DRAFT DRAFT DRAFT

A "Displacement" Technique for Robust Portable Measurement of Communications Processing Overheads

C. M. Woodside, Marc Courtois, Cheryl Schramm
Department of Systems & Computer Engineering
Carleton University
1125 Colonel By Drive
Ottawa, Canada K1S 5B6
email: {cmw,courtois,schramm}@sce.carleton.ca

Internal Report,
Copyright Real-time And Distributed Systems (RADS) Lab,
Carleton University,
May 5, 1997

Abstract

The problem of recording CPU demand for communications operations is handled poorly by some instrumentation techniques. A simple technique is described that does not require instrumenting the code (so it can be used without having the source code), and that can be used with any operating system. It records the displacement of a CPU-intensive looping process by the process under test, and computes the CPU effort taken by the process under test. Experiments show that standard UNIX instrumentation misses a large fraction of the CPU time to handle socket messages, and misses more for larger messages. Displacement captures it all, apart from errors due to daemons.

1. Introduction

It is difficult to measure the CPU usage of communications protocols and midware, such as sockets and RPCs. The difficulty is that in many implementations a significant amount of the protocol execution is done by interrupt service routines (ISRs), and the usual kernel instrumentation does not capture all of this. As there is no process switch, the clock ticks which occur during the ISR execution are allocated to the process which was interrupted, not the one for which the messages are being handled. Indeed, it is impossible to know which process is the destination of a message at the moment of the interrupt which begins its handling, because the information is in the header which has not yet been read. So the normal means for accounting for operations by a process fail for ISRs, and as a consequence, for a significant fraction of protocol processing (our experiments indicate that the fraction may be of the order of 50% in some cases!).

McCanne and Torek [1] have documented how ISR execution is not captured by the standard CPU-demand tools in UNIX. They introduced a modification into the 4.4BSD UNIX release which identified an ISR state and counted clock ticks that occurred in that state. This only half solves the problem, since the process is still not identified, and solves it for only one version of one operating system. We would like to have a generic solution that could be applied across as many platforms as possible. The displacement technique appears to be as universal as one could realistically hope for.

The "standard" tools for determining CPU demand are not standard across different versions of UNIX, much less on other platforms such as OS/2 or NT. This work was stimulated by a need to calibrate communications overheads for a family of applications running on a full range of distributed platforms, and running a range of communications midware (sockets, DCE RPCs, MQSeries, CORBA). The communications overheads were an important part of the entire workload, and we wanted a procedure that would apply not only to the present platforms but, if possible, to new ones as they are introduced.

The one measurement that is available in all environments is the “wall-clock” time, or time-of-day, so we searched for a technique that is based entirely on time-of-day measurements. Because a time-of-day utility does not track which process is running, it would make sense to have just one process running, the Process Under Test. However, this does not work for communications utilities, because communicating software often blocks when waiting for input events, so the duration of a scenario is not all execution time.

The displacement technique determines how much of a given time interval is spent running the Process Under Test, by running a second process to fill the gaps. By analogy with Archimedes' Principle, the second process will be called the Fluid Process; the time taken by the Process Under Test is found by the displacement of the execution of the Fluid Process. The displacement technique can be applied in any environment with a wall-clock timer, or even with a stop-watch, to estimate the execution time of an operation involving uninstrumentable parts, such a remote procedure call or socket message interchange. The idea is so simple that we assume it has been used before, however we have not found a published description. This paper therefore describes our version of the idea, analyzes its properties and describes examples.

This paper describes the technique, analyzes the assumptions upon which it depends for accuracy, compares its results with those of standard UNIX calls to determine communications costs, and shows its effectiveness when predictions are made based on the measurements.

2. Displacement Procedure

The Process Under Test (to be named *PUT*) is configured so that it repeats the operation to be measured, for some programmable number L_{PUT} of loops. It is convenient to have one send operation, or one receive operation, or one of each, in one repetition of the loop. The auxiliary Fluid Process (to be named *F*) executes a fairly short loop over some arithmetic operations; the duration of its loop affects the resolution of the measurements to be obtained. Process *F* is configured to run for a given number L_F of loops, and to obtain and print out the wall-clock time at the beginning and the end.

