

# CPU TIME MEASUREMENT ERRORS

Paper 2038 - CMG98

The most common method of measuring CPU time in Unix is to sample the state of each CPU at each clock interrupt, and to accumulate global and per-process counters. The hope is that sampling a large number of values the average will be accurate over time. Sampling theory requires an unbiased sample, but the clock interrupt is also used to schedule wake-ups. The bias causes large errors to accumulate, particularly for processes that wake up for a short time at regular intervals. Using alternative CPU measurement techniques, errors are analyzed over a range of workloads. The highest error levels are found at low load levels with fast CPUs. Measured rather than sampled CPU time should be used where it is available.

## 1.0 INTRODUCTION

CPU resource consumption is measured by most operating systems at two levels. One is the systemwide usage, the other is the per-process usage. The data is used for problem identification and capacity planning purposes.

For capacity planning, workload analysis is performed by summing the resource usage of groups of related processes. This is then fed into a model that can be used to predict resource usage with different workload mixes and different system configurations. Such a model is only as good as the data it uses as input, and assumptions are made about the kind of errors that are present. Specifically, if the error is random, then averaging many measurements together reduces the error.

This topic was discussed in a paper by Agrawal, Newman, Forsyth and Somin at CMG96, where the statistical basis of measurement errors was explored in some detail. That paper provided the inspiration for this study of actual error levels.

In the case of CPU measurements, it is very hard to obtain validation of the error levels in a measurement, as most operating systems only provide a single measurement method. The Solaris 2 operating system does provide an alternative measurement method, so the results can be compared.

Differences between the measurement methods are explained and analyzed in this paper. Experiments were performed on a range of systems running various workloads, and an analysis of the errors was performed.

Results show that the errors range from a few percent at high usage levels to 80% or more at low levels. Usage is under reported, and the error increases on faster CPUs. At a real usage level of 5% busy, `ps` and `vmstat` may report that the system is only 1% busy, under reporting by 80% of the true value. You could also look at this as a 400% error in the reported value. As an example of the kind of problem this can cause, consider a system planned to cope with a load of up to 1000 users. Measuring the average process activity of the first 20 users they only appear to use 1% of the system (but in fact use 5%). There seems to be capacity for 2000 users, whereas there is really only enough capacity for 400. As the total user load increases, the amount of CPU used by each user also appears to increase as the measurement error reduces.

## 2.0 CPU USAGE MEASUREMENTS

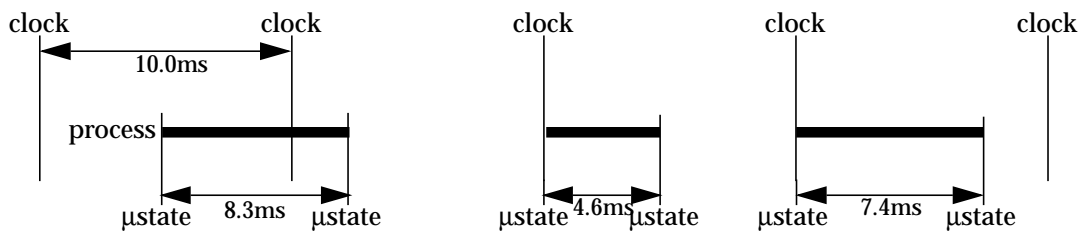
The way CPU time is measured in most Unix systems is to sample, 100 times a second, the state of all the CPUs during the clock interrupt. This method is used by the common Berkeley and System V derived versions. The alternative method in Solaris 2 is called Microstate Accounting. Microstate accounting works by taking a high-resolution timestamp on every state change, every system call, every page fault, every scheduler change. Process scheduling in many versions of Unix, including Solaris 2, uses the same clock interrupt used to measure CPU usage, and this approach leads to systematic errors in the sampled data. The microstate-measured CPU usage data does not suffer from those errors.

For example, consider a process that wakes up every few seconds, does a little work then sleeps. On a fast system, the total CPU time consumed per wake-up might be a few milliseconds. On exit from the clock interrupt, the scheduler wakes up processes and kernel threads that have been sleeping until that time. Processes that sleep then consume less than their allotted CPU time quanta always run at the highest timeshare priority. On a lightly loaded system there is no queue for access to the CPUs, so immediately after the clock interrupt, it is likely that the process will be scheduled. If it runs for less than 10 milliseconds it will have completed and be sleeping again by the time the next clock interrupt comes along. Remember that the only way that CPU time is allocated is based on what was running when the clock interrupt occurs. The process could be sneaking a bite of CPU time whenever the clock interrupt isn't looking. This is an artifact of the dual functions of the clock interrupt. When there is a significant amount of queuing for CPU time, the process will be delayed by a random amount of time, so it will be seen by the clock interrupt some of the time. Increasing the CPU performance decreases the time spent by the process in each wakeup, so increases the error. Increasing the clock interrupt rate reduces this error but adds to system overhead.

