703667	0.00003387	0.039079	0.000010029
acct	0.0271	0.0004343				
getdomainname	0.0218	0.0002576			0.02164	0.000255195
getegid	0.0225	0.0003063	0.004676667	0.00005359	0.017536667	0.00020022
getgid	0.0257	0.0002263	0.004673333	0.00007777	0.01835	0.00029340
gethostid	0.0549	0.0005760	0.0063	0.00005456	0.04311333	0.00006173
getpeername	0.0237	0.0002345	0.006356667	0.00001726	0.01774967	0.00001057
getuid	0.0209	0.0002576				
nice	0.0483	0.0007273				
socket*	0.0779	0.0001354				
getsockopt	0.0240	0.0000000				
access	0.2029	0.0002576	0.053786667	0.00037590	0.09658333	0.000075716
getlimit	0.0946	0.0006946	0.026956667	0.00001079	0.05488333	0.000117362
mknod*	0.6707	0.0006682				
unlink	0.1628	0.0011847				
write*	0.3538	0.0008091				
fsync	0.0320	0.0002097				
listen	0.0237	0.0005785				
read	0.3397	0.0006935	0.03182833	0.00008672	0.11981633	0.000212280
flock	0.0269	0.0007903				
ioctl*	0.0508	0.0007598				
sigblock	0.0217	0.0004472				
sigvec	0.0756	0.0001472				
open	0.2377	0.0024510	0.18829333	0.00089291	0.42107666	0.000311822
select	0.0491	0.0001188				
profil	0.0286	0.0002405				
sigsetmask	0.0215	0.0004493				
fcntl	0.0241	0.0008358				

10. EXTRAPOLATION OF RESULTS

These results can be extrapolated to systems that differ from a measured system with respect to clock rate. The assumption is that the systems run the same operating system and essentially have the same implementation. In this case, the same number of cycles is required to execute a given system call on either system. If the measured system requires *n* cycles to execute a system call, then one can compute a mean and standard deviation for a similar system by using this cycle count and the difference in Mhz ratings of the two systems. We validated this extrapolation by running our measurement and data analysis programs on a 33 MHz Sun Microsystems SPARC rated at 21 MIPS, estimating system call service times on a 87 MHz Sun Microsystems SPARC via this extrapolation technique, and running our measurement and data analysis programs on the 87 MHz system. The estimated system call service times are in excellent agreement with the measured service times.

11. FUTURE WORK

The results of these experiments have been used in simulations that study the performance of memory hierarchies of multiprogrammed computer systems [Oliver, 1997]. We plan to use this data in other computer system simulations as well. In addition, we plan to use this experimental method to ascertain system call service times in multiprocessor systems. The results of this will be used in simulations of memory hierarchies of multiprocessor systems.

12. BIBLIOGRAPHY

Bedichek, R., "Talisman: Fast and Accurate Multicomputer Simulation," *Proceedings of the 1995 ACM Sigmetrics Joint International Conference on Measurement and Modeling of Computer Systems*, pp. 14-24, May 14-19, 1995.

Bugge, H., E. Kristiansen, and B. Bakka, "Trace Driven Simulations for a Two-Level Cache Design in Open Bus Systems," *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pp. 250-259, May 28-31, 1990.

Dowdy, S., and S. Wearden, *Statistics for Research*, N.Y.C., N.Y., Second Edition, John Wiley & Sons, Inc., 1991.

McGregor, W., *UNIX-like System Calls Analysis*, Master's Thesis, New Mexico State University, Department of Computer Science, May 1997.

Oliver, R., *Process Execution Characteristics and their Use in Memory Hierarchy Design*, Doctoral Dissertation, New Mexico State University, August 1997.

Oliver, R., and P. Teller, "Generating Dynamically Scheduled Memory Address Traces," *Proceedings of the 1998 IEEE International Performance, Computing and Communications Conference,* Phoenix/Tempe, Arizona, February 16-18, 1998.

Rosenblum, M., S. Herrod, E. Witchel, and A. Gupta, "Complete Computer System Simulation: The SimOS Approach," *IEEE Parallel & Distributed Technology*, pp. 34-43, Winter 1995.

Tannenbaum, A., *Modern Operating Systems*, Englewood Cliffs, N.J., Prentice Hall, Inc., 1992.