Step 1: The CPU time per loop of the Fluid Process *F* is calibrated by running it alone on a workstation for some large number $L_F(1)$; let the wall-clock time interval be $T_F(1)$ seconds. The estimated CPU time for one loop traversal of *F* is

$$\tau_F = T_F(1)/L_F(1) \text{ sec/loop}$$

Step 2: The two processes *F* and *PUT* are then run together in such a way that the Fluid Process starts a little before and ends a little after the other, as illustrated in Figure 1. The loop counters are set to the values $L_F(2)$ and $L_{PUT}(2)$; it may require some experiments to adjust *LF*(2) to be long enough. The wall-clock time interval for the Fluid Process is recorded as *TF*(2).

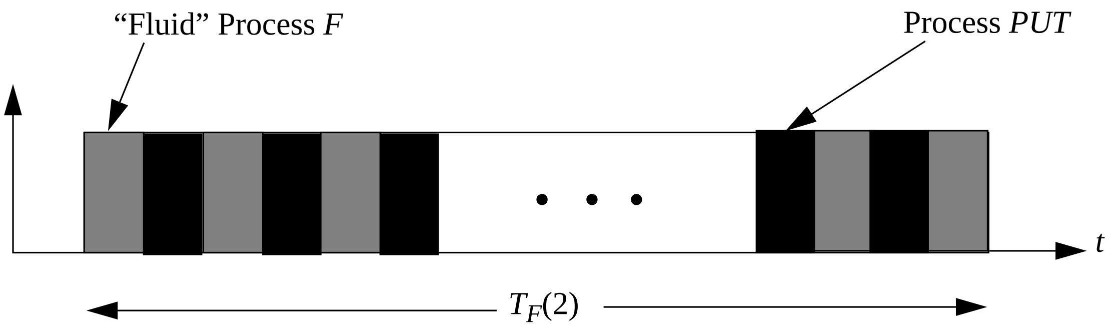

Figure 1. Step 2 Measures How *PUT* Displaces Some of the Time for *F*.

Then the CPU time taken by the Process Under Test during Step 2 is

$$T_{PUT}(2) = T_F(2) - L_F(2)\ \tau_F$$

The second term is the time taken by the Fluid Process, during the interval of length $T_F(2)$, according to the calibration experiment. The estimated CPU cost of *PUT* is then

$$\tau_{PUT} = T_{PUT}(2) / L_{PUT}(2)$$

τ_{PUT} is the CPU cost of executing the code in the loop of the Process under Test.

3. Assumptions and Errors

The procedure above assumes

- that the loop in *PUT* is essentially entirely made up of the operation to be calibrated; loop count overhead can be ignored, as well as overhead in starting and stopping *PUT*.
- that the clock resolution is small,
- that only *F* and *PUT* are running during the two tests,

and that when *F* and *PUT* are run together,

- *F* always runs whenever *PUT* blocks,
- the processes do not interact, so the time per loop of the Fluid Process is the same when it is running alone in the calibration experiment, and with the Process Under Test.

In fact the clock resolution may be significantly large, there are also other processes (e.g. the daemons in UNIX) and the two given processes might interact by context switching or by cache interference. The potential errors can be analyzed as follows.

Daemons

First consider the daemon processes. Suppose that during Step 1 and Step 2, a duration $D(1)$ and $D(2)$ sec is taken, respectively, by daemons. Thus the recorded durations are actually $T_F(1) + D(1)$, and $T_F(2) + H(2)$, and the calculated values for τ are

$$\tau_F = [T_F(1)/L_F(1)] + [D(1)/L_F(1)]$$

$$\tau_{PUT} = \frac{[T_F(1) - L_F(2)T_F(1)/L_F(1)]}{L_{PUT}(2)} + e_{daemons}$$

$$e_{daemons} = \frac{D(2) - D(1)L_F(2)/L_F(1)}{L_{PUT}(2)} \textit{ sec/loop}$$

In the best case the value of $D(i)$ has a fixed ratio α to the loop count $L_F(i)$, (i.e. $D(i) = \alpha L_F(i)$) and $e_{daemons}$ is zero. If the proportion is different in each step, with values $\alpha(1)$ and $\alpha(2)$, then

$$e_{daemons} = [\alpha(2) - \alpha(1)]L_F(2)/(L_{PUT}(2)) \text{ sec/loop.}$$

Context Switching

Context switching will occur naturally due to the operation of the communications software, for instance whenever *PUT* blocks to receive a message. The construction of *PUT* makes this occur once per loop, or $L_{PUT}(2)$ times. The overhead of these context switches however is part of the execution time that we want to estimate, so there is no error in including it. Additional switches due to time-slicing between the two processes may also occur, and the additional overhead is similar to additional daemon execution, in that their number increases with L_F and L_{PUT}.