If two independent unsynchronized interrupts were used, one for scheduling and one for performance measurement, then the errors would be averaged away over time. This approach has higher overhead, but allows finer resolution wake-ups. IBM's AIX uses independent interrupts, so when averaged over a sufficiently long time, biased errors should not accumulate. The microstate measurements provided by Solaris 2, and a similar facility in HP-UX, do not need to be averaged over a long time to be accurate. It is also possible to increase the Solaris 2 clock rate to 1000Hz by tuning the kernel with `set hires_tick=1`.

To illustrate the difference in measurements, in FIGURE 1., a process wakes up, then sleeps twice. The first wakeup occurs between clock ticks and the period is interrupted by the subsequent tick, which charges a full 10 milliseconds to the process. The next two wakeups occur as a result of the clock interrupt scheduling the process, and they complete before the subsequent interrupt, so there is no charge. The true measured CPU usage is measured by microstate accounting as  $8.3 + 4.6 + 7.4 = 20.3\text{ms}$ . The first wakeup is over estimated, the second and third are missed completely.

FIGURE 1. SAMPLING EXAMPLE



### 3.0 PROCESS DATA SOURCES

These data structures are described in full in the Solaris 2 `proc(4)` manual page.

The interface to `/proc` involves sending `ioctl` commands or opening special pseudo-files and reading them (a new feature of Solaris 2.6). The data that `ps` uses is called `PIOCPSINFO`, and this is what you get back from `ioctl`. You get slightly different data if you read it from the pseudo-file.

FIGURE 2. PROCESS PSINFO DATA

proc(4)	File Formats	proc(4)
PIOCPSINFO		
This returns miscellaneous process information such as that reported by <code>ps(1)</code> . <code>p</code> is a pointer to a <code>prpsinfo</code> structure		

FIGURE 2. PROCESS PSINFO DATA

containing at least the following fields:

```
typedef struct prpsinfo {
    char      pr_state;      /* numeric process state (see pr_sname) */
    char      pr_sname;     /* printable character representing pr_state */
    char      pr_zomb;      /* !=0: process terminated but not waited for */
    char      pr_nice;      /* nice for cpu usage */
    u_long    pr_flag;      /* process flags */
    int       pr_wstat;     /* if zombie, the wait() status */
    uid_t     pr_uid;       /* real user id */
    uid_t     pr_euid;      /* effective user id */
    gid_t     pr_gid;       /* real group id */
    gid_t     pr_egid;     /* effective group id */
    pid_t     pr_pid;       /* process id */
    pid_t     pr_ppid;      /* process id of parent */
    pid_t     pr_pgrp;      /* pid of process group leader */
    pid_t     pr_sid;       /* session id */
    caddr_t   pr_addr;      /* physical address of process */
    long      pr_size;      /* size of process image in pages */
    long      pr_rssize;    /* resident set size in pages */
    u_long    pr_bysize;    /* size of process image in bytes */
    u_long    pr_byrssize;  /* resident set size in bytes */
    caddr_t   pr_wchan;     /* wait addr for sleeping process */
    short     pr_syscall;   /* system call number (if in syscall) */
    id_t      pr_aslwpid;   /* lwp id of the aslwp; zero if no aslwp */
    timestruc_t pr_start;   /* process start time, sec+nsec since epoch */
    timestruc_t pr_time;    /* usr+sys cpu time for this process */
    timestruc_t pr_ctime;   /* usr+sys cpu time for reaped children */
    long      pr_pri;       /* priority, high value is high priority */
    char      pr_oldpri;    /* pre-SVR4, low value is high priority */
    char      pr_cpu;       /* pre-SVR4, cpu usage for scheduling */
    u_short   pr_pctcpu;    /* % of recent cpu time, one or all lwps */
    u_short   pr_pctmem;    /* % of system memory used by the process */
    dev_t     pr_ttydev;    /* controlling tty device (PRNODEV if none) */
    char      pr_clname[PRCLSZ]; /* scheduling class name */
    char      pr_fname[PRFNSZ]; /* last component of exec()ed pathname */
    char      pr_psargs[PRARGSZ]; /* initial characters of arg list */
    int       pr_argc;      /* initial argument count */
    char      **pr_argv;    /* initial argument vector */
    char      **pr_envp;    /* initial environment vector */
} prpsinfo_t;
```

For a multithreaded process, you can get the data for each lightweight process separately. There's some useful-looking information in PSINFO, but

high-resolution microstate accounting data uses a separate `ioctl`, `PIOCUSAGE`.

FIGURE 3. PROCESS USAGE DATA

proc(4) File Formats proc(4)

#### PIOCUSAGE

When applied to the process file descriptor, `PIOCUSAGE` returns the process usage information; when applied to an lwp file descriptor, it returns usage information for the specific lwp. `p` points to a `prusage` structure which is filled by the operation. The `prusage` structure contains at least the following fields:

```
typedef struct prusage {
    id_t      pr_lwpid;     /* lwp id. 0: process or defunct */
```

FIGURE 3. PROCESS USAGE DATA

```

    u_long      pr_count; /* number of contributing lwps */
    timestruc_t pr_tstamp; /* current time stamp */
    timestruc_t pr_create; /* process/lwp creation time stamp */
    timestruc_t pr_term; /* process/lwp termination timestamp */
    timestruc_t pr_rtime; /* total lwp real (elapsed) time */
    timestruc_t pr_utime; /* user level CPU time */
    timestruc_t pr_stime; /* system call CPU time */
    timestruc_t pr_ttime; /* other system trap CPU time */
    timestruc_t pr_tftime; /* text page fault sleep time */
    timestruc_t pr_dftime; /* data page fault sleep time */
    timestruc_t pr_kftime; /* kernel page fault sleep time */
    timestruc_t pr_ltime; /* user lock wait sleep time */
    timestruc_t pr_slptime; /* all other sleep time */
    timestruc_t pr_wtime; /* wait-cpu (latency) time */
    timestruc_t pr_stoptime; /* stopped time */
    u_long      pr_minf; /* minor page faults */
    u_long      pr_majf; /* major page faults */
    u_long      pr_nswap; /* swaps */
    u_long      pr_inblk; /* input blocks */
    u_long      pr_oublk; /* output blocks */
    u_long      pr_msnd; /* messages sent */
    u_long      pr_mrcv; /* messages received */
    u_long      pr_sigs; /* signals received */
    u_long      pr_vctx; /* voluntary context switches */
    u_long      pr_ictx; /* involuntary context switches */
    u_long      pr_sysc; /* system calls */
    u_long      pr_ioch; /* chars read and written */
} prusage_t;

PIOCUSAGE can be applied to a zombie process (see
PIOCPINFO).

Applying PIOCUSAGE to a process that does not have micro-
state accounting enabled will enable microstate accounting
and return an estimate of times spent in the various states
up to this point. Further invocations of PIOCUSAGE will
yield accurate microstate time accounting from this point.
To disable microstate accounting, use PIOCRESET with the
PR_MSACCT flag.

```

There is a lot of useful data here, on top of the CPU time measurements. The time spent waiting for various events is very useful for capacity plan-

ning and performance modeling. The timers can be summarized as shown in

FIGURE 4. MICROSTATE DATA SUMMARY

Elapsed time	3:20:50.049	Current time	Fri Jul 26 12:49:28 1996
User CPU time	2:11.723	System call time	1:54.890
System trap time	0.006	Text pfault sleep	0.000
Data pfault sleep	0.023	Kernel pfault sleep	0.000
User lock sleep	0.000	Other sleep time	3:16:43.022
Wait for CPU time	0.382	Stopped time	0.000

#### 4.0 CPU USAGE ERROR CHECKING TOOL

There is a freely available performance toolkit for Solaris 2 that provides access to all the kernel data sources via a C-based scripting language.

Known as the SE toolkit it includes a process monitoring class. This uses both the above data sources to report the measured CPU usage, but if microstate accounting is not enabled for a process then the USAGE value returned is just the same

as the PSINFO value. The process class was modified to report sampled CPU time as a separate value, and to explicitly set the microstate accounting flags so that every process and children of those processes will have accurate measurements enabled. There is new programming interface that was introduced in Solaris 2.6, so the process class contains code for both the old and the new interfaces.

In earlier releases, Solaris 2.4 to 2.5.1, microstate data is obtained by issuing an `ioctl` call with the `PIOCUSAGE` flag. This also automatically turns on microstate data collection (this interface is still supported but will go away in a future release). In Solaris 2.6, Data is obtained by reading `/proc/pid/usage`, which no longer requires special permissions, but which *no longer turns on microstate data collection*. The data returned is an approximation based on the sampled measurements. To turn on the flags a control message is written to `/proc/pid/ctl`, which does require access permissions. To collect microstate enabled data for all the processes on the system this code must be run as root.