Cache Interference

Cache interference will occur if, while running one process, the other is displaced from the cache. It is plausible that the arithmetic loop in the Fluid Process is small enough to fit entirely in the cache, but it might be displaced at a context switch. Thus time lost due to cache reloads is also similar to context switching and daemon execution, and is proportional to L_F and L_{PUT}.

Clock Resolution

The effect of clock resolution Δ can be counteracted by running a long enough experiment. Suppose T_F and T_{PUT} are measured with an error range of $\pm\Delta/2$ sec., due to clock resolution Δ. Then τ_F has an error range $(\pm\Delta)/2L_F(1)$ and τ_{PUT} has an error range of $\pm\Delta(1 + L_F(2)/L_F(1))/(2L_{PUT}(2))$. If $L_F(2) \approx L_F(1)$ then the range is roughly

$$\pm e_{res} = \pm\Delta/(L_{PUT}(2)).$$

Overhead of Starting and Stopping

However the launching and termination of the two processes is arranged there will be some overhead distinct from loop repetition, but executed by the process. Suppose it is S sec for each process. The the recorded values are actually $T_F(i)+S$,

$$\tau_F = T_F(1)/L_F(1) + S/L_F(1)$$

$$T_{PUT}(1) + S = T_F(2) + S - L_F(2)\tau_F$$

$$\tau_{PUT} = \frac{T_F(2)}{L_{PUT}(2)} - \frac{T_F(1)L_F(2)}{L_F(1)L_{PUT}(2)} - \frac{S}{L_{PUT}(2)}$$

$$= \textit{calculated value} + e_{start}$$

$$e_{start} = S/(L_{PUT}(2))$$

Overall then, assuming the errors are small, the error in τ_{PUT} is

$$\textit{error} = e_{daemons} + e_{cache} + e_{res} + e_{start}$$

$e_{daemons}$ (including context switching and cache interference) is a constant proportion which we must hope is small; we will attempt to observe it. e_{res} and e_{start} are proportional to $L_{PUT}(2)/L_F(2)$. Thus for small errors, $L_F(2) >> L_{PUT}(2)$.

Robustness

The robustness of the method comes from the robustness of the assumptions.

Several factors which bedevil other techniques for measuring CPU time do not influence the displacement technique. It does not matter what the execution time of the Fluid Process loop is, exactly, so a knowledge of the instructions, the loop overhead, etc. is immaterial. The wall-clock measurements are taken only twice, so their overhead effect is small (and it can also be estimated and subtracted out). The synchronization of measurement instants in the two processes is not necessary, we only require that the Fluid Process covers all the execution time of the Process Under Test.

Consider the cache transient assumption. If execution of one Fluid Process loop out of cache is K times faster than the first time when the cache is being loaded, and each interval of Fluid Process execution (filling in between intervals of the Process Under Test, while it is blocked) requires L_F loops, the fractional error compared to the calibration is $K/(L_F)$. The factor K can be estimated from a knowledge of the processor technology and the number of blocked intervals of the Process Under Test during the run (estimated in turn from L_T).

4. Displacement vs. UNIX Instrumentation

Simple examples will show that standard UNIX instrumentation gives a very different value for socket communications cost, than Displacement, and that when the numbers are used to predict performance Displacement is correct.