The tool that collects data is called `cpuchk.se`, and is loosely based upon the process monitor example script `pea.se`. It compares the sampled and measured data for each interval for each active process, calculates the error and prints the results. It also calculates the overall CPU usage totals and the total, absolute and maximum errors. The total error is lower as positive and negative errors are allowed to cancel each other out. The absolute error is the sum of errors without any cancellation, the maximum is the highest absolute error seen. All errors are calculated relative to the accurately measured result, for processes that report non-zero usage. If you start with the inaccurate sampled result and try to calculate errors they are much larger, in some cases infinite, where the sampled value is zero.

Test were run using `cpuchk.se` using several sample intervals, it doesn't seem to affect the results so long term data collection on several machines was run with a 10 minute interval. This only collects long running processes, but keeps the load level from the `cpuchk.se` command itself to a minimum. Some sample output data is shown in FIGURE 5.. The first line shows the time of day, the number of processes and the number

of processes seen for the first time. Subsequent lines show the error for each active process. The last line shows how many processes were totalled (system processes like `sched` and `fsflush` cannot have microstate enabled so they are excluded).

FIGURE 5. EXAMPLE CPUCHK OUTPUT

```

00:17:10 cpu time accuracy check  proc 45  new 0
pid 1435 meas 0.000 samp 0.000 err 100.00%
pid 316 meas 0.001 samp 0.000 err 100.00%
pid 1438 meas 0.000 samp 0.000 err 100.00%
pid 227 meas 0.001 samp 0.002 err 28.05%
pid 211 meas 0.011 samp 0.008 err 25.80%
pid 226 meas 0.032 samp 0.035 err 7.69%
pid 229 meas 0.018 samp 0.002 err 90.92%
pid 246 meas 0.083 samp 0.060 err 28.00%
pid 318 meas 0.000 samp 0.000 err 100.00%
pid 380 meas 0.143 samp 0.003 err 97.67%
pid 1439 meas 0.000 samp 0.000 err 100.00%
pid 357 meas 0.000 samp 0.000 err 100.00%
pid 518 meas 0.125 samp 0.000 err 100.00%
pid 7376 meas 0.041 samp 0.000 err 100.00%
pid 7377 meas 0.000 samp 0.000 err 100.00%
pid 6276 meas 0.156 samp 0.156 err 0.37%
pid 6262 meas 2.413 samp 0.002 err 99.93%
pid 9199 meas 0.221 samp 0.225 err 1.56%
pid 9200 meas 0.206 samp 0.202 err 2.20%
pid 9209 meas 2.308 samp 2.333 err 1.05%
msac 42 meas 5.763 samp 3.027 err -47.48% abs
48.56% max 100.00%

00:27:11 cpu time accuracy check  proc 45  new 0
pid 1435 meas 0.000 samp 0.000 err 100.00%
pid 316 meas 0.001 samp 0.000 err 100.00%
pid 1438 meas 0.000 samp 0.000 err 100.00%
pid 227 meas 0.003 samp 0.002 err 46.53%
pid 211 meas 0.010 samp 0.007 err 35.85%
pid 226 meas 0.032 samp 0.022 err 31.33%
pid 229 meas 0.015 samp 0.002 err 88.77%
pid 246 meas 0.074 samp 0.052 err 30.70%
pid 318 meas 0.002 samp 0.000 err 100.00%
pid 380 meas 0.143 samp 0.000 err 100.00%
pid 1439 meas 0.000 samp 0.000 err 100.00%
pid 357 meas 0.000 samp 0.000 err 100.00%
pid 518 meas 0.122 samp 0.000 err 100.00%
pid 379 meas 0.144 samp 0.126 err 12.00%
pid 7376 meas 0.040 samp 0.000 err 100.00%
pid 7377 meas 0.000 samp 0.000 err 100.00%
pid 6276 meas 0.155 samp 0.155 err 0.00%
pid 6262 meas 2.410 samp 0.003 err 99.86%
msac 39 meas 3.152 samp 0.368 err -88.33% abs
88.33% max 100.00%

```

## 5.0 ANALYSIS AND GRAPHED RESULTS

The measured CPU time and the absolute error from the output were extracted using `awk` and fed into a statistics package (S-PLUS from [www.statsci.com](http://www.statsci.com)). After looking at the data for individual processes for a while attention was concentrated on the summaries for each measurement interval. First the error and value were plotted together in time sequence, then error was plotted as a function of CPU usage. The relationship is basically an inverse one, so an inverse

relationship line was fitted and displayed. The systems monitored were a SPARCstation 10 with dual 60 MHz CPUs; an E4000 with two 167 MHz CPUs; a uniprocessor Ultra-1 167 MHz desktop workstation, a Tadpole 85 MHz microSPARC laptop and an E6000 with twenty 248 MHz processors.

## 5.1 Analysis Scripts

S-PLUS reads tabular data into an object called a data frame. The data frame is then processed using commands that can be combined into scripts to automate repetitive operations. The scripts used in this case included an awk script to pick out the measured cpu and absolute error totals:

```
% awk '/msac/ {printf("%s %s\n", $4, substr($10, 0,
length($10)-1));}' <cpuchk.output >cpuchk.msac
```

This is read into S-PLUS using the command

```
> cpuchk.msac <- read.table("cpuchk.msac",
col.names=c("cpu", "error"))
```

A script that plots the data sample by sample was written:

```
> ichk <- function(msac, main = "CPU %busy and error over
time")
{
  plot(msac$error, ylim = c(0, max(c(msac$error,
msac$cpu))), main = main,
      ylab = "CPU busy line and error *")
  lines(msac$cpu)
}
```

It is invoked using this command:

```
> ichk(cpuchk.msac, main="CPU %busy and error over time for
E6000 Email")
```

The script that plots the error rates with a fitted curve looks like this:

```
> lmchk <- function(msac, main = "Error versus CPU load",
f = T)
{
  l <- glm(error ~ cpu, data = msac, family = Gamma())
  fit <- predict(l, newdata = data.frame(cpu =
0:max(msac$cpu)))
  plot(msac$cpu, msac$error, ylim = c(0,
max(msac$error)), main = main,
      xlab = "Total measured CPU %busy", ylab =
"% error in sampled CPU reported")
  if(f) {
    lines(0:max(msac$cpu), 1/fit)
    return(fit)
  }
}
```

The `lmchk` function uses a generalized linear model with a gamma link function to fit error as an inverse function of `cpu`, then predicts fitted values over the range before plotting the data points and

the fit. The fit line can be disabled. It is invoked using this command:

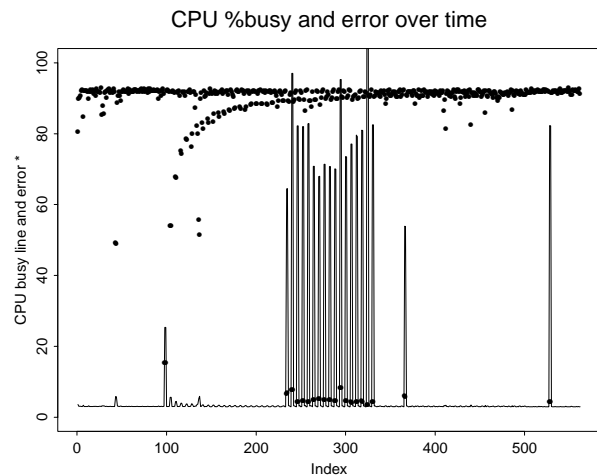
```
> lmchk(cpuchk.msac, main="Error versus CPU load for E6000
Email")
```

These two functions produce the two plots shown for each system in the subsequent sections.

## 5.2 SPARCstation 10

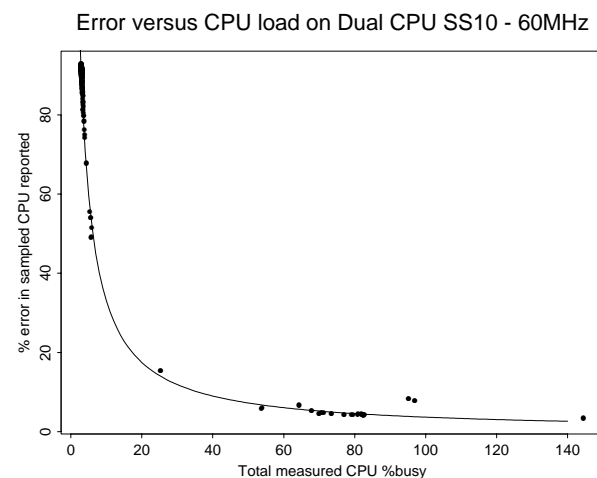
The SPARCstation 10 with dual 60MHz SuperSPARC CPUs is an obsolete design dating from about 1994. It is a lightly used web server and runs CPU intensive batch jobs from `cron` at regular intervals. The time based plot shows that it is mostly idle with regular batch jobs.

FIGURE 6. SS10 Error Over Time



Errors show a good fit to the inverse line, probably because the workload doesn't vary much.