4.1. Plain Computation

First we shall check that the two approaches give the same result for a plain CPU-intensive computation. *PUT* was just a spin loop with a CPU demand per loop cycle “operation” which could be approximately specified by the user. The two processes *F* and *PUT* were run on one computer. Process execution times were measured by the Solaris kernel call “ioctl” which provides timing results with small granularity (typically a few microseconds). Experiments were run varying the CPU demand of the *PUT* “operation” from 400 us to 3200 us. Each run used an L_{PUT} of 10000, so the resulting CPU time was divided by 10000 to measure CPU time per operation. (The range of CPU times was chosen to include the CPU times which will be encountered in the next section when we measure TCP/IP send. An L_{PUT} of 10000 was chosen to sufficiently amortize the overhead associated with the displacement method (this is discussed further below). All experiments were performed on a quiet SPARC 2 processor running under Solaris. The times reported by displacement and `ioctl` were almost identical, as show in Figure 2 and Table 1.

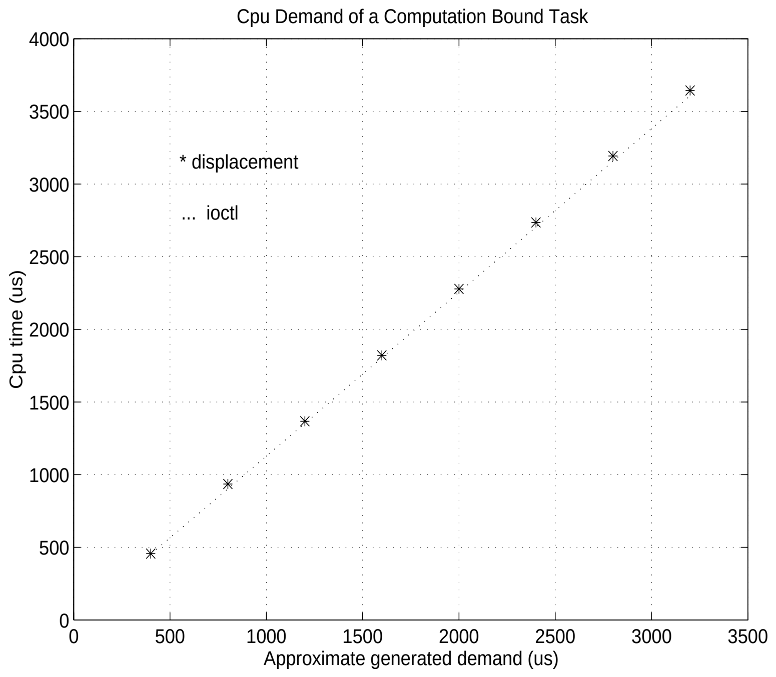

Figure 2. Displacement vs. UNIX Instrumentation `ioctl` on Plain Computation.

Table 1: Comparing on Plain Computation

Approximate CPU time generated(µs)	`Ioctl` measurements (µs)	Displacement measurements (µs)	%age difference
400	454	456	0.41
800	902	936	3.77
1200	1353	1367	1.03
1600	1805	1821	0.89
2000	2256	2278	0.98
2400	2705	2736	1.15
2800	3157	3193	1.14
3200	3607	3644	1.03

The difference between the two measurements, therefore is very small. Displacement is always slightly higher than `ioctl` measurements as expected.

4.2. *PUT* with Communications

When a measurement is made with communications the two approaches disagree substantially. Figure 3 shows the configuration, with two processes *F* and *PUT* on one node, and a Server on the other. The role of the Server is simply to receive messages from *PUT* and send replies back. *PUT* blocks and waits for each reply.

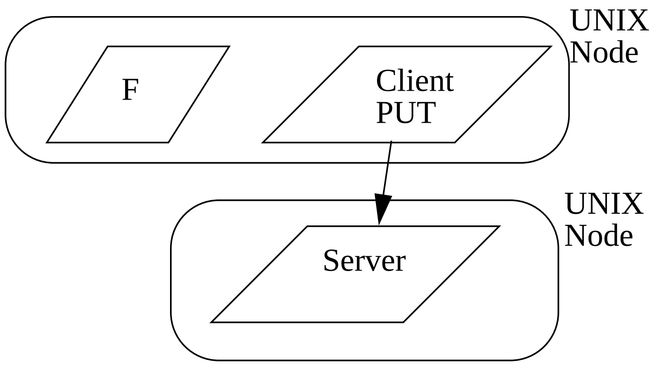

Figure 3.

In this case we measured the CPU time of a TCP/IP send operation, varying the message size from 1K to 8K in steps of 1K. There was also a receive operation, but for a message of roughly zero length. L_{PUT} was set to 10000. Displacement and `ioctl` measurements are plotted in the following graph.

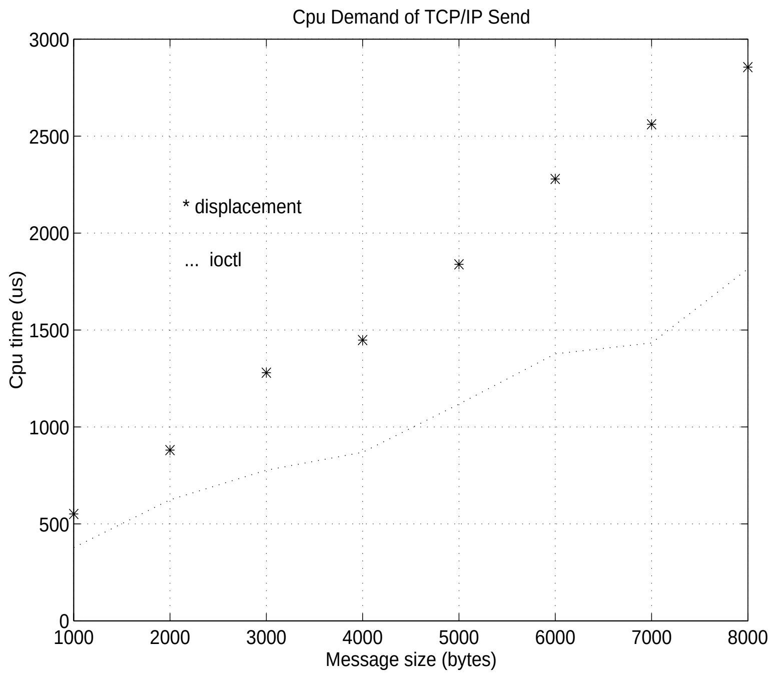

Figure 4. Displacement vs. UNIX Instrumentation on Message Sending.

Notice the systematic difference between displacement and `ioctl`. This discrepancy is due to work performed by Interrupt Service Routines (ISRs) - work which is not all captured by the Solaris (UNIX) instrumentation. The corresponding table of values indicates that at least 40% of the activity on a TCP/IP send operation is not captured by UNIX.

Table 2: Displacement vs. `Ioctl` for TCP/IP Send

Message Size (bytes)	`Ioctl` CPU time (µs)	Displacement CPU time (µs)	%age difference
1000	378	551	45.77
2000	624	915	46.63
3000	777	1280	64.74
4000	869	1448	66.63

Table 2: Displacement vs. `Ioctl` for TCP/IP Send

Message Size (bytes)	`Ioctl` CPU time (μs)	Displacement CPU time (μs)	%age difference
5000	1118	1839	64.49
6000	1377	2279	65.50
7000	1433	2561	78.72
8000	1815	2856	57.36

Table 1 indicated that the displacement method provides similar measurements to `ioctl` on computation bound tasks. Table 2, however, indicates a large discrepancy between the two methods. In order to confirm that displacement was reporting the correct CPU times we performed a simple throughput experiment which is described in the following section.

4.3. Confirmation of Displacement Estimates

A CPU estimate can be used to predict CPU saturation in an appropriate situation. To confirm that the Displacement values are more accurate than the `ioctl` measurements we used both results to make predictions for an experiment in which meassaging contributes to a saturated CPU. In the experiment, a sending process would send 10000 messages of a given size to a receiving process. In order to make sure that the network or the receiving task was not the bottleneck, (i.e. we wanted the throughput to be solely determined by the sending task), each send was followed by a period of pure computation which consumed a precisely known amount of CPU time proportional to the message size. This compute time was generated using a calibrated spin loop and was calculated so as to be slightly larger than the transmission time using 8 Mb/s as the nominal transmission rate with a minimum time of 2 ms. The measured throughput was simply the elapsed time divided by 10000. The predicted throughput was calculated as:

1 / (CPU_time_to_send_message + CPU_time_to_compute)

As demonstrated earlier, the *CPU_time_to_compute* is the same whether you measure it with displacement or with `ioctl`. The debate, however, is centered on *CPU_time_to_send_message*. In our experiment, we recorded the throughput of a sending task using message sizes of 1K to 8K in steps of 1K. Figure 5 compares the measured throughputs to the two estimates.

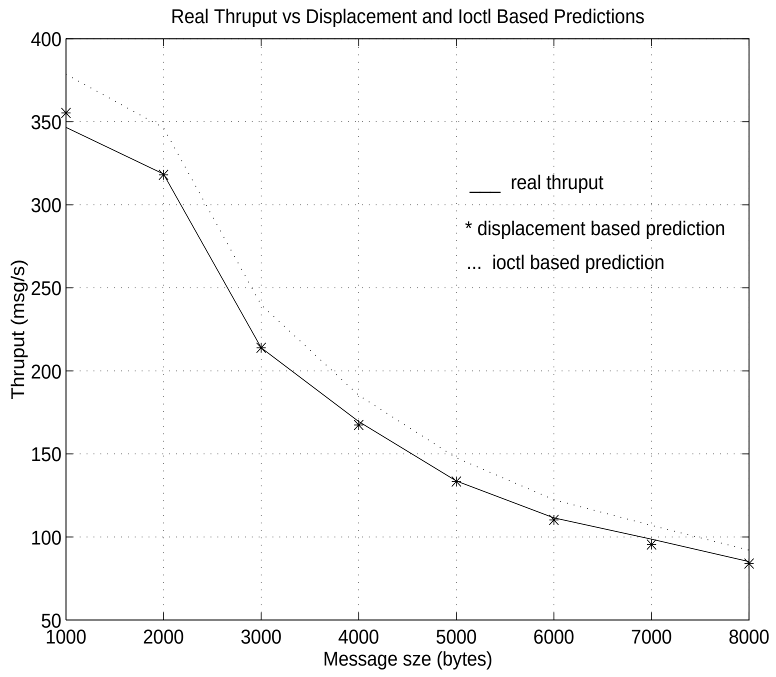

Figure 5. Comparison of Predictions

As can be seen, the displacement-based predictions are almost perfect whereas the `ioctl`-based figures consistently overestimate the throughput. This confirms that `ioctl` is losing some of the execution. The results are summarized in Table 3.

Table 3: Measured vs. Predicted Throughputs

Message Size (bytes)	Real Throughput (msg/s)	Displ-based prediction	%age difference	`Ioctl`-based prediction	%age difference
1000	346.62	355.35	2.52	378.63	9.23
2000	318.52	314.65	-1.21	346.37	8.74
3000	213.87	213.92.	0.02	239.71	12.08
4000	169.51	167.39	-1.25	185.35	9.34
5000	133.70	133.39	-0.23	147.59	10.39
6000	111.47	110.27	-1.07	122.45	9.86
7000	98.67	95.40	-3.32	106.91	8.34

Table 3: Measured vs. Predicted Throughputs

Message Size (bytes)	Real Throughput (msg/s)	Displ-based prediction	%age difference	`Ioctl`-based prediction	%age difference
8000	85.16	83.97	-1.39	92.02	8.06

5. Accuracy

The error analysis in Section 3 identifies errors e_{start} for start-up costs and e_{daemon}, e_{res}, e_{cache} for activities during an experiment (background activity). The effect of background activity is to introduce variance into displacement measurements. The effect of start-up overhead is to introduce a one-time cost. As L_{PUT} increases, the overhead is spread over a larger number of trials thus decreasing the measured CPU time per operation. Eventually the CPU time will reach an asymptotic value where the overhead has virtually no effect. The trick therefore is to find an L_{PUT} which amortizes start-up effects and which provides an acceptable variance.

To get an idea of the effect of L_{PUT} on accuracy, the same send operation as in Section 4.2 was measured with a 2K message and L_{PUT} values of 200, 1000, 5000, 10000 and 15000. For each of these L_{PUT} values, 20 replications were made. Figures 6(a),(b) show how the mean and variance are related to L_{PUT}.

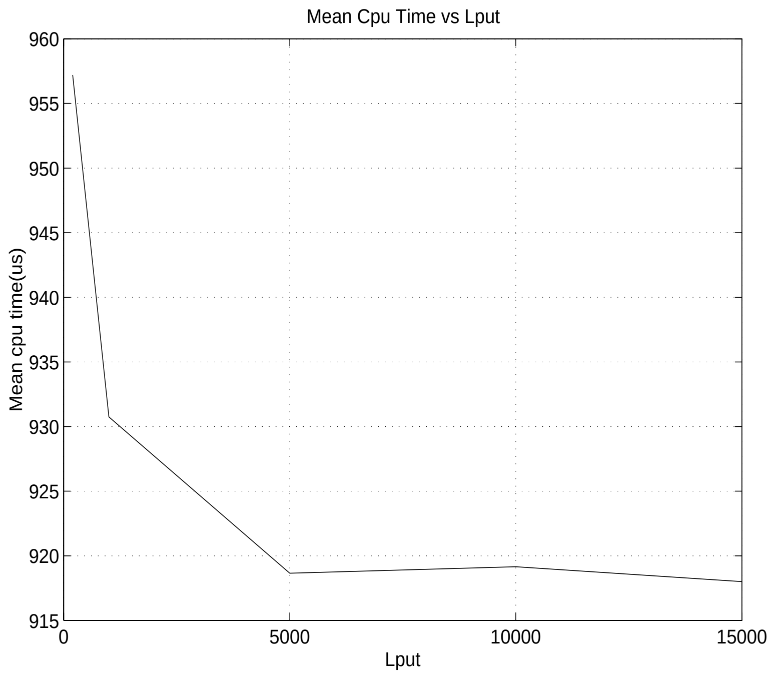

Figure 6(a). Measured Mean of τ_{PUT}

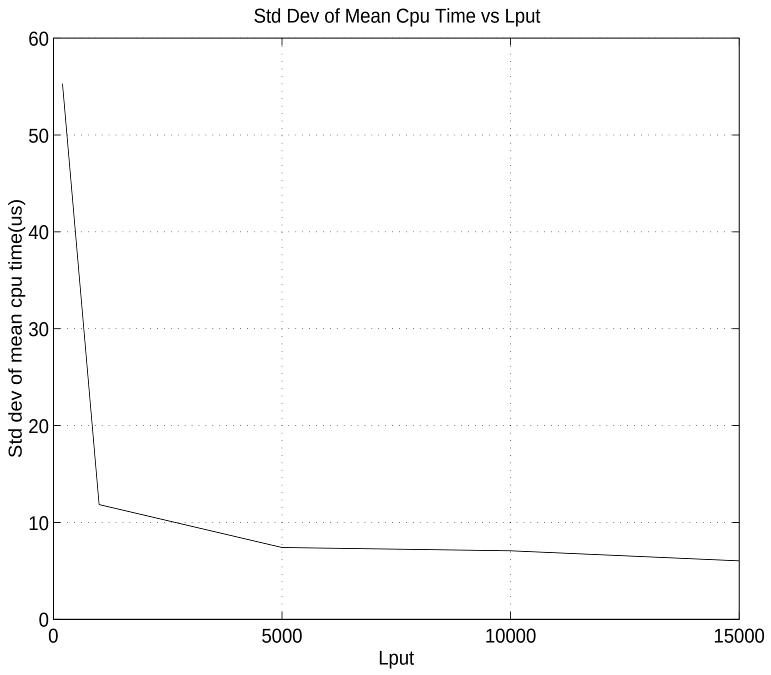

Figure 6(b). Measured Standard Duration of τ_{PUT}

From the graph, the start-up effect is satisfactorily amortized with an L_{PUT} of 5000. At this point the standard deviation is less than 1% of the mean. It is for this reason that we employed 10000 as an L_{PUT} in our previous experiments. The following table summarizes that results and provides a 95% confidence interval for the mean.

Table 4: Impact of L_{PUT} on mean/variance

L_{PUT}	Mean (μs) using 20 replications	StD	StD as a %age of the mean	95% Confidence half interval
200	957.20	55.30	5.77	34.17
1000	930.75	11.84	1.27	7.32
5000	918.65	7.41	0.81	4.58
10000	919.15	7.08	0.77	4.37
15000	918.00	6.03	0.66	3.73

References

S. McCanne, C. Torek, "A Randomized Sampling Clock for CPU Utilization Estimation and Code Profiling", 1993 Winter USENIX Conference, Jan. 1993, San Diego.