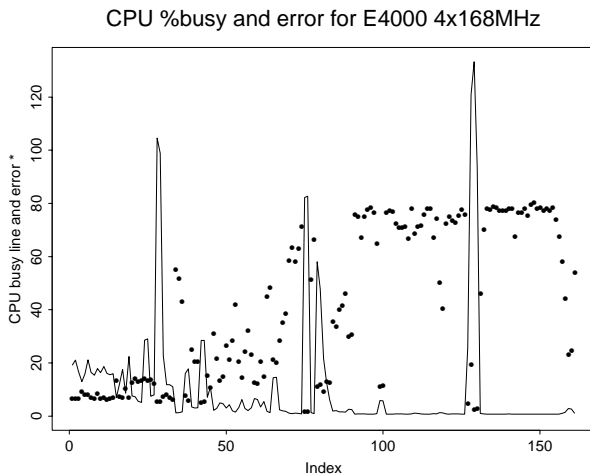
FIGURE 7. SS10 Error vs. Load



### 5.3 E4000 Server

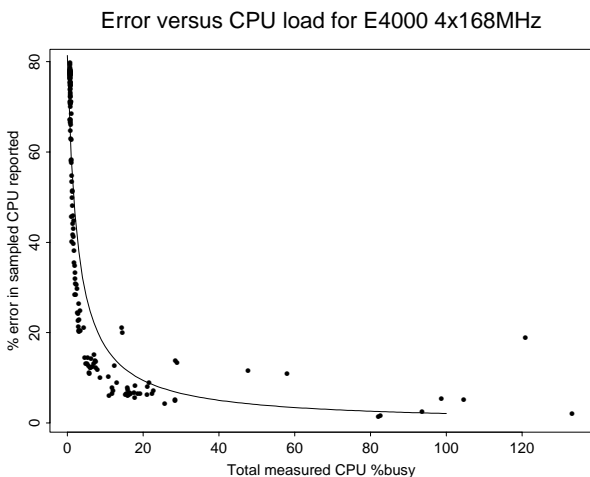
The E4000 with two 167 MHz UltraSPARC CPUs is a workgroup server, runs email and NFS services amongst other things. This is a small E4000 configuration, with slow processors compared to current products. It supports a small number of "power users" and the load is spiky.

FIGURE 8. E4000 Error Over Time



The workload mix varies, but the fit shown in FIGURE 9 is still a reasonable one. The data looks like several distinct curves overlaid on one another, but close together.

FIGURE 9. E4000 Error vs. Load



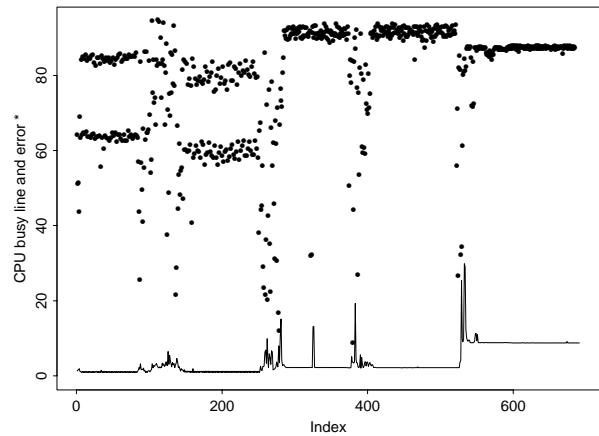
### 5.4 Ultra 1 Desktop

The Ultra 1/170, with a single 167 MHz UltraSPARC CPU was running the X-based common desktop environment (CDE) window system and included some web browser screens with animated gifs, and a java application that was started

towards the end. The java application ran an idle loop that slept then polled for input, and consumed about 6% of the CPU while reporting less than 0.5%. Overall this period sustained a real usage rate of 8.8% with only 1.1% reported via sampling.

FIGURE 10. Ultra 1 Error Over Time

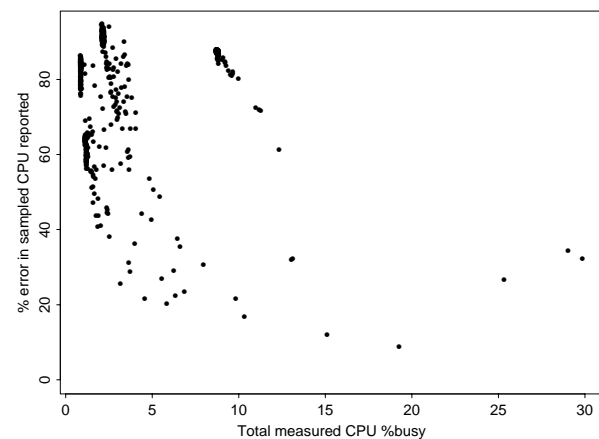
CPU %busy and error over time for Ultra1 167MHz



When we look at the error as a function of the measured usage, it shows several separate clusters of data, each of which could have its own fitted curve. No overall curve could be fitted to this data.

FIGURE 11. Ultra 1 Error vs. Load

Error versus CPU load for Ultra1 167MHz

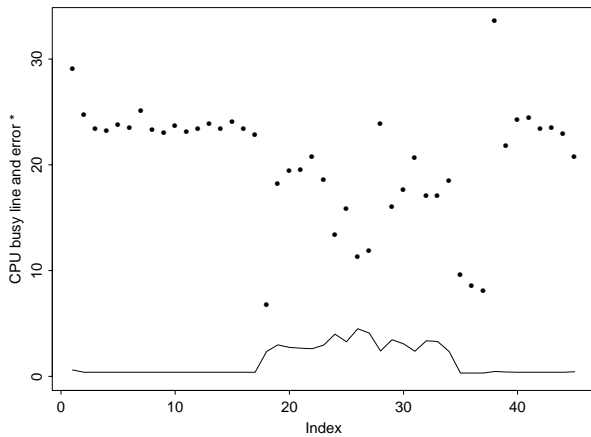


### 5.5 MicroSPARC Laptop

Some data from a much slower CPU - 85MHz microSPARC - shows that the error levels are smaller, as we would expect. The microSPARC processor has about half the performance at a given clock rate compared to the other systems tested. This system is functionally equivalent to

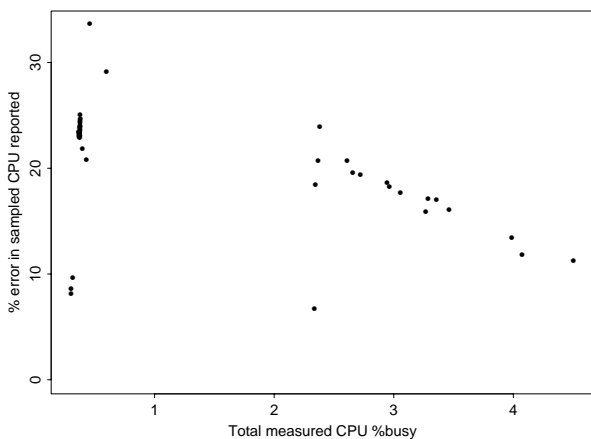
the now obsolete entry level SPARCstation 4 and SPARCstation 5 desktop systems.

FIGURE 12. MicroSPARC Error Over Time  
CPU %busy and error over time for 85MHz microSPARC



The measured load level is low all of the time, and the results are too scattered to obtain a good fit.

FIGURE 13. MicroSPARC Error v.s Load  
Error versus CPU load for 85MHz microSPARC



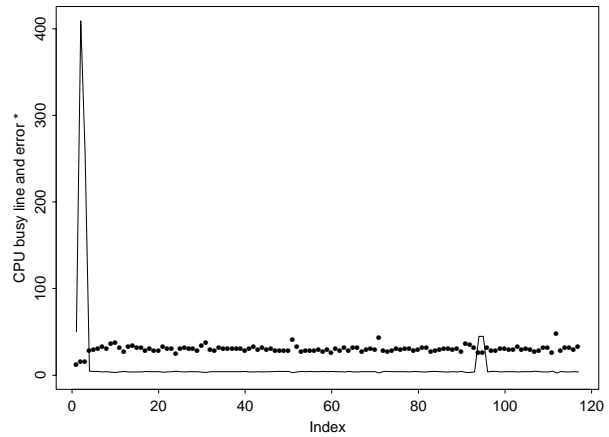
### 5.6 E6000 Server

The twenty processor 248 MHz UltraSPARC CPU E6000 server was being used to run benchmark workloads. Results were obtained during tests that did not attempt to saturate the whole machine, running high levels of SPECweb96 (web server), and a POP/IMAP email server workload, both followed by idle time.

The SPECweb98 test finished very quickly. For the few samples taken at ten minute intervals during high load levels of 400%, the error is 10-15%. This corresponds to four busy processors and an

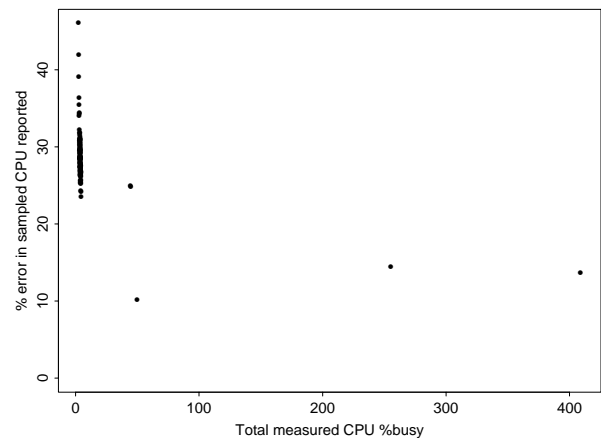
error of about half a processor. When the machine became idle again the error increased to about 30-40%.

FIGURE 14. E6000 Web Error Over Time  
CPU %busy and error over time for E6000 Web



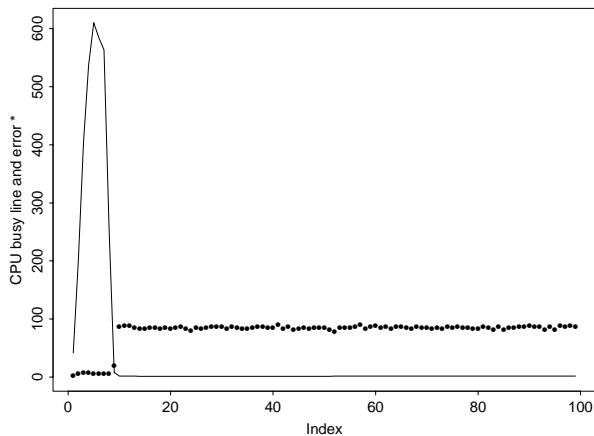
There is insufficient data here to fit a good curve to it.

FIGURE 15. E6000 Web Error vs. Load  
Error versus CPU load for E6000 Web



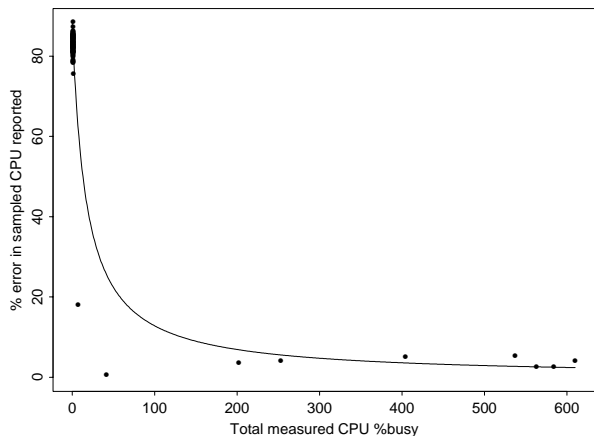
The email test involved IMAP and POP based mail client emulation running against Sun's Solstice Internet Mail Server version 3.2. This is a complex multithreaded server process, and the load was high and random enough to keep the error under 5%. In this case the load peaked at about 600%, using six out of the total of twenty CPUs, and the error is about a quarter of a CPU. When the load reduced to idle (2-3%) the relative error increased again to about 80%.

FIGURE 16. E6000 Email Error Over Time  
CPU %busy and error over time for E6000 Email



This time there was enough data to get a reasonable fitted curve as shown in FIGURE 17.

FIGURE 17. E6000 Email Error vs. Load  
Error versus CPU load for E6000 Email



## 6.0 ADDITIONAL VARIATIONS

CPU time measurements do not take into account some additional sources of error. Interrupt processing can never be seen by the clock interrupt, which runs at a lower priority than most device interrupts. The sampled CPU measures do not include interrupt time, and higher priority device interrupts can cause jitter in the exact timing of the clock interrupt. Microstate measurements see the effect of interrupts as an elongation of the time spent running a process, but do not separate out the time spent servicing the interrupts. There are also times when the CPU traps into the kernel without making a system call. These traps are not measured by the clock interrupt, so sampled system time may be under-reported. Some “fast

traps” are not measured as a state by microstate accounting as the overhead would be too great. In particular, on UltraSPARC based systems, the memory management unit entries are managed using extremely brief fast traps. The effect of these traps is to elongate the duration of a state as measured by microstate accounting, but to underestimate the proportion of system time. It is possible to measure all these overheads using hardware counters in the UltraSPARC CPU to get accurate cycle counts in each state. Current versions of Solaris do not include a way to obtain these measurements.

## 7.0 SUMMARY

These errors are significant, and may explain why you never seem to be able to scale a workload up as far as you might think from an apparently low usage level to a high one.

This problem gets worse on faster CPUs. In the future, CPU measurement will get less and less accurate. This is not a Solaris 2 specific problem. It is a generic Unix problem that probably also affects other operating systems. Not many operating systems support high resolution measured CPU usage data.

There is not a lot you can do to solve this problem. The sampled data collection is inaccurate, but it is very low overhead. Performance tools that look at per-process CPU usage should use microstate enabled data whenever it is available. Microstate accounting on Solaris 2 also includes a direct per process measurement of CPU scheduling delay, and delays due to paging.

Even on a single system there is no simple calibration that can be applied to correct the errors, as they vary depending upon the workload.

## 8.0 REFERENCES

The SE Performance Toolkit <http://www.sun.com/sun-on-net/performance/se3>

Measurement and Analysis of Process and Workload CPU Times in Unix Environments. Agrawal, Newman, Forsyth, Somin, CMG